

# А.Ф.ШТЕРНБЕРГ Курс программирования микро - калькуляторов

БЗ-34  
МК-52  
МК-54  
МК-56  
МК-61



Л.Ф.ШТЕРНБЕРГ

Курс  
программирования  
микро -  
калькуляторов  
БЗ-34, МК-52, МК-54,  
МК-56, МК-61

-25492-



МОСКВА  
«МАШИНОСТРОЕНИЕ»

1988

Паш

ББК 32.973  
Ш90  
УДК 681.3.06

Рецензент канд. физ.-мат. наук *А.С. Марков*

**Штернберг Л.Ф.**

Ш90 Курс программирования микрокалькуляторов БЗ-34, МК-52, МК-54, МК-56, МК-61. — М.: Машиностроение, 1988. — 208 с.: ил.

ISBN 5-217-00636-6

Изложены теоретические основы и практические приемы работы на программируемых микрокалькуляторах наиболее распространенных типов. Рассмотрены разработка алгоритма решения задачи, его запись в виде программы, ввод ее в память микрокалькулятора, тестирование и отладка программы.

Для широкого круга читателей, желающих освоить работу на микрокалькуляторах указанных типов.

Ш  $\frac{2405000000-613}{038(01)-88}$  48-89

ББК 32.973

ISBN 5-217-00636-6

© Издательство "Машиностроение", 1988

## ПРЕДИСЛОВИЕ

Эта книга адресована тем, для кого программируемый микрокалькулятор (ПМК) стал первой вычислительной машиной, на которой происходит знакомство с программированием.

На какой бы вычислительной технике и на каком бы языке программирования не начинал человек свое знакомство с ЭВМ и с программированием, ему придется ознакомиться со многими общими для этого раздела науки понятиями, освоить технику составления алгоритмов (приобрести так называемое алгоритмическое мышление) и столкнуться с проблемами, не зависящими от марки ЭВМ. Хотя разумеется, конкретная машина требует овладения некоторыми приемами, специфичными именно для нее.

От читателей этой книги не требуется никакой специальной подготовки. Даже там, где встречаются математические термины, они объяснены в тексте. Материал книги представляет собой курс программирования для начинающих. Изложение ведется с современных позиций структурного программирования и основывается на специально предложенном для этих целей языке высокого уровня, который содержит многие конструкции, характерные для современных языков программирования. Освоивший этот курс читатель при переходе к программированию для других ЭВМ обнаружит, что очень многое ему уже знакомо.

Помимо собственно техники программирования в книге освещены вопросы, без которых невозможно грамотное использование ЭВМ: тестирование и отладка программ, их документирование, а также проблемы точности результатов расчетов.

## ВВЕДЕНИЕ

Идея создания вычислительной техники насчитывает тысячелетия: счеты в каком-то смысле — тоже вычислительная техника. В становление этой идеи внесли огромный вклад английские ученые Ч. Бэббидж (1791—1871) и Ада Лавлейс (1815—1852), американский ученый Г. Холлерит (1860—1929) и болгарский ученый Д.В. Атанасов (род. 1903). Но годом рождения первой *действующей* электронной вычислительной машины принято считать 1945 г., когда заработала созданная Джоном Мочли и Джоном Эккертом ЭВМ ЭНИАК, на которой выполняли свои расчеты физики-ядерщики. Итак, электронной вычислительной технике чуть более 40 лет. Но за это время она развивалась в таком темпе, в каком не развивалась ни одна из отраслей техники. Более чем в 1000 раз возросли скорости работы ЭВМ и объемы памяти, при этом машины резко уменьшились в размерах, стали потреблять меньше энергии и существенно повысили свою надежность. В результате вычислительная техника стала активно внедряться в самые разнообразные сферы человеческой деятельности.

Сегодня с вычислительной техникой работают не только научный работник и инженер, но и, например, кассир Аэрофлота, продающий билеты с помощью системы "Сирена". Завтра она станет обычным предметом домашнего обихода. И уже сейчас появилась новая дисциплина в школьном курсе — "Основы информатики и вычислительной техники", знакомящая школьников с принципами работы и использования ЭВМ.

Создание микропроцессоров позволило уменьшить размеры ЭВМ до такой степени, что некоторые ЭВМ стали уменьшаться на столе и даже в кармане. Программируемые микрокалькуляторы (ПМК) как раз и представляют собой такие карманные ЭВМ. Их достоинства очень быстро оценили многие: от студентов и школьников до инженеров и научных работников. На микрокалькуляторе можно выполнить и достаточно серьезный расчет, а можно и поиграть с ним в какую-нибудь игру.

Программируемый микрокалькулятор удовлетворяет всем

требованиям, которые входят в понятие "вычислительная машина", он содержит:

процессор, который может автоматически выполнять заложенную в память программу и обрабатывать данные;

память, способную хранить программу и данные;

устройства ввода и вывода, позволяющие обмениваться информацией между микрокалькулятором и использующим его человеком (ими являются клавиатура и индикатор).

Микрокалькуляторы имеют довольно небольшие скорость работы и емкость памяти, поэтому они годятся только для решения не слишком больших вычислительных задач. Но, как показывает практика, даже в учреждениях, где есть мощные ЭВМ, микрокалькуляторы находят себе применение, так как есть задачи, где ответ на микрокалькуляторе можно получить быстрее, чем дойти до комнаты, где стоит ЭВМ.

Овладение навыками работы на ПМК предполагает знание программирования. При этом человек должен освоить как некоторые общие приемы, характерные для программирования в целом, так и приемы, специфичные для работы на микрокалькуляторах. Следует понимать, что лет через 10 программируемые микрокалькуляторы уступят место более совершенным ЭВМ, которые также будут уместиться в кармане, но иметь гораздо более широкие возможности. При работе на них уже не понадобятся специфичные "калькуляторные" приемы, но общие приемы, наверное, останутся почти без изменений. Поэтому изложение материала этой книги строится по такой схеме: рассматривается общее понятие программирования, а затем — его реализация на микрокалькуляторе.

В настоящее время распространены два семейства: семейство БЗ-21, включающее ПМК БЗ-21, МК-46 и МК-64; и семейство БЗ-34, включающее модели БЗ-34, МК-52, МК-54, МК-56 и МК-61. Все ПМК одного семейства имеют одинаковую систему команд и программа для одного из них с небольшими ограничениями годится и для другого. Свое название семейства получили по первой выпускавшейся модели. Все микрокалькуляторы предназначены только для обработки числовой информации.

Рассмотрим ПМК семейства БЗ-34, поскольку они имеют большее распространение, а также более простую и понятную структуру. Модели БЗ-34, МК-54 и МК-56 абсолютно идентичны по внутренней структуре и различаются только размерами и внешним видом: МК-56 выполняется в настольном варианте и имеет несколько иное расположение клавиш. Кроме того, на БЗ-34 на трех клавишах отличаются надписи. МК-61 и МК-52 — это более совершенные модели: они имеют несколь-

ко дополнительных команд, большую емкость памяти, а МК-52 имеет еще и внешнюю память, которая может хранить информацию после выключения микрокалькулятора. Поэтому любая программа с БЗ-34 и МК-54 может использоваться и на других ПМК этого семейства, но не всякая программа с МК-61 или МК-52 может быть выполнена на БЗ-34 или МК-54. Однако для изучения принципов работы эти отличия совершенно несущественны.

По традиции в литературе по микрокалькуляторам применяют обозначения, имеющиеся на клавишах БЗ-34, но этот ПМК уже снят с производства, поэтому в книге используется символика, имеющаяся на клавиатуре МК-54 (на остальных ПМК типа МК такая же символика).

Читать эту книгу рекомендуется с микрокалькулятором в руках.

# Глава 1. ЗНАКОМСТВО С МИКРОКАЛЬКУЛЯТОРОМ

## 1. СТРУКТУРА МИКРОКАЛЬКУЛЯТОРА И ВЫПОЛНЕНИЕ ВЫЧИСЛЕНИЙ

На лицевой панели микрокалькулятора МК-54 расположены (рис. 1): устройство вывода — индикатор, на котором высвечиваются данные; рычажок включения — с надписью "ВКЛ" рядом с ним; переключатель "Р-ГРД-Г" — "радианы — градусы — градусы" и 30 клавиш с различными надписями на них, над и под ними. Внешний вид МК-61 отличается только некоторыми дополнительными надписями над и под клавишами, а у МК-52 имеется несколько дополнительных клавиш и по-другому оформлен корпус. С назначением клавиш будем знакомиться по мере изучения ПМК.

Внутренняя структура калькулятора — основные блоки, с которыми мы будем иметь дело, показана на рис. 2. Работа всех устройств ПМК идет под управлением *процессора*. Для хранения чисел служат устройства, называемые *регистрами*. На ПМК имеется пять *операционных регистров* (они участвуют в выполнении арифметических операций), обозначаемых буквами X, Y, Z, T и X1, а также 14 адресуемых регистров, пронумерованных цифрами от 0 до 9 и буквами A, B, C, D (A соответствует номеру 10, B — номеру 11 и т.д.). На моделях МК-61 и МК-52 имеется еще 15-й регистр, обозначаемый буквой E. Регистры сокращенно обозначаются буквой R: обозначения RX или R8 соответственно именуют регистр X и ре-

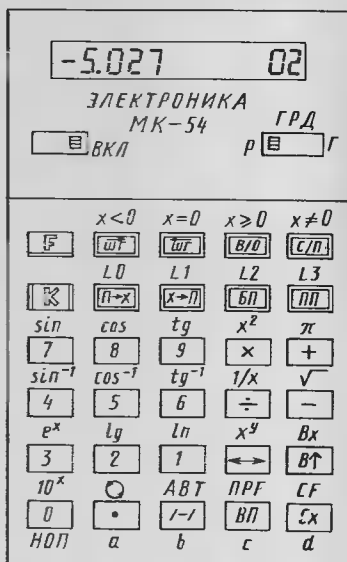


Рис. 1. Внешний вид лицевой панели микрокалькулятора МК-54



гистр 8. Стрелки показывают возможные пути передачи информации между регистрами.

Индикатор, "наложенный" на РХ, позволяет увидеть то число, которое в нем хранится.

Кроме того, имеется *программная память*, состоящая из 98 ячеек (на МК-61 и МК-52 — из 105 ячеек), служащая для хранения программы, и группа специальных регистров: *счетчик адреса команды* и *стек возвратов*, о которых будет рассказано позднее.

Микрокалькулятор может работать от батареек или от сети.

После включения микрокалькулятора на индикаторе высвечивается ноль и микрокалькулятор готов к работе. Он находится в так называемом *режиме вычислений* и с ним можно работать как с непрограммируемым калькулятором.

Набор чисел выполняется в естественном порядке: надо нажать клавиши, на которых записаны нужные цифры. Для отделения целой части от дробной используется клавиша "." (в программировании для этой цели используется не запятая, а точка). По мере нажатия клавиш на индикаторе загораются цифры. После того, как на индикаторе появилось восемь цифр,

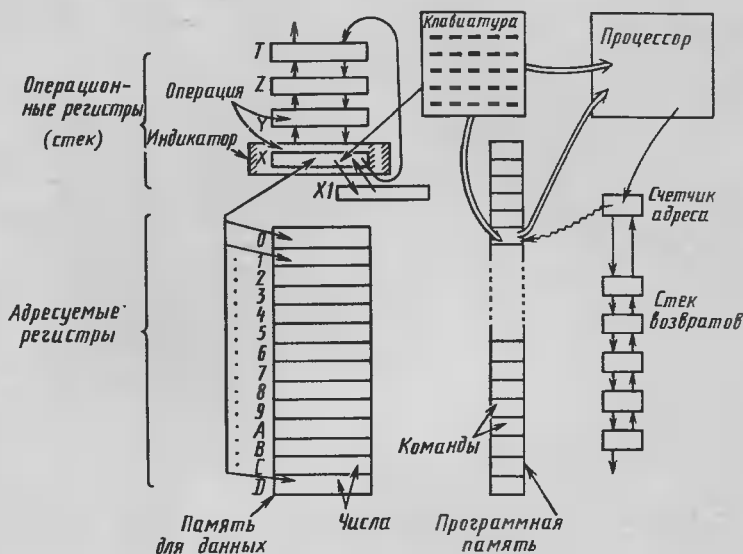


Рис. 2. Внутренняя структура ПМК:

==> передача команд;  
 —> передача данных (чисел и адресов)

на дальнейшие нажатия цифровых клавиш микрокалькулятор не реагирует. Для набора отрицательного числа надо *после набора всех цифр* нажать клавишу  $\text{" / - "}$ , она называется "Изменение знака". Еще одно нажатие этой клавиши опять сменит знак числа.

В научных расчетах часто используют запись чисел с порядком, что позволяет не писать множество нулей: например, вместо 23400000 пишут  $234 \cdot 10^5$  или  $2.34 \cdot 10^7$  (в книге везде вместо десятичной запятой используется десятичная точка, как это принято в программировании) или вместо 0.000567 пишут  $0.567 \cdot 10^{-3}$ . Степень, в которую возводится 10, называется *порядком* числа, а то, что умножается на 10 в какой-то степени, называется *мантиссой*. С точки зрения уменьшения погрешностей (о чем подробно сказано в гл. 6) наиболее выгодно пользоваться *нормализованным* представлением числа, т.е. выбрать степень числа 10 так, чтобы мантисса оказалась между 1 и  $10^*$ . Например,  $2.34 \cdot 10^7$  — это нормализованная запись числа.

Для ввода числа с порядком надо набрать мантиссу числа, которая не обязательно должна быть нормализованной, затем нажать клавишу  $\text{" / - "}$ , если число отрицательное, далее нажать клавишу "ВП" (ввод порядка) — на индикаторе справа загорятся два нуля: т.е. порядок пока нулевой, далее нужно набрать цифры порядка, а если порядок отрицательный, то затем нажать клавишу  $\text{" / - "}$ , чтобы изменить знак порядка.

При ошибке в наборе мантиссы нужно нажать клавишу "Сх" (сброс X) и повторить набор числа. При ошибке в наборе порядка нужно просто правильно нажать клавиши: ошибочные цифры порядка сдвигаются влево и "выдвигаются" из порядка.

Наберем, например, число  $123.45678 \cdot 10^{-3}$ :

Кла- виши	Инди- катор	Комментарий
1	1.	Соответствующие нажатым клавишам цифры загораются на индикаторе.
2	12.	
4	124.	Ошибка — придется сбросить число и начать набор с начала.
Сх.	0.	
1	1.	

\* В математике нормализованным считается представление, при котором мантисса находится в интервале 0.1...1, но для ПМК нам удобнее несколько отклониться от этой традиции.

Кла- виши	Инди- катор	Комментарий
2	12.	Точка смещается на индикаторе и оказывается за последней цифрой
3	123.	
.	123.	Клавиша "." ничего не меняет на индикаторе, но останавливает перемещение точки
4	123.4	
5	123.45	
6	123.456	
7	123.4567	
8	123.45678	Набрана восьмая цифра — далее на нажатие цифровых клавиш микрокалькулятор не реагирует
9	123.45678	
/-/ /-/	-123.45678 123.45678	Клавишей "/-/" можем изменять знак числа сколько угодно раз
ВП	123.45678 00	Переходим к вводу порядка: сначала он нулевой, цифры "вдвигаются" в порядок справа
4	123.45678 04	
0	123.45678 40	Была ошибка — и снова набираем порядок "03"
3	123.45678 03	
/-/ /-/	123.45678-03	В конце меняем знак порядка

Набираемое число заносится в РХ, поэтому мы его видим на индикаторе. Как видно на рис. 2, в остальные регистры число может попасть только из РХ. При пересылке числа всегда пересылается его копия, а в том регистре, откуда выполнялась пересылка, число сохраняется; в том регистре, куда пересылается число, старое содержимое автоматически стирается.

Пересылка в РУ выполняется с помощью клавиши "В↑" (на БЗ-34 — "↑"). Работу с регистрами Z и Т рассмотрим в следующем параграфе. После пересылки в регистр У можно набирать следующее число. Заметим, что при любых действиях с числом (а также при пересылках), оно приводится к стандартному виду по следующим правилам: если при приведении числа к нулевому порядку десятичная точка может разместиться на индикаторе, то число приводится к нулевому порядку и порядок не высвечивается, в противном случае (абсолютное значение числа больше 99999999 или меньше 1) оно приводится к нормализованному виду.

Увидеть содержимое РУ можно с помощью клавиши "↔" (на БЗ-34 — "X<sup>←</sup>Y<sup>→</sup>"), которая обменивает местами содержимое РХ и РУ.

Для выполнения арифметических операций нужно нажать соответствующую клавишу: первое число (слагаемое, уменьшаемое, делимое, множимое) берется из РУ, второе число (слагаемое, вычитаемое, делитель, множитель) — из РХ, результат операции помещается в РХ (значит он виден на индикаторе), содержимое РУ исчезает, а прежнее содержимое РХ копируется в РХ1. Если после выполнения операции начать набор нового числа, то результат предыдущей операции автоматически переходит в РУ, что очень удобно для выполнения цепочечных вычислений.

Клавиши	Индикатор	Комментарии
4 . 2	4.2	
ВП 3 В↑	4.2 4200.	03 Набрано число $4.2 \cdot 10^3$ , при пересылке оно пришло к стандартному виду
0 . 4 2 ↔	0.42 4200.	Набрали новое число Можем посмотреть, что было в РУ. Теперь в РХ — 4200, а в РУ — 0.42
÷	1.	—04 Выполнили деление: содержимое РУ делится на содержимое РХ — получили 0.0001
2 +	2. 2.0001	При наборе нового числа предыдущий результат перешел в РУ, он может использоваться в следующей операции

Элементарные функции для ГМК являются одноместными операциями, т.е. операциями над одним числом. Они выполняются над содержимым РХ, результат также помещается в РХ, прежнее содержимое РХ сохраняется в РХ1, а другие регистры не изменяются.

Для вычисления элементарных функций следует нажать желтую клавишу "F" и ту клавишу, над которой желтым цветом написана нужная функция, в результате срабатывает совмещенная функция клавиши. Например, с помощью клавиш "F" и "8" выполняют вычисление косинуса; записывать, какие клавиши нажимаются, в подобном случае принято "F" "cos".

На микрокалькуляторе МК-54 можно вычислять следующие функции:  $\sin x$ ,  $\cos x$ ,  $\operatorname{tg} x$ ,  $\sqrt{x}$ ,  $1/x$ ,  $\sin^{-1} x$ ,  $\cos^{-1} x$ ,  $\operatorname{tg}^{-1} x$  (последние три функции — это обратные тригонометрические функции  $\arcsin x$ ,  $\arccos x$  и  $\operatorname{arctg} x$  соответственно),  $e^x$ ,  $\lg x$ ,  $\ln x$  и  $10^x$ .

На микрокалькуляторах МК-52 и МК-61 можно вычислить еще ряд полезных функций, которые обозначены на клавишах голубым цветом (третья функция клавиши), это функции

$|x|$ ,  $[x]$  (выделение целой части),  $\{x\}$  — (выделение дробной части) и др. Для их выполнения предварительно нажимают голубую клавишу "К". С помощью клавиши "F" выполняется также операция возведения в степень  $x^y$ , которой пользоваться не рекомендуется: эта операция дает большую погрешность и работает нестандартно — первое число берется из РХ, а не из РУ, значение РУ сохраняется. С той же клавишей выполняются: вызов числа  $\pi$  ("F  $\pi$ ") и засылка в РХ содержимого РХ1 ("F Vx") — при засылках старое содержимое РХ уходит в РУ.

Нажатие клавиш "F" и "К" вызывает срабатывание совмещенной функции только один раз. Если "F" или "К" нажаты ошибочно, то надо нажать клавишу "Сх", ее совмещенная функция — CF (Сброс F).

Вычисление тригонометрических и обратных тригонометрических функций зависит от положения переключателя "Р—ГРД—Г". Если переключатель находится в левом положении ("Р"), то считается, что аргумент тригонометрических и результат обратных тригонометрических функций вычисляется в радианах, если в среднем ("ГРД") — в градусах (прямой угол равен 100 град), если в правом ("Г") — в градусах.

Клавиши	Индикатор	Комментарии
6 0 F cos	60. 5.0000002--01	Наберем некое число, и поставив переключатель в положение "Г", вычислим косинус
F Vx F cos	60. --9.5241269--01	Восстановим аргумент и, переведя переключатель в положение "Р", опять вычислим косинус, получаем другое значение: косинус 60 радиан

Если операцию выполнить невозможно (пытаемся поделить на ноль, взять корень или логарифм от отрицательного числа), а также если результат получается по абсолютному значению больше, чем  $10^{100}$ , то на индикаторе появляется "ЕГГОГ" (от английского *error* — ошибка). При этом, если операция в принципе невыполнима (например, деление на ноль), на РХ остается исходное число (хотя мы его и не видим) и с ним можно работать дальше (а чтобы его увидеть, нужно дважды нажать клавишу " $\leftrightarrow$ ").

Адресуемые регистры предназначены только для хранения чисел, которые могут попасть в них из РХ, и которые могут из них выдаваться на РХ. Запись копии содержимого РХ в адресуемый регистр выполняется нажатием клавиши " $x \rightarrow \Pi$ " (X в память) и цифровой клавиши с нужным номером, а для зада-

ния регистров А, В, С, D и Е служат клавиши ":", "/-/", "ВП", "Сх" и "В↑" соответственно. Символы, обозначающие регистры, записаны белым цветом под ними (регистр Е есть только на МК-61 и МК-52). Вызов копии значения регистра выполняется нажатием клавиши "П → х" и клавиши с номером регистра. Значение из регистра заносится в РХ, старое содержимое РХ переходит в РУ.

Клавиши	Индикатор	Комментарии
2	2.	Набираем какое-нибудь число, и записываем его в РЗ. Число остается и в РХ и с ним можно
х → П 3	2.	выполнять операции
F 1/x	5.	-01 Записываем в РА результат операции
х → П А	5.	-01 Из РЗ читаем то, что ранее записали, а прежнее
П → х 3	2.	содержимое РХ уходит в РУ
↔	5.	-01 Вызванное число может участвовать в опе-
+	2.5	рациях

После включения микрокалькулятора в его регистрах записаны нули. Значения в адресуемых регистрах могут появиться только в результате засылки их из РХ. Значение в РХ может появиться в результате вызова из адресуемого регистра, операции или набора на клавиатуре, и микрокалькулятор эти значения никак не различает — они равно могут участвовать во всех действиях. По-разному работают только клавиши набора числа: если мы нажимаем цифровую клавишу или клавишу ":", а перед этим выполнялась операция или чтение-запись адресуемых регистров, то ПМК считает, что начался набор нового числа, пересылает содержимое РХ в РУ и заносит цифру на индикатор; если же перед этим шел набор числа, то ПМК считает, что очередная цифра продолжает это число и заносит ее на индикатор. При необходимости набрать два числа подряд их разделяют нажатием клавиши "В↑", которая сигнализирует о конце набора первого числа.

Используя рассмотренные возможности ПМК, можно выполнять достаточно длинные расчеты, храня все данные и результаты в памяти ПМК. В качестве примера выполним не очень длинный расчет: по заданным размерам  $d$  и  $h$  (диаметру и высоте) цилиндра вычислим площадь его боковой поверхности (она равна  $\pi dh$ ) и объем ( $\pi d^2 h/4$ ). Пусть значение  $d$  хранится в Р0, а значение  $h$  — в Р1. Для определенности занесем в эти регистры значения 2 и 3 соответственно.

Клавиши	Индикатор	Комментарии
$P \rightarrow x$ 0 $F x^2$	2. 4.	Вызываем из P0 значение $d$ и возводим его в квадрат
$F \pi$ $x$	3.1415926 12.56637	Вызываем значение $\pi$ , $d^2$ переходит в PУ, перемножаем
$P \rightarrow x$ 1	3.	Вызываем значение $h$ , $\pi d^2$ переходит в PУ
$x$ 4 $\div$ $x \rightarrow P$ 2	37.69911 4 9.4247775 9.4247775	Перемножаем, делим на 4, и полученное значение объема записываем в P2
$P \rightarrow x$ 0 $F \pi$ $x$	2. 3.1415926 6.2831852	Еще раз вызываем значение $d$ , умножаем его на $\pi$ ,
$P \rightarrow x$ 1 $x$ $x \rightarrow P$ 3	3. 18.849556 18.849556	еще раз вызываем значение $h$ , и получаем значение площади, которое записываем в P3

Разумеется, если на этом нужные вычисления закончены, то записывать результаты в регистры нет необходимости. Но с записанными в P2 и P3 значениями можем продолжить вычисления.

## 2. ИСПОЛЬЗОВАНИЕ СТЕКА

В ПМК имеются еще два регистра: PZ и PT, которые вместе с PX и PY образуют так называемый *стек* (от английского *stack* — пачка). Конструкция стека такая же, как у магазина стрелкового автомата, куда поместить патрон или откуда взять патрон можно только с одной стороны, при этом остальные патроны сдвигаются, и доставать их можно только в порядке, обратном тому, в котором их закладывали.

Итак, стек ПМК имеет четыре позиции, помещать в него и брать из него можно только через PX. Стек работает при всех операциях и есть несколько специальных команд для работы с ним. Работа стека иллюстрируется диаграммами, приведенными на рис. 3, где слева показаны регистры до операции, справа — после операции, стрелки показывают пересылки информации, кружочки соответствуют операциям над числами.

Занесение нового числа в PX любым способом вызывает сдвиг стека в сторону PT: содержимое PX переходит в PY, содержимое PY — в PZ, а содержимое PZ — в PT, стирая прежнее



Рис. 3. Перемещение информации в стеке

содержимое РТ; однако стек не сдвигается при занесении нового числа, если он уже сдвинут командой "В↑", выполненной непосредственно перед занесением числа, или же содержимое РХ стерто командой "Сх", которая показывает, что новое число должно заменить старое. По диаграммам видно, что при двухместных операциях на место содержимого РУ приходит число, бывшее в РZ а в РZ — число, содержавшееся в РТ, это число остается и в РТ. Одноместные операции работают только с РХ. Последние три команды, показанные на рис. 3, — это чисто стековые операции, их работа частично уже рассматривалась: по диаграммам ясно, как они работают с остальными регистрами. Совсем новой является только команда "F↺" (читается "эф кольцо"), выполняющая кольцевой сдвиг стека. В дальнейшем для удобства полиграфического набора она будет обозначаться "FO".

Чтобы умело пользоваться стеком, надо понять, откуда взялась эта конструкция и почему для нее предусмотрены именно такие операции. Использование стека базируется на интересном математическом построении, изобретенном в 1921 г. польским математиком Яном Лукасевичем и получившем название *обратная польская запись*, или *польская инверсная запись*, сокращенно — ПОЛИЗ.

Мы привыкли размещать знаки двухместных операций между операндами (так называют выражения, над которыми выполняется операция): мы пишем " $a + b$ " читаем это " $a$  сложить с  $b$ ". В ПОЛИЗе знак операции ставится за операндами: " $ab +$ " — такую запись тоже можно прочесть естественным образом — как команду " $a$  и  $b$  сложить". Аналогично знаки одноместных операций (а к ним на микрокалькуляторе относятся и все элементарные функции) будем писать не перед, а после операнда: т.е. вместо " $\sin x$ " запишем " $x \sin$ ". Каждый операнд



может в свою очередь быть сколь угодно сложным выражением. Например:  $(a - b)/(c + e)$  записывается  $ab - ce + /$ ,

где скобки под записью показывают, над какими значениями выполняется операция. В этом примере вычитание выполняется над значениями  $a$  и  $b$ , сложение — над  $c$  и  $e$ , а деление — над результатами вычитания и сложения.

Чтобы понять преимущества польской записи перед обычной, рассмотрим пример более сложного выражения:

$$\ln \sin (a - b/c) + d - 3 \cdot e,$$

где цифры над знаками операций показывают порядок их выполнения. Польская запись этого выражения имеет вид:

$$\underbrace{\underbrace{\underbrace{a \ bc \ /}_{-} \ \sin}_{\ln} \ d + \underbrace{3 \ e \times}_{-}}_{-}$$

Как видим операции выстроились в порядке их выполнения, не нужны ни скобки, ни учет старшинства операций. Если попытаться сформулировать правила вычисления выражения в обычной записи, то они получатся весьма длинными: надо упомянуть и скобки, и старшинство операций, и то, что операции одного старшинства выполняются слева направо для двухместных операций и справа налево для одноместных (см. операции  $\sin$  и  $\ln$  в примере), и привести таблицу старшинства операций. Для выражения же, записанного в ПОЛИЗе, правила его вычисления формулируются в четырех пунктах:

- 1) выражение читается слева направо один раз;
- 2) значения констант и переменных помещаются в стек;
- 3) при операции берется из стека столько значений, сколько нужно для ее выполнения, а результат помещается назад в стек;
- 4) в итоге результат оказывается в вершине (окне) стека, т.е. в той единственной ячейке стека, которая доступна для чтения-записи.

Итак, при одноместной операции (например, "F sin") из стека вынимается одно значение и помещается в PX1 (вот почему оно там оказывается после операции), стек при этом сдвигается. Над содержимым PX1 выполняется операция, результат которой вталкивается в стек: стек сдвигается в обратную сторону, результат оказывается в PX. Схема работы одноместных операций стала понятна. При двухместной операции (например, вычитании) из стека выбирается одно значение и помещается в PX1, стек сдвигается и теперь доступно для рабо-

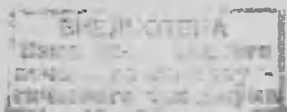
ты содержимое РХ (где находится прежнее содержимое РУ) и РХ1. Затем из стека выбирается второе значение (стек еще раз сдвигается), выполняется операция и результат вдвигается в стек — стек сдвигается в обратную сторону. Стала понятна и схема работы двухместных операций.

Проведем вычисление последнего выражения, следя за содержимым стека. Запись вида " $\Pi \rightarrow x (p)$ " будет означать "Вызов значения из того регистра, где хранится  $p$ ". Просматриваем выражение слева направо и записываем в таблицу прочитанный символ и выполняемые действия, а также состояние стека.

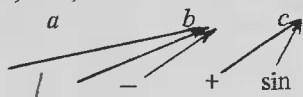
Сим- вол	Действия	Регистры стека				Комментарии
		X	Y	Z	T	
$a$	$\Pi \rightarrow x (a)$	$a$				Значение переменной за- сылается в- стек всегда
$b$	$\Pi \rightarrow x (b)$	$b$	$a$			
$c$	$\Pi \rightarrow x (c)$	$c$	$b$	$a$		
$/$	$\div$	$b/c$	$a$			При двухместных опера- циях из стека берутся два значения и выдается один результат
$-$	$-$	$a - b/c$				
$\sin$	$F \sin$	$\sin(a - \frac{b}{c})$				При одноместных опера- циях и берут и помещают в стек одно число
$\ln$	$F \ln$	$\ln \sin(a - \frac{b}{c})$				
$d$	$\Pi \rightarrow x (d)$	$d$				Переменную -- в стек
$+$		$\dots + d$				Операцию -- выполнить
$3$	$3$	$3$				Константу -- тоже в стек
$l$	$\Pi \rightarrow x (l)$	$l$	$3$			И после двух операций в стеке остается только результат
$\times$	$\times$	$3 \cdot l$				
$+$	$+$	результат				

Итак, стек — это специальное средство для удобного вычисления выражений, записанных в польской записи. Его небольшая длина почти не мешает; как видим, в нашем примере РТ даже не использовался. Остается понять, как по обычной записи выражения получить ПОЛИЗ. В принципе можно сформулировать три простейших правила перевода выражения в ПОЛИЗ:

- 1) записать выражение в одну линию, т.е. без многострочных формул, и выписать константы и переменные в том порядке, в котором они встречаются в такой записи;
- 2) выписать операции в том порядке, в котором они выполняются;
- 3) вставить операции непосредственно после своих операндов.



Например, выражение  $\sin(p - \frac{5}{a+b} + c)$  при записи в  
 линию принимает вид:  $\sin(p - 5/(a+b) + c)$   
 его константы и переменные:  $p \quad 5 \quad a \quad b \quad c$



операции в порядке выполнения: +

Первый плюс относится к  $a$  и  $b$ , он вставлен после них; деле-  
 ние относится к числу 5 и результату операции сложения, его  
 знак вставлен после них и т.д.

Но на практике можно один раз прочесть выражение слева  
 направо и, рассуждая определенным образом, сразу записать  
 оптимальный способ его вычисления, что мы и рассмотрим на  
 примере выражения, приведенного выше. Читаем его символ  
 за символом, пропуская скобки, и записывая нужные для его  
 вычисления действия.

Сим- вол	Действия	Регистры стека					Рассуждения
		X	Y	Z	T	N	
sin							Операцию пока выполнить не мо- жем, пропускаем ее
p	$\Pi \rightarrow x(p)$	p					1 Переменную засы- лаем в стек всегда
-							И эту операцию придется отложить
5	5	5	p				2 Константу тоже за- сылаем в стек
/							Откладываем, так как делить пока не на что
a	$\Pi \rightarrow x(a)$	a	5	p			3 Переменную -- в стек
+							Операцию откла- дываем
b	$\Pi \rightarrow x(b)$	b	a	5	p		4 Переменную -- в стек. И в стеке готовы операнды для отложенной операции сложе- ния
	+	$a + b$	5	p			3 Выполним эту опе- рацию сложения.
	$\div$	$\frac{5}{a + b}$	p				2 Теперь все готово для операции де- ления
	-	$p - \frac{5}{a + b}$					1 Теперь можно вы- читать, а вычис- лять синус еще нельзя

Сим- вол	Действия	Регистры стека					Рассуждения
		X	Y	Z	T	N	
c	$P \rightarrow x(c)$	c	$p - \frac{5}{a+b}$				2 Продолжаем чтение выражения, как и раньше
	$+ \quad p - \frac{5}{a+b} + c$						1 Выполняем отложенные операции
F	sin						1

Фактически здесь выполняется перевод в ПОЛИЗ: константы и переменные выписываются в порядке их появления, операции — в порядке выполнения и сразу же расставляются по местам. Очевидно, что получающаяся последовательность действий оптимальна, так как без засылки значений переменных и констант обойтись нельзя, без операций — тоже, а больше никаких действий нет. В общем случае может потребоваться только операция "V↑", если необходимо заслат в стек две константы подряд. Заметим, что отложенные операции также как бы складываются в стек и выполняются в порядке, обратном тому, в котором их складывали.

Некоторые проблемы при вычислении выражений создает только операция "Fxy", которая на ПМК этого типа реализована некорректно: свой первый операнд она берет из RX, а не из RY, как положено двухместным операциям, и сохраняет в RY значение второго операнда, которое мешает дальнейшей работе. В линейной записи операция "XY" записывается "x↑y", в ПОЛИЗе — "x y↑", а вычислять такой элемент (если он находится в середине выражения) приходится так:

↔, F x<sup>y</sup>, ↔, FO

или же вместо x<sup>y</sup> записать e<sup>y ln x</sup> и выполнять соответствующие действия.

Остается выяснить, как вычислять выражения, для которых четырех регистров стека не хватает. Для этого надо, во-первых, научиться определять такую ситуацию. Рисовать всю таблицу перемещения информации в стеке, как это мы делали выше с чисто иллюстративной целью и как это рекомендуют учебники по ПМК, нет никакой необходимости: при работе по польской записи в стеке все всегда автоматически оказывается на своих местах. Достаточно выписать правый столбец N таблицы, который до сих пор не рассматривался; в нем показано количество занятых регистров стека. Правила его заполнения таковы: занесение значения в стек увеличивает число в этом столбце на единицу, выполнение двухместной операции число уменьшает,

а выполнение одноместной операции не изменяет. И если в столбце появилось число 5, то это значит, что в этот момент будет потерян один из промежуточных результатов. Этот результат и следует записать в адресуемый регистр, а затем вызвать в нужный момент. Можно сформулировать алгоритм оптимальной работы и на этот случай, но он встречается не столь часто и, как правило, с ним проще справиться, преобразовав выражение. Например, выражение  $a \cdot b + \sin(p - \frac{c}{d - e})$  для своего вычисления требует пять ячеек стека, что видно по его польской записи, над элементами которой проставлены те числа, которые мы писали в столбце занятости регистров:

1	2	1	2	3	4	5	4	3	2	2	1
$a$	$b$	$\times$	$p$	$c$	$d$	$e$	$-$	$/$	$-$	$\sin$	$+$

Но достаточно переписать его в виде  $\sin(p - \frac{c}{d - e}) + a \cdot b$  и регистров хватит. А чтобы выражение нельзя было переписать, необходимо иметь в нем в большом числе непрерывных операций, например:  $a \cdot b - \sin(p - \frac{c}{d - e})$ . Но и в этом случае проще записать его в виде:  $-\sin(p - \frac{c}{d - e}) + a \cdot b$  и запрограммировать с помощью одной дополнительной операции  $"/-/"$ .

Как видим, до сих пор не применялись чисто стековые операции  $"FO"$  и  $"\leftrightarrow"$ . Это не случайно: при классической работе со стеком они просто не нужны. Операция  $"\leftrightarrow"$  применяется только при вычислении  $x^y$  потому, что соответствующая операция микрокалькулятора некорректно реализована. Операция  $"FO"$  также потребуется нам в дальнейшем по аналогичной причине.

#### Упражнение

Записать в польской записи выражение

$$4 \times (a + \frac{b}{c}) - \ln(\frac{\sin x}{42 - f}).$$

### 3. ЗАПОМИНАНИЕ И ИСПОЛНЕНИЕ ПРОГРАММЫ

На практике часто встречается необходимость выполнения однотипных расчетов, отличающихся только значениями данных. Например, продавец всегда проводит одинаковый расчет, умно-

жая стоимость единицы товара на его массу. Если нужно провести вычисления типа рассмотренных в предыдущем параграфе для цилиндров с разными диаметрами и высотами, то нужно будет поместить в P0 и P1 другие значения  $d$  и  $h$ , а далее нажимать те же клавиши и в том же порядке и повторить это многократно. Программируемый микрокалькулятор облегчает эту задачу: он может запомнить необходимую последовательность нажатий клавиш, а затем многократно выполнять всю последовательность по нажатию одной лишь клавиши.

Необходимая для данного вычисления последовательность нажатия клавиш называется *программой*. Если говорить более точно, то программа состоит из последовательности команд (под термином "команда" будем понимать комбинацию клавиш, при которой выполняется какое-то законченное действие). Некоторые команды состоят из нажатия одной клавиши, например, "—" — "вычесть"; другие — двух клавиш, например "П→х 5" — "вызвать из P5"; есть команды, состоящие из нажатий трех или четырех клавиш.

Для запоминания программы ПМК имеет программную память, содержащую 98 ячеек (на МК-61 и МК-52 — 105 ячеек), пронумерованных от 0 до 97 (или до 104 соответственно), каждая из которых может хранить одну команду. Так как команды приходится высвечивать на индикаторе, который может показывать только цифры и небольшое количество других знаков, команды запоминаются в памяти в виде двухсимвольного кода: каждой команде соответствует свой код. Коды приведены в табл. 1, где в строках указан 1-й символ кода, в столбцах — второй символ, так что команду с кодом 62 надо искать на пересечении 6-й строки и 2-го столбца — это команда "П→х 2". Пока рассмотрены далеко не все приведенные в таблице команды.

При работе с микрокалькулятором в режиме вычислений, каждая команда немедленно исполняется. Чтобы показать микрокалькулятору, что команды надо не исполнять, а запоминать, его необходимо переключить в режим программирования, что осуществляется нажатием клавиш "F" и "ПРГ". На индикаторе справа появляются два нуля — это высвечивается значение счетчика адреса, который показывает, в какую ячейку будет записана поданная команда. В начальный момент в нем нули, т.е. команда будет записана в ячейку № 00. Теперь нажимаем те же клавиши и в том же порядке, что и при вычислении параметров цилиндра. После каждой команды на индикаторе появляется ее код, а значение счетчика адреса увеличивается на еди-

Первая цифра кода	Вторая цифра кода							
	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	+	-	x	÷	↔	$F10^x$	$Fe^x$	$Flg$
2	$F\pi$	$F\sqrt{\quad}$	$Fx^2$	$F\frac{1}{x}$	$Fx^y$	$FO$	$K\vec{O}^*$	
3	$K0^{'''}*$	$K x *$	$K\text{эн}^*$	$K0^{''}*$	$K x *$	$K\{x\}^*$	$K\max^*$	$K\Delta^*$
4	$x\rightarrow\Pi 0$	$x\rightarrow\Pi 1$	$x\rightarrow\Pi 2$	$x\rightarrow\Pi 3$	$x\rightarrow\Pi 4$	$x\rightarrow\Pi 5$	$x\rightarrow\Pi 6$	$x\rightarrow\Pi 7$
5	$C/\Pi$	$Б\Pi$	$В/О$	$\Pi\Pi$	$KHO\Pi$			$FX\neq 0$
6	$\Pi\rightarrow x 0$	$\Pi\rightarrow x 1$	$\Pi\rightarrow x 2$	$\Pi\rightarrow x 3$	$\Pi\rightarrow x 4$	$\Pi\rightarrow x 5$	$\Pi\rightarrow x 6$	$\Pi\rightarrow x 7$
7	$KX\neq 0 0$	$KX\neq 0 1$	$KX\neq 0 2$	$KX\neq 0 3$	$KX\neq 0 4$	$KX\neq 0 5$	$KX\neq 0 6$	$KX\neq 0 7$
8	$KБ\Pi 0$	$KБ\Pi 1$	$KБ\Pi 2$	$KБ\Pi 3$	$KБ\Pi 4$	$KБ\Pi 5$	$KБ\Pi 6$	$KБ\Pi 7$
9	$KX>0 0$	$KX>0 1$	$KX>0 2$	$KX>0 3$	$KX>0 4$	$KX>0 5$	$KX>0 6$	$KX>0 7$
-	$K\Pi\Pi 0$	$K\Pi\Pi 1$	$K\Pi\Pi 2$	$K\Pi\Pi 3$	$K\Pi\Pi 4$	$K\Pi\Pi 5$	$K\Pi\Pi 6$	$K\Pi\Pi 7$
L	$Kx\rightarrow\Pi 0$	$Kx\rightarrow\Pi 1$	$Kx\rightarrow\Pi 2$	$Kx\rightarrow\Pi 3$	$Kx\rightarrow\Pi 4$	$Kx\rightarrow\Pi 5$	$Kx\rightarrow\Pi 6$	$Kx\rightarrow\Pi 7$
C	$KX<0 0$	$KX<0 1$	$KX<0 2$	$KX<0 3$	$KX<0 4$	$KX<0 5$	$KX<0 6$	$KX<0 7$
Г	$K\Pi\rightarrow x 0$	$K\Pi\rightarrow x 1$	$K\Pi\rightarrow x 2$	$K\Pi\rightarrow x 3$	$K\Pi\rightarrow x 4$	$K\Pi\rightarrow x 5$	$K\Pi\rightarrow x 6$	$K\Pi\rightarrow x 7$
E	$KX=0 0$	$KX=0 1$	$KX=0 2$	$KX=0 3$	$KX=0 4$	$KX=0 5$	$KX=0 6$	$KX=0 7$

F Bx

\* - команда есть только на МК-61 и МК-52.

Таблица 1

							Общая характеристика группы команд
8	9	-	L	C	Г	E	
8	9		/-/	ВП	Cx	B†	Набор числа
Fln	Fsin <sup>-1</sup>	Fcos <sup>-1</sup>	Ftg <sup>-1</sup>	Fsin	Fcos	Ftg	Операции и функции
$\begin{matrix} \rightarrow \\ K 0''^* \end{matrix}$							
KV*	K⊕*	Kинв*	Kсч*				Специальные операции
x→П8	x→П9	x→Па	x→Пb	x→Пc	x→Пd	x→Пе*	Запись в адресный регистр
FL2	Fx>0	FL3	FL1	Fx<0	FL0	Fx=0	Переходы
П→x8	П→x9	П→xa	П→xb	П→xc	П→xd	П→xe*	Чтение из регистров
Kx≠08	Kx≠09	Kx≠0a	Kx≠0b	Kx≠0c	Kx≠0d	Kx≠0e*	Косвенный условный переход
KБП8	KБП9	KБПа	KБПb	KБПc	KБПd	KБПе*	Косвенный безусловный переход
Kx>08	Kx>09	Kx>0a	Kx>0b	Kx>0c	Kx>0d	Kx>0e*	Косвенный условный переход
KПП8	KПП9	KППa	KППb	KППc	KППd	KППe*	Косвенный переход к подпрограмме
Kx→П8	Kx→П9	Kx→Па	Kx→Пb	Kx→Пc	Kx→Пd	Kx→Пе*	Косвенная запись
Kx<08	Kx<09	Kx<0a	Kx<0b	Kx<0c	Kx<0d	Kx<0e*	Косвенный условный переход
KП→x8	KП→x9	KП→xa	KП→xb	KП→xc	KП→xd	KП→xe*	Косвенное чтение
Kx=08	Kx=09	Kx=0a	Kx=0b	Kx=0c	Kx=0d	Kx=0e*	Косвенный условный переход



ницу, причем на индикаторе будут видны коды трех последних команд.

Клавиши	Индикатор	Комментарии
F ПРГ	00	Перешли в режим программирования, в счетчике адреса — ноль.
П → x 0	60 01	Появился код команды "П → x 0", команда записана в ячейку № 00, а содержимое счетчика увеличилось на единицу.
F x <sup>2</sup>	22 60 02	Новый код появляется слева, предыдущие сдвигаются вправо,
F π	20 22 60 03	на индикаторе остаются только три последних кода, а счетчик адреса показывает, в какую ячейку пойдет очередной код.
X	12 20 22 04	
П → x 1	61 12 20 05	
X	12 61 12 06	
4	04 12 61 07	
÷	13 04 12 08	
x → П 2	42 13 04 09	
П → x 0	60 42 13 10	
F x <sup>2</sup>	22 60 42 11	Если мы по ошибке нажали не ту клавишу,
← ПГ	60 42 13 10	что надо, то нажатие клавиши "← ПГ" позволяет вернуться и исправить ошибку.
F π	20 60 42 11	
X	12 20 60 12	Далее продолжаем запоминание команд.
П → x 1	61 12 20 13	
X	12 61 12 14	
x → П 3	43 12 61 15	
С/П	50 43 12 16	Назначение этого действия объяснено ниже.

Нужная последовательность действий запомнена. Вернемся в режим вычислений (который также называется *автономным* режимом): при нажатии клавиш "F", "АВТ" на индикаторе появится то число, которое там было до нажатия клавиш "F", "ПРГ", после чего с ПМК можно продолжать работать в режиме вычислений, но можно и выполнить запомненные действия. Например, в режиме вычислений можно занести в P0 и P1 значения *d* и *h*. Занесем те же значения 2 и 3 (впрочем, если ПМК не выключали после решения примера с цилиндром, то эти значения там и остались).

Для выполнения запомненных действий служит клавиша "ПП" ("пошаговый проход"), по нажатию которой выполняется одна команда из числа записанных в памяти, а именно та, адрес которой (т.е. номер ячейки, где она хранится) содержится в счетчике адреса. После этого значение в счетчике адреса увеличивается на единицу и при следующем нажатии клавиши

"ПП" выполняется следующая команда. На счетчике адреса — значение 16, которое там появилось после запоминания команды "С/П" (то, что оно не видно на индикаторе, роли не играет). Нам же необходимо сначала выполнить команду, начиная с № 00. Для этого нужно занести в счетчик ноль, что осуществляется с помощью клавиши "В/О" ("Ввод ноля") — на индикаторе при этом ничего не меняется.

Содержимое счетчика адреса до нажатия	Нажатая клавиша	Содержимое счетчика адреса после нажатия	Выполненная команда	Индикатор	Комментарий
16	В/О	00	В/О	не меняется	"0" → счетчик
00	ПП	01	П → x 0	2.	Выполняются
01	ПП	02	F x <sup>2</sup>	4.	команды № 00, 01 и т.д.
02	...	...	...	...	

Как видим, на индикаторе появляются те же значения, что и при работе в автономном режиме, — это естественно, поскольку выполняются те же команды, только подаются они не с клавиатуры, а из памяти. Но такой способ выполнения команд применяется в основном для проверки запомненных команд. Для быстрого выполнения всей запомненной последовательности команд можно использовать клавишу "С/П" ("Стоп/Пуск"), при нажатии которой ПМК переходит к работе в автоматическом режиме. В этом режиме после выполнения команды, полученной из памяти, и увеличения значения адреса тут же начинается выполнение следующей команды и т.д. Так продолжается до тех пор, пока не выполнится команда "С/П" в программе, она переводит ПМК обратно в автономный режим (вот для этого она и была поставлена в конце нашей программы).

Для работы в автоматическом режиме также необходимо в счетчике адреса установить адрес той команды, с которой надо начинать выполнение. Нажмем клавишу "В/О", занеся в счетчик ноль, и нажмем клавишу "С/П" после нескольких секунд работы, в течение которых индикатор мигает, ПМК переходит в автономный режим, и на индикаторе появляется число. Теперь, прочтя с помощью команд "П → x 2" и "П → x 3" содержимое P2 и P3, получим результаты. Кстати, в силу случайных особенностей нашей программы в момент окончания ее выполнения второй результат оказывается на индикаторе, но не для всех программ это так.

Теперь можно понять, ради чего была проделана вся наша работа — в результате нее в памяти ПМК имеется программа, по которой можно *легко провести множество однотипных расчетов*. Заносим в P0 и P1 другие исходные данные, нажимаем клавиши "В/О" и "С/П" — и через несколько секунд получаем на P2 и P3 результаты. Эту процедуру можно повторять многократно.

Итак, мы познакомились с основными возможностями ПМК. Он может работать в следующих режимах:

в автономном (режиме вычислений), немедленно исполняя поданные с клавиатуры команды;

в программном (режиме программирования), запоминая подаваемые с клавиатуры команды;

в автоматическом, выполняя всю запомненную последовательность команд.

Отметим, что в руководстве к эксплуатации ПМК автономный режим ошибочно назван автоматическим. Термин "автономный" означает, что в этом режиме ПМК работает независимо (автономно) от программы.

При включении ПМК находится в автономном режиме; переходы из режима в режим выполняются следующим образом:

из автономного в программный — нажатием клавиш "F", "ПРГ";

из программного в автономный — нажатием клавиш "F", "АВТ";

из автономного в автоматический — нажатием клавиши "С/П";

из автоматического в автономный — командой "С/П" в программе или с помощью клавиши "С/П", или в случае некорректной операции по ситуации "ЕГГОГ".

Переход из автоматического режима в автономный с помощью клавиши "С/П" применяется только при необходимости остановить работу ошибочной программы (аварийная остановка): программа прекращает работу в заранее непредсказуемом месте. Если при выполнении какой-либо команды в программе возникло переполнение, деление на ноль или еще что-либо аналогичное, то ПМК также переходит в автономный режим, а на индикаторе загорается сообщение о некорректной операции "ЕГГОГ".

Как мы видели, ПМК неодинаково реагирует на нажатие одних и тех же клавиш при работе в разных режимах. Нужно уметь отличать, в каком режиме работает ПМК, для чего надо знать, следующее:

если индикатор мигает, то ПМК работает в автоматическом режиме;

если на индикаторе есть десятичная точка, то ПМК работает в автономном режиме;

если десятичной точки нет, то ПМК работает в программном режиме.

Если ПМК реагирует на ваши действия не так, как нужно, значит, он находится не в том режиме, и его следует перевести в нужный режим так, как это описано выше.

## Глава 2. ОСНОВНЫЕ ПОНЯТИЯ ПРОГРАММИРОВАНИЯ

### 1. АЛГОРИТМ, ИСПОЛНИТЕЛЬ, ПРОГРАММА

Познакомившись с основными приемами работы на ПМК, необходимо разобраться, какие из них специфичны именно для ПМК, а какие являются общими для всего программирования в целом.

Начнем со знакомства с понятием *алгоритм*. Это понятие первично, и потому его невозможно определить с полной математической строгостью. Интуитивно алгоритм определяется как однозначная последовательность понятных исполнителю предписаний совершить действия, направленные на достижение некоторой цели.

Поясним это примерами. Приказ "сходи в один из ближайших магазинов..." — не алгоритм, так как не ясно, в какой именно магазин надо идти — нет однозначности. А приказ "сходи в магазин № 5, если там нет..., то сходи в магазин..." вполне алгоритмичен. Понятие алгоритма тесно связано с понятием *исполнителя*, т.е. человека или любого аппарата, способного понимать и исполнять указания алгоритма. Каждый алгоритм должен быть рассчитан на конкретного исполнителя. Например, в алгоритм, исполнителем которого является ученик 1-го класса не может входить указание "решить квадратное уравнение...", так как оно непонятно исполнителю. Но такое указание вполне приемлемо в алгоритме, который будет исполнять инженер.

Элементарное действие алгоритма называется *командой*, или *предписанием*. Из сказанного выше ясно, что в алгоритме: каждая команда в отдельности должна быть понятна исполнителю;

выбор исполнителем очередной исполняемой команды должен быть однозначен.

Введем еще одно понятие — *данные*. *Данными* называется то, с чем работает алгоритм. В качестве данных может выступать все, что угодно. Например, при производстве стали в качестве данных выступают: руда, уголь, флюсы, чугуны, кокс, сталь и т.д. Среди многих данных выделяются *исходные данные* (*аргументы алгоритма*) — то, над чем начинает работу алгоритм, и *результатирующие данные* (результаты алгоритма) — то, ради чего исполняется алгоритм. В процессе работы используются также и *промежуточные данные*. Например, при производстве стали исходными данными являются руда, уголь и некоторые другие компоненты; промежуточными данными — кокс и чугуны, которые получаются из исходных данных, но как таковые не нужны, а служат для дальнейшего использования; результатом же является сталь, ради которой и исполняется алгоритм ее производства.

Исполнение алгоритма — это чисто *формальный* процесс, исполнитель не обязан понимать назначение своих действий. Он просто исполняет одно предписание алгоритма за другим. В этом смысле человек — “плохой” исполнитель, так как он часто в исполнение алгоритма привносит свои знания и соображения, которых в алгоритме нет, и не становится в тупик перед командами типа “сходи в один из ближайших магазинов...”. Примером формального исполнителя может служить станок с числовым программным управлением. Знает ли станок, что он обрабатывает, например, ось? Конечно, нет. Он просто выполняет одну за другой команды своей программы: “взять инструмент из гнезда № ...”, “включить подачу” и т.д. И если среди этих команд встречается ошибочная, станок все равно ее выполнит. Именно такие формальные исполнители нас и будут интересовать.

Хороший алгоритм должен обладать рядом свойств. Первое свойство — *массовость*: алгоритм должен давать возможность решать некоторый класс задач, отличающихся исходными данными. Второе свойство — *конечность*: за конечное время алгоритм должен завершать свою работу. Например, алгоритм поиска нужного листа в пачке бумаги, при котором просмотренный лист подкладывается в низ пачки, может работать бесконечно, если нужного листа в пачке вообще нет.

Так же как мысль можно выразить разными словами, алгоритм можно записать разными способами. *Алгоритм, записанный на языке, понятном исполнителю, называется программой для данного исполнителя.*

Программируемый микрокалькулятор вполне может выполнять роль формального исполнителя. Список понятных ему

команд перечислен в табл. 1. Кроме того, имеются еще три команды: "F АВТ", "F ПРГ" и "С/П", переключающие его из режима в режим, а также некоторые специальные команды. Как команды подаются и исполняются, мы уже рассмотрели. Очевидно, что ПМК не понимает, что на нем вычисляют, т.е. является формальным исполнителем. Запись алгоритма на его языке — это последовательность его команд, т.е. введенные в этом и в предыдущем параграфе определения программы не противоречат друг другу. В качестве данных ПМК может использовать только числовую информацию.

А каким способом записывается алгоритм для человека? Конечно, его можно записать на русском (или каком-либо другом) языке, но это не лучший вариант. Нетрудно заметить, что, как только мы внедряемся в какую-то специфическую область человеческой деятельности, мы встречаем и специальную систему записи, позволяющую удобно, кратко и точно выразить то, что нужно. Попытка записать музыкальную мелодию или радиосхему словами даст очень длинную и совершенно нечитаемую запись. Для записи алгоритмов также имеется своя система письма, причем не одна. Например, используются *схемы* алгоритмов, или специальные языки, называемые *алгоритмическими*, где алгоритм записывается в виде текста специального вида. Мы также будем использовать этот способ.

Алгоритмических языков существует много, их подразделяют на несколько классов, и языки одного класса похожи друг на друга. Различия внутри класса связаны частично с различной ориентацией языков, частично с вкусами их авторов. Нам потребуется алгоритмический язык, в котором легко можно будет обнаружить знакомые черты таких известных языков, как Паскаль, ПЛ/1; Русского Алгоритмического Языка из школьного учебника по информатике и др. Но требуемый нам язык предназначен для написания алгоритмов для определенного семейства ПМК, поэтому он должен иметь ряд специфических средств. Он должен содержать только средства для обработки числовой информации, так как ПМК может обрабатывать только ее. Этот язык назовем *А-языком*. Его конструкции, как мы увидим, достаточно естественны с точки зрения обычного русского языка. Обучившись составлять на нем алгоритмы, человек без труда обучится написанию алгоритмов (программированию) на любом из родственных языков. Эта группа языков называется *языками высокого уровня*.

Как мы видели, язык микрокалькулятора мало похож на естественный русский язык или математическую запись.

Он является типичным представителем так называемых *языков низкого уровня*, к которым относятся языки различных ЭВМ (напомним, что ПМК — это тоже ЭВМ) и так называемые *автокоды (ассемблеры)*, в которых команды ЭВМ записываются в чуть более удобном для человека виде. Запись программы в виде последовательности кодов из табл. 1 — это и есть программа на машинном языке, т.е. то, что непосредственно понимает ЭВМ. А запись в виде таких команд, как "P → x 0", "F x<sup>2</sup>" — это и есть типичный автокод, который ПМК автоматически переводит в язык ЭВМ.

При работе на больших ЭВМ достаточно написать программу на языке высокого уровня. ЭВМ сама переводит ее в свой код (как она это делает, для нас неважно), а исполняет уже программу, написанную в своем коде. Этот процесс перевода с языка высокого уровня в код ЭВМ называется *трансляцией* (по-английски *translate* — переводить). В первом приближении можно считать, что большие ЭВМ просто понимают языки высокого уровня. При работе на микрокалькуляторах перевод программы с А-языка, на котором мы предварительно будем записывать алгоритмы, в команды ПМК придется делать вручную.

Итак, для решения задачи на ЭВМ нужно:

написать алгоритм решения задачи на языке высокого уровня (этап необходим и для больших ЭВМ, и для ПМК);

перевести его на язык ЭВМ (выполнение такого процесса вручную — это специфика работы на ПМК);

ввести программу в память ЭВМ;

задать конкретные исходные данные;

выполнить программу;

прочитать результаты.

Последние четыре пункта на разных ЭВМ выполняются по-разному (и у ПМК здесь есть свои особенности), но суть дела от этого не меняется. Таким образом, единственным сугубо специфичным для ПМК этапом является ручной перевод алгоритма на язык команд ПМК.

Элементы А-языка будем изучать по мере необходимости, решая все более сложные задачи. Соответственно будут даны и правила их перевода на язык команд ПМК.

## 2. ВВОД, ВЫВОД, ПЕРЕРАБОТКА ДАННЫХ

Для изучения основ А-языка вернемся к примеру с вычислением площади боковой поверхности и объема цилиндра.

Любое описание чего бы то ни было обычно начинается с каким-то заголовком и завершается признаком конца. Заголовок алгоритма имеет вид

алг название;

где в качестве названия может выступать любой текст, а завершаться описание алгоритма будет словом

кон

(сокращение от слова "конец"). Такие слова, смысл которых в А-языке фиксирован, называют *служебными*. В печатных текстах их записывают полужирным шрифтом, в рукописных — подчеркивают. Между алг и кон записывают собственно те действия, которые алгоритмом предусматриваются для исполнения; они называются *телом алгоритма*. Итак, общий вид записи алгоритма

алг название;

действия (тело алгоритма)

кон

Наши алгоритмы будут выполнять обработку объектов, которые в математике носят название *переменных величин*. Так же как и для любой работы полезно сначала подготовить рабочее место, в описании алгоритма следует сначала перечислить те переменные, с которыми будет идти работа. В отличие от математики, где переменные обозначаются, как правило, одной буквой, в программировании они обозначаются *именами*, которые могут быть любой длины, но должны начинаться с буквы и состоять из букв и цифр без знаков препинания и знаков операций. Для тех переменных, с которыми мы имели дело в рассмотренном примере, выберем имена: *d*, *h*, *объем* и *площ*. При перечислении переменных указывают класс значений, которые могут принимать эти переменные. Упомянутые переменные могут принимать числовые значения, поэтому их следует перечислить с *описателем числ* (впоследствии понадобятся переменные и с другими описателями, или, как говорят, переменные *другого типа*). Сам алгоритм назовем *цилиндр*. Тогда можно записать

алг цилиндр;

числ *d*, *h*, *объем*, *площ*;

собственно действия по вычислению объема и площади

кон

В математике действия по вычислению площади записываются формулой  $S = \pi \cdot d \cdot h$ , но в этой записи неясно, во-пер-



вых, что дано, а что вычисляется, во-вторых, с какими именно значениями следует проводить вычисления. Запрос машиной конкретных значений исходных данных называется вводом и записывается так:

**ввод список переменных;**

где *список переменных* состоит из разделенных запятыми имен переменных. В нашем примере следует запросить значения диаметра и высоты, т.е. переменных  $d$  и  $h$ :

**ввод  $d, h$ ;**

Получив значения  $d$  и  $h$ , можно приступить к переработке данных: по значениям  $d$  и  $h$  вычислить значения переменных *объем* и *площ.* Действие, в результате которого переменная получает значение, в программировании называется *присваиванием* и записывается так:

*переменная* := *выражение*;

или

*выражение* → *переменная*;

Значение выражения вычисляется, и переменная получает вычисленное значение, ее прежнее значение при этом теряется. Знаки " := " и " → " читаются "присвоить". Для программистов более привычна первая форма записи, но для последующего перевода на язык ПМК более удобна вторая форма. Мы будем пользоваться обеими формами. Для нашего примера необходимы такие присваивания

$площ := d \cdot \pi \cdot h$ ;  $объем := d^2 \cdot \pi \cdot h / 4$ ;

которые можно записать в любом порядке.

Наконец, значения переменных, которые являются целью исполнения алгоритма, должны быть сообщены человеку. Этот процесс в программировании называется *выводом*, и в общем случае записывается

**вывод список выражений;**

где *список выражений* включает одно или несколько выражений, разделенных запятыми. Перечисленные выражения вычисляются и их значения сообщаются человеку любым доступным исполнителем способом: печатаются на бумаге, высвечиваются на индикаторе и т.д. Выражение может состоять и из одной переменной, тогда его значение равно значению этой переменной. Для нашего примера нужно написать

**вывод *объем, площ.*;**

Собрав вместе все элементы, получим полную запись:

алг *цилиндр*;

числ  $d, h$ , объем, площ;

ввод  $d, h$ ;

объем :=  $d^2 \cdot \pi \cdot h / 4$ ; площ :=  $d \cdot \pi \cdot h$ ;

вывод объем, площ;

кон

Как видим, алгоритм состоит из последовательности команд, каждая из которых завершается точкой с запятой. Чтобы не путать команды А-языка с командами микрокалькулятора, для команд А-языка будем применять общепрограммистский термин — *оператор*: оператор ввода, оператор присваивания и т.д.

Проследим за исполнением этого алгоритма. Исполнитель (ЭВМ) читает операторы и производит соответствующие действия. Заголовок (оператор алг) действий не требует. По описаниям переменных (оператор числ) ЭВМ *распределяет память*: из имеющейся у нее памяти для хранения данных отбирается нужное количество ячеек, пригодных для хранения чисел, и отмечается, где что будет храниться. Переменную величину можно представить себе как коробку, внутри которой лежит значение, а снаружи наклеен ярлык — имя этой переменной. Итак, по оператору числ будет отведено четыре ячейки, на которые будут "наклеены ярлыки":  $d, h$ , объем, площ. По оператору ввод будут запрошены два значения, первое из которых будет помещено в ячейку с ярлыком  $d$ , второе в ячейку с ярлыком  $h$ . Далее по оператору присваивания ЭВМ возьмем копии значений  $d$  и  $h$  (переменные  $d$  и  $h$  сохраняют свои значения), значения констант  $\pi$  и 4, и вычислит новое значение, которое будет помещено в переменную объем. Аналогично получит значение переменная площ. Далее по оператору вывод ЭВМ даст возможность увидеть значения переменных объем и площ. И по оператору кон завершит исполнение алгоритма.

Рассмотрим теперь, как должен быть реализован этот алгоритм на ПМК. Переменной величине А-языка на ПМК могут соответствовать только регистры, именно они могут принимать различные значения. Для хранения значений придется брать адресуемые регистры, так как операционные регистры нужны для выполнения операций. Для хранения чисел пригодны все регистры, поэтому брать их будем подряд, начиная от R0. Ввод и вывод — это операции, которые выполняются вручную; ПМК должен приостановить выполнение программы и дать человеку возможность записать или прочитать значения. С учетом сказанного выше составляем программу.

Адрес	Код	Команда	А-язык	Комментарий
			алг цилиндр;	Заголовок действий не требует
$d$	— P0	числ $d$ ,		По описаниям переменных составляем таблицу распределения памяти: какой регистр соответствует каждой переменной
$h$	— P1	$h$ ,		
объем	— P2	объем,		
плоч	— P3	плоч;		
			ввод $d, h$ ;	Эта операция ручная. ПМК еще не начал работу, так что и команды "С/П" не требуется
00	60	П → x 0		Вычисление выражения выполняем по правилам ПОЛИЗа, когда при чтении выражения встречаем переменную, то по таблице распределения памяти определяем соответствующий ей регистр и пишем нужную команду "П → x".
01	22	F $x^2$		
02	20	F $\pi$		
03	12	x		
04	61	П → x 1		
05	12	x		
06	04	4	/	По той же таблице определяем регистр для записи результата
07	13	÷	4	
08	42	x → П 2	→ объем;	
09	60	П → x 0	$d$	Аналогично действуем для второго присваивания
10	20	F $\pi$	•	
11	12	x	$\pi$	Теперь видно, почему для перевода на язык ПМК присваивание удобнее записывать в виде "... → ..."
12	61	П → x 1	•	
13	12	x	$h$	
14	43	x → П 3	→ плоч;	
15	50	С/П	вывод объем, плоч;	ПМК останавливается, чтобы дать прочесть результаты
			кон	Останавливать его второй раз уже не надо, так как он был остановлен для вывода

Как видим, получилась в точности та программа, которая была рассмотрена в п. 1 и 3 гл. 1. Но эта программа далеко не лучшая по многим показателям. Программирование характерно тем, что даже простейшую задачу можно решить множеством разных способов, в чем можно убедиться на этом примере.

Во-первых, можно сэкономить несколько операций, если записать присваивания так:

$$\text{плоч} := d \cdot \pi \cdot h; \text{объем} := \text{плоч} \cdot d / 4;$$

Тогда для ПМК потребуется такая последовательность команд:

$$\begin{array}{llllll} \Pi \rightarrow x 0 & F \pi & x & \Pi \rightarrow x 1 & x & x \rightarrow \Pi 3 \\ \Pi \rightarrow x 3 & \Pi \rightarrow x 1 & x & 4 & \div & x \rightarrow \Pi 2 \quad \text{С/П} \end{array}$$

Программа сократилась примерно на 20 %, но здесь явно видна избыточность команды " $\Pi \rightarrow x 3$ ": ведь после засылки значения в R3 предыдущей командой оно остается и в RX. Чтобы отразить эту особенность микрокалькулятора, разрешим выполнять *присваивания внутри выражения*, тогда присваивания в программе можно записать так:

или  $\text{объем} := (\text{плоч} := d \cdot \pi \cdot h) \cdot d/4;$   
 $(d \cdot \pi \cdot h \rightarrow \text{плоч}) \cdot d/4 \rightarrow \text{объем};$

Вторая запись лучше, так как легко читается слева направо: вычисленное значение  $d \cdot \pi \cdot h$  заслать в переменную *плоч*, а затем умножить на  $d$ , поделить на 4 и заслать результат в *объем*.

Теперь займемся вводом и выводом. Пока они выполнялись вручную. При вводе набранное число заносилось в нужный регистр, для чего необходимо помнить, в каком регистре, что хранится. То же можно сказать и о выводе. Такой способ ввода-вывода назовем *R-вводом* (*регистровый* ввод, или *ручной* ввод) и соответственно *R-выводом*. Но этот процесс можно несколько автоматизировать, ограничившись ручной работой только с RX, а остальное поручив микрокалькулятору.

Заставим микрокалькулятор для ввода значений остановить выполнение программы, подождать набора значения, а при дальнейшем запуске заслать это значение в нужный регистр; затем остановиться, подождать набора следующего значения и т.д. Аналогично поступим при выводе — только при остановке значение не набирают, а читают с индикатора.

Запрограммировать ввод значений  $d$  и  $h$  можно так:

Адрес	Команда	А-язык	Комментарий
00	$x \rightarrow \Pi 0$	ввод $d$ , С/П	Перед началом выполнения программы на регистре индикатора набираем значение $d$ . Затем нажимаем клавишу "С/П"
01	С/П		
02	$x \rightarrow \Pi 1$	$h$	Содержимое RX, где в данный момент находится значение $h$ , будет записано в R1; далее начнут выполняться команды вычисления
03	...	...	

Команда "С/П" переводит ПМК в режим вычислений, а в счетчике адреса при этом остается номер следующей команды — в данном примере 02. Так как со счетчиком адреса никаких действий не проводили, то это значение там сохраняется. По

второму нажатию клавиши "С/П" будут выполняться команды, начиная с той, адрес которой записан в счетчике, т.е. с команды по адресу 02. Аналогично поступим с выводом. После 11 рассмотренных выше команд вычисления нужных значений пишем:

Адрес	Команда	А-язык	Комментарий
14	П → x 2	вывод	Вызываем значение на индикатор,
15	С/П	объем,	и останавливаем работу,
16	П → x 3	площ;	вызываем второе значение
17	С/П		и опять останавливаем работу
18		кон	

Здесь остановки служат для того, чтобы прочитать показания с индикатора. Дальнейший пуск производится клавишей "С/П".

Такой способ ввода-вывода называется *С/П-вводом* и *С/П-выводом*. Программа несколько удлиняется, но ею становится удобнее пользоваться. Именно таким вводом-выводом, как правило, мы будем пользоваться, а при необходимости применения Р-ввода или Р-вывода в записи на А-языке будем писать служебные слова *Рввод* и *Рвывод*.

Теперь обратим внимание на то, что после ввода на РХ остается значение  $h$  и ничто не мешает использовать его в расчетах. Собственно, тогда для  $h$  и регистр отводить не надо, так как  $h$  участвует в вычислениях только однажды. Значение  $d$  после ввода оказывается в РУ и тоже может быть непосредственно использовано, но оно нам потребуется еще раз, так что запомнить в регистре его нужно. Наконец, вспомним, что значения *площ* и *объем* также оказываются в РХ в определенные моменты и было бы удобно прочитать их именно в эти моменты, а не запоминать в регистрах с целью последующего вызова в РХ. Чтобы описать это на А-языке, разрешим использовать конструкцию *ввод* и *вывод* внутри выражений. Запись

$$\text{площ} := (\text{ввод } d) \cdot \pi \cdot \text{ввод};$$

означает, что введенное значение запоминается в регистре, отведенном переменной  $d$ , тут же умножается на  $\pi$ , а затем умножается на следующее введенное значение, причем второе введенное значение предварительно нигде не запоминается. Аналогично запись

$$\text{вывод} (\text{вывод} (d \cdot \pi \cdot h) \rightarrow \text{площ}) \cdot d/4;$$

означает: значение  $d \cdot \pi \cdot h$  вывести на индикатор, т.е. приоста-

новить работу, чтобы дать возможность человеку его прочитать, затем заслать в переменную *площ*, умножить на  $d/4$  и результат снова вывести на индикатор, не запоминая ни в каком адресуемом регистре.

Если целью нашего алгоритма является только вычисление объема и площади, а в дальнейших вычислениях они не участвуют, то нет необходимости запоминать их в регистрах. Если предположить, что нам безразлично, что вычислять вначале — площадь или объем, то можно написать второй вариант алгоритма так:

алг цилиндр 2:

числ  $d$ ;

вывод (вывод (ввод  $d \cdot \pi \cdot \text{ввод}$ )  $\cdot d/4$ );

кон

Этот алгоритм дает нам оптимальную программу.

Адрес	Команда	А-язык	Комментарий
алг цилиндр 2:			
$d \rightarrow R0$		числ $d$ ;	Осталась всего одна переменная
00 $x \rightarrow P0$		ввод $d$	Перед пуском набираем значение $d$ , которое запоминается в $R0$ ,
01 $R \pi$		.	Умножаем на $\pi$ .
02 $x$		$\pi$	
03 $C/P$		$\cdot$ ввод	При этом останове набираем значение $h$ , которое умножается на предыдущий результат
04 $x$			
05 $C/P$		$\rightarrow$ вывод	Еще останов — теперь для считывания.
06 $P \rightarrow x0$		.	Выведенный результат продолжает использоваться в последующих вычислениях
07 $x$		$d$	
08 4		$/4$	
09 $\div$			
10 $C/P$		$\rightarrow$ вывод;	Последний останов — для считывания второго результата
кон			

Программа сократилась в 1,5 раза по сравнению с первым вариантом и ею стало очень просто пользоваться: о ее внутреннем строении не надо знать ничего. Инструкция по ее использованию такова: набрать значение диаметра, нажать клавишу "C/P", при останове набрать значение высоты, нажать клавишу "C/P", при останове прочитать значение площади, нажать клавишу "C/P", при следующем останове прочитать с индикатора значения объема.

Но сам алгоритм стал менее наглядным: по его тексту трудно догадаться, что и зачем делается. Чтобы устранить этот недостаток введем в наш А-язык еще один элемент — *комментарий*. Комментарий — это произвольный текст, взятый в фигурные скобки, который может размещаться в любом месте текста на А-языке. Этот текст никак не переводится в команды ПМК — он служит только для облегчения понимания читателем текста алгоритма. Разумеется комментарий должен быть таким, чтобы пояснять содержание алгоритма. Тогда наш алгоритм можно записать так:

```

алг цилиндр 2;
  числ  $d$ ;
  вывод {объем=}
  (вывод {площадь=} (ввод  $d \cdot \pi \cdot$  ввод  $\{h\} \cdot d/4$ ));
кон

```

Здесь комментарии после слов **вывод** поясняют, какие именно значения выводятся, а  $\{h\}$  после слова **ввод** — какое значение вводится.

Итак, мы видели, что по алгоритму на А-языке можно написать и самую оптимальную программу, если алгоритм составлен должным образом. Сразу скажем, что написать алгоритм с первого подхода наилучшим образом удастся редко. Обычно сначала пишут простейший вариант типа алгоритма *цилиндр*, а затем его постепенно улучшают, как это мы и делали, в результате чего может изменяться порядок действий, состав обрабатываемых переменных и т.д. Когда же программист считает, что его алгоритм достаточно хорош, можно перевести его в команды микрокалькулятора, что уже не требует умственных усилий и выполняется по чисто формальным правилам, который мы и сформулируем.

1. Заголовок алгоритма в команды ПМК не переводится.
2. По оператору описания переменных составляется таблица распределения памяти, причем регистры отводятся подряд, начиная с  $R_0$ , в порядке перечисления переменных.
3. Оператору **Рввод** соответствует команда " $C/P$ ", если он не является первым после описаний переменных оператором.
4. Оператор **Рвывод** переводится в команду " $C/P$ ".
5. Оператор **ввод** программируется последовательностью команд " $x \rightarrow P$ " с номерами регистров, соответствующих перечисленным в этом операторе переменным, перед каждой из команд " $x \rightarrow P$ " ставится команда " $C/P$ ". Первая команда

"С/П" не ставится, если ввод стоит непосредственно после описаний.

6. Оператор **вывод** программируется как вычисление перечисленных в нем выражений, после вычисления каждого выражения ставится команда "С/П".

7. Операции **ввод** и **вывод**, стоящие внутри выражений, программируются по команде "С/П". Если после команды "С/П", соответствующей вводу, нужны команды набора числовой константы, то после "С/П" необходимо добавить команду "В↑" (иначе команды набора будут дописывать цифры к введенному значению).

8. Оператор **присваивания** программируется как вычисление выражения и засылка значения в регистр, соответствующий указанной в этом операторе переменной.

9. Программирование любого выражения (в операторах присваивания, вывода и т.д.) выполняется так: замена в польской записи выражения переменных командами "П → х" с номером отведенного этой переменной регистра, а операций — соответствующими командами ПМК дает оптимальную программу вычисления этого выражения. Если при этом встречается набор двух констант подряд, то их надо разделить командой "В↑".

10. Оператор **кон** программируется командой "С/П", если непосредственно перед ним не стоит **вывод** или **Рвывод**, иначе ему не соответствуют в программе для ПМК никакие команды (в последнем случае трудно точно описать, чему соответствует конечная команда "С/П" — оператору **кон** или **вывод**).

**Задачи для самостоятельного решения.** При работе с задачами для самостоятельного решения следует помнить, что даже самая простая программа может быть написана разными способами, поэтому несоответствие алгоритмов с приведенными в конце книги в указаниях к решению не является признаком ошибочности вашего решения. Критерием правильности программы является выдача ею правильных результатов с несколькими вариантами исходных данных, подобранных так, что по ним результат можно получить вручную, без ПМК. К указаниям следует обращаться, если алгоритм не получается, а также для сравнения выполненных решений с приведенными в указаниях с целью выяснения качества своего решения. Программы для ПМК пишутся по алгоритму практически однозначно.

1. Труба имеет внутренний диаметр  $d_1$  и внешний  $d_2$  (в метрах), плотность материала трубы  $\rho$   $T/m^3$ . Определить ее сечение и массу погонного метра трубы.

2. Время задано в часах, минутах и секундах. Перевести его в секунды.



### 3. ФОРМЫ ЗАПИСИ ПРОГРАММЫ

На протяжении предыдущих параграфов мы встречались с разными формами записи программы: в столбец, в строку, с кодами команд и без них и т.д. Это связано с тем, что одного наилучшего для всех применений варианта нет и в зависимости от назначения записи используют и различные формы.

Если мы хотим разобраться в том, как работает программа, то сделать это по ее командам значительно сложнее, чем по описанию реализуемого ею алгоритма на А-языке. Поэтому в таком случае алгоритм на А-языке (или на другом языке высокого уровня либо в виде схем) необходим, возможно, еще и дополненный словесными комментариями. Для удобства чтения алгоритма на А-языке его следует записать по определенным правилам: программисты их называют *"запись лесенкой"* — группируя в одну строку несколько присваиваний, родственных по своим функциям, сдвигая вправо на несколько позиций тело алгоритма, чтобы выделить начало и конец алгоритмов (что важно при записи нескольких алгоритмов подряд). Далее эти правила будут уточняться по мере введения новых конструкций А-языка.

Собственно команды программы можно записывать в строку или столбец, с кодами или без них, с указанием адресов или без них. Коды нужны только при вводе программы в микрокалькулятор для визуального контроля ввода, выписываются они по таблице кодов после того, как программа написана. Это сможет сделать любой человек без труда, поэтому, как правило, мы будем выписывать программу без кодов. Запись в строку экономнее записи в столбец, но менее удобна для восприятия, поэтому ее применяют только для записи готовой программы. Наличие адреса у каждой команды удобно при вводе программы в микрокалькулятор и необходимо в некоторых случаях при составлении программы (эти случаи описаны в последующих параграфах). Как правило, достаточно знать начальный адрес программы или учитывать, что программа всегда размещается с нулевого адреса. Поэтому при записи программы в строку иногда указывают только начальный адрес и для удобства размещают по 10 команд в строке: тогда легко определить адрес каждой команды. Такую запись можно часто увидеть в справочниках.

Нам надо будет проследить процесс написания программы. Поэтому будем записывать программу в столбец и с адресами, а рядом размещать текст на А-языке. Такой способ, дополнен-

ный отчеркиванием групп команд, соответствующих какому-то элементу текста на А-языке, позволяет легко понять функции каждой группы команд. Читателям рекомендуется всегда записывать программы по такой схеме — в четыре столбца "Адрес—Код—Команда—А-язык". Здесь, кстати, видно преимущество А-языка перед схемами: схему не нарисуеть рядом с программой так, чтобы было ясно соответствие между ее элементами и программой (точнее, нарисовать-то можно, но тогда она окажется не более понятной, чем сама программа).

Из методических соображений, а также с целью экономии места мы будем в основном применять слегка видоизмененную систему записи: в одной строке будут часто размещаться несколько команд ПМК, выполняющих какое-то законченное с точки зрения алгоритма действие или соответствующих одному оператору А-языка, при этом в столбце "Адрес" будет размещаться адрес первой из команд строки, а коды записывать не будем. Такая запись наиболее удобна для понимания программы. Например:

Адрес	Команды	А-язык	Комментарий
00	$x \rightarrow П 0$	ввод $a$ ,	Ввод значения $a$ реализуется одной
01	$C/П \ x \rightarrow П 1$	$b$ ;	командой, а $b$ — двумя
03	2 3 5 $x \rightarrow П 3$	$c := 235$ ;	А здесь четыре команды соответствуют
			одному оператору присваивания

Заметим, что используемый в книге столбец "Комментарий" введен для пояснения процесса составления программы, при составлении собственно программы он программисту не нужен, равно как и для разбора чужой программы человеком, который программирование знает.

Суммируя сказанное выше, будем применять следующие формы записи. Анализ программы всегда будет начинаться с алгоритма на А-языке, выписанного отдельно по всем правилам записи "лесенкой". В некоторых случаях, чтобы не распылять внимание будет записываться только тело алгоритма, иногда даже без описаний и операторов ввода-вывода (т.е. "действующая часть" алгоритма — она-то нередко и называется телом алгоритма). По сформулированным в предыдущем параграфе правилам (которые будут расширены в последующих параграфах) программа для ПМК получается по алгоритму однозначно, поэтому в ряде случаев обсуждения алгоритма достаточно. Если же нам потребуется программа, чтобы провести *машинный эксперимент*, т.е. провести по ней вычисления, не вникая в про-

цесс ее составления, то вслед за алгоритмом будет записываться программа в строку без всяких комментариев к ней, при этом, если не оговорено противное, программа пишется с нулевого адреса, а регистры распределяются по порядку описания переменных, начиная с нулевого.

Но чаще после разбора алгоритма будет обсуждаться и процесс составления программы. В этих случаях программа будет записываться в виде таблицы "Адрес—Код—Команда—А—язык" (возможно, без столбца "Код"), а помещенный рядом с таблицей комментарий будет пояснять действия по составлению программы: комментарий относится к тем строкам программы, рядом с которыми он размещен.

Отдельно выписанный алгоритм на А-языке плюс таблица "Адрес—Код—Команда—А-язык" составляют существенную часть документации программы. Подробнее об этом сказано в гл. 5, здесь же отметим только, что в таком виде наиболее удобно хранить описания программ, так как по ним легко и вспомнить свою старую программу, и разобраться в чужой.

#### 4. СИНТАКСИЧЕСКАЯ И СМЫСЛОВАЯ ПРАВИЛЬНОСТЬ ПРОГРАММЫ

Написание и алгоритмов, и программ, как и любая другая человеческая деятельность, подвержено ошибкам. Процессы поиска и устранения ошибок (так называемые *тестирование* и *отладка* программ) требуют специального обсуждения, которое мы отложим до гл. 5, но с первых же шагов необходимо понять, что может и чего не может распознать исполнитель (в частности, ЭВМ или микрокалькулятор) в составленной Вами программе.

Все ошибки при составлении алгоритмов (программ) можно разделить на три группы:

*синтаксические ошибки* — алгоритм записан с нарушением правил записи, и исполнитель его не понимает; в этом случае исполнитель каким-то способом должен сообщить, что и где ему неясно — программисты говорят в таких случаях: выдается *синтаксическая диагностика*, общий смысл таких диагностик — "НЕ ПОНИМАЮ";

*смысловые ошибки 1-го рода* — правильно записанный алгоритм содержит операцию, которая по каким-либо причинам выполнена быть не может, в этом случае исполнитель выдает *смысловую диагностику*, общий смысл такой диагностики — "НЕ МОГУ ВЫПОЛНИТЬ";

*смысловые ошибки 2-го рода* — алгоритм записан правильно с точки зрения исполнителя, все его команды исполнимы, но в итоге получается не тот результат, который предполагался.

Поясним на примере. Пусть исполнителем является Ваш сын, которому задается алгоритм "Принеси ПМК со стола в комнате". Если он не знает сокращения "ПМК", то в ответ вы получите "синтаксическую диагностику": "Не понял: что принести?". Задаем алгоритм на понятном исполнителю языке: "Принеси микрокалькулятор со стола в комнате". Этот алгоритм правилен и может быть успешно выполнен, но не всегда — это зависит от "исходных данных": Вы можете получить и смысловую диагностику "его там нет" — это ошибка 1-го рода, так как исполнитель не смог выполнить понятную ему команду, а может появиться и ошибка 2-го рода. Вам принесут микрокалькулятор со стола и сын будет считать, что он алгоритм выполнил, но там лежал непрограммируемый микрокалькулятор, а тот, что был вам нужен, лежит в шкафу.

Теперь рассмотрим правильность алгоритмов на А-языке. Когда текст на А-языке или другом подобном языке читает ЭВМ, то требуется scrupulous выполнение всех синтаксических правил: одна пропущенная точка с запятой — и вы тут же получаете диагностику "не понимаю". Но поскольку тексты на А-языке пока предназначаются для чтения человеком, на синтаксических ошибках останавливаться не будем, так как пропущенную точку с запятой человек скорее всего просто не заметит.

Для анализа смысловой правильности возьмем в качестве примера наш старый алгоритм вычисления площади боковой поверхности и объема цилиндра по его диаметру и высоте и запишем его в виде

```
алг конус;  
  числ  $d, h$ , площ, объем, мыш, кролик;  
  ввод  $d, h$ ;  
   $площ := \pi \cdot d^2 \cdot h / 4$ ;  $объем := \pi \cdot d \cdot h$ ;  
   $мыш := площ - объем$ ;  
  вывод  $площ, объем$ ;
```

кон

Правилен ли этот алгоритм? Первое впечатление, что написана какая-то чушь: почему алгоритм назван *конус*; почему перепутаны формулы площади и объема, что за странное значение получает переменная с именем *мыш*, зачем нужна переменная с именем *кролик*, которая вообще не используется? Однако попробуем разобраться подробнее.

Первый вопрос: синтаксически алгоритм правилен? Да, так как все операторы записаны правильно, исполнителю все понятно. Правильно ли это по смыслу? Алгоритм называется *конус*, но знает ли исполнитель, что вычисляет этот алгоритм? Если мы имеем дело с *формальным* исполнителем, то не знает, а значит, ему безразлично, правильно ли отражает название алгоритма его суть. Далее исполнитель запрашивает у нас два числа, производит с ними некоторые действия и выдает нам два других числа. Для того, кто собирается воспользоваться алгоритмом, нет необходимости знать детали его работы, достаточно изучить инструкцию по его использованию. Для нашего алгоритма эта инструкция такова: алгоритм называется *конус*; ему надо задать сначала значение диаметра, затем — высоты цилиндра; алгоритм выдает сначала значение объема, затем значение площади. Итак, мы получили два числа, которые ожидали? Да. Первое число — значение объема, а второе — значение площади? Да. Следовательно, алгоритм делает все, что нужно, а значит — он абсолютно правильный! А тот факт, что исполнитель называет площадь объемом и наоборот, нас не интересует. Что касается действий с переменной *мышь*, то они бесполезны, но получению результатов *не мешают*. То же относится и к переменной *кролик*: если повар приготовил кастрюлю, которой затем не воспользовался, то это вряд ли скажется на вкусе блюда.

Чтобы лучше понять сказанное выше, представим себе, что кто-то реализовал этот алгоритм на микрокалькуляторе, занес программу в память ПМК и дал вам ПМК с инструкцией: набрать значение  $d$ , нажать клавишу "С/П", набрать  $h$ , нажать "С/П", при останове прочитать значение объема, опять нажать "С/П", прочитать значение площади. Понятно, что с этой программой можно работать, так и не узнав, как назывался алгоритм, как обозначались переменные и то, что в таблице распределения памяти у того, кто писал программу, была строка "*кролик* — P5".

Но, разумеется, рассмотренный алгоритм написан плохо, так как в нем тяжело разбираться: имена переменных сбивают с толку. Поэтому впредь будем стремиться писать алгоритмы не только правильные, но и оптимальные по другим критериям: с *мнемоничными* именами переменных (т.е. такими, которые способствуют пониманию того, что обозначает это имя), без лишних действий, наиболее быстро выполняемые или наиболее короткие и т.д.

Чтобы понимать, что является правильным и что ошибочным с точки зрения микрокалькулятора, надо познакомиться более

подробно с его особенностями; это мы и сделаем на примере программы, в которой по значениям длины сторон треугольника  $a$ ,  $b$  и  $c$  вычисляется косинус угла, лежащего против стороны  $a$  по формуле  $\cos \alpha = (b^2 + c^2 - a^2) / (2 \cdot b \cdot c)$ .

Тело простейшего варианта алгоритма имеет вид

числ  $a, b, c$ , кос;

ввод  $a, b, c$ ;

кос :=  $(b^2 + c^2 - a^2) / (2 \cdot b \cdot c)$ ;

вывод кос;

С учетом того, что результат нужен только для вывода и что значение  $a$  используется только один раз, алгоритм можно записать

алг косинус;

числ  $b, c$ ;

вывод  $(-(\text{ввод } \{a\})^2 + (\text{ввод } b)^2 + (\text{ввод } c)^2) / (2 \cdot b \cdot c)$ ;

кон

В результате программирования алгоритма по нашим правилам получим следующую программу.

Адрес	Команды	А-язык	Комментарий
00	F x <sup>2</sup> /-/	(вывод $(-(\text{ввод } \{a\})^2 +$	
02	C/П x → П 0 F x <sup>2</sup> +	(ввод $b)^2 +$	Здесь вводимые значения
06	C/П x → П 1 F x <sup>2</sup> +	(ввод $c)^2$ )	сразу после запоминания ис-
10	2 П → x 0 x	/ $(2 \cdot b$	пользуются в операциях,
13	П → x 1 x ÷	• c)	результат высвечивается на
16	C/П	;	индикаторе

Теперь начнем проводить эксперименты с ПМК, допуская разнообразные ошибки. Включаем ПМК (если до этого он был включен, то выключите его и через 10 с — раньше нельзя — включите вновь, чтобы работать на "чистом" микрокалькуляторе, ни в регистрах, ни в программной памяти которого нет "остатков" предыдущей работы).

Клавиши	Индикатор	Комментарии
F ПРГ	0.	После включения везде нули
	00	Переходим в режим программирования
F x <sup>2</sup>	22	01 Вводим команды программы
/-/	0L 22	02
C/П	50 0L 22	03
x → П 0	40 50 0L	04

Клавиши	Индикатор	Комментарии
К х <sup>2</sup>	28 40 50 05	Ошибка (вместо "F" нажали "K"), которую
+	10 28 40 06	"не заметим", и продолжим ввод
С/П	50 10 28 07	
х → П 2	42 50 10 08	Еще ошибка (нажали не тот номер регистра), которую тоже "не заметим"
F х <sup>2</sup>	22 42 50 09	
+	10 22 42 10	
3	03 10 22 11	Еще ошибка
П → х 0	60 03 10 12	
х	12 60 03 13	И в этот момент замечаем ошибочный код 03
←ШГ	60 03 10 12	Клавиша "←ШГ" нам знакома, только на этот
←ШГ	03 10 22 11	раз ее придется нажать несколько раз, пока
←ШГ	10 22 42 10	ошибочный код не исчезнет с индикатора
2	02 10 22 11	Теперь набираем правильную команду
→ШГ	60 02 10 12	Клавишей "→ШГ" возвращаемся на прежнее
→ШГ	12 60 02 13	место: на индикаторе появляются ранее набранные коды
П → х 1	61 12 60 14	Завершаем ввод программы
÷	13 61 12 15	и допускаем еще одну ошибку
÷	13 13 61 16	
С/П	50 13 13 17	

Итак, программа введена в память с четырьмя ошибками, причем в одном случае мы ввели несуществующую команду с кодом 28 (такой команды нет на МК-54 и БЗ-34, такая команда есть на МК-61 и МК-52), при этом ПМК никак на ошибки не отреагировал. Вывод: для ПМК нет понятия "синтаксическая ошибка" — ему понятна любая последовательность команд, которая оказывается в программной памяти. Поэтому проверять программы приходится вручную, что можно сделать двумя способами с помощью клавиш "→ШГ" и "←ШГ", которые в дальнейшем для удобства полиграфического набора будем записывать "ШГп" и "ШГл" соответственно (шаг вправо и шаг влево).

Действие этих клавиш заключается в том, что "ШГл" уменьшает, а "ШГп" увеличивает на единицу значение счетчика адреса. В режиме программирования индикатор работает как окно,

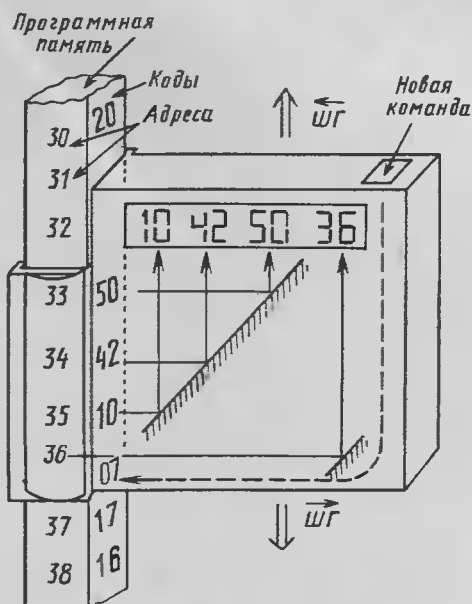


Рис. 4. Иллюстрация работы индикатора в программном режиме

сквозь которое видны три ячейки программной памяти с номерами, предшествующими значению счетчика адреса. Изменяя значение в счетчике адреса, мы "катаем" это окно вдоль программной памяти (рис. 4).

Первый способ проверки: от текущей позиции двигаемся назад по программе:

ШГл	13 13 61 16	
ШГл	13 61 12 15	Обнаружив ошибочный код, ждем, пока он уйдет с индикатора.
ШГл	61 12 60 14	
х	12 61 12 15	Нажимаем клавиши правильной команды,
ШГл	61 12 60 14	и продолжаем проверку.
...	...	Прочие ошибки можно удалить аналогично, но мы их пока "не заметим".
ШГл	0L 22 02	
ШГл	22 01	Дальше уже можно и не идти.

При таком способе приходится нажимать меньше клавиш, но программу мы читаем от конца к началу, что не всем по вкусу. Второй способ проверки:



FAVT	0.		Выходим в режим вычислений,
V/O	0.		заносим ноль в счетчик адреса,
F ПРГ		00	и возвращаемся в режим программирования
ШГп	22	01	А теперь клавишей "ШГп" двигаем "окно".
ШГп	0L 22	02	индикатор вдоль программы и читаем
...	...		коды
...	...		Также сделаем вид, что не заметили две ошибки
ШГп	61 12 60	14	
ШГп	13 61 12	15	Когда появляется ошибочный код, делаем шаг
ШГл	61 12 60	14	назад, уводя его с индикатора.
х	12 61 12	15	Вводим правильную команду,
ШГп	13 12 61	16	и продолжаем проверку.
ШГп	50 13 12	17	
ШГп	00 50 13	18	А если попытаться пройти дальше, то там, куда
ШГп	00 00 50	19	мы ничего не заносили, увидим нули.

Проверка закончена, мы предполагаем, что все правильно. Посмотрим, как ПМК будет выполнять такую программу. Клавишей "ПП" программу проверять не будем, попробуем ее выполнить сразу в автоматическом режиме. Возьмем  $a=3$ ,  $b=4$ ,  $c=5$  (так называемый египетский треугольник, для которого нужный нам косинус равен 0.8).

FAVT	0.	Выходим в режим вычислений,
V/O	0.	подготавливаем программу к работе.
3	3.	Набираем значение $a$ ,
С/П	-9.	При останове на индикаторе значение $-a^2$ ,
4	4.	набираем значение $b$ ,
С/П	ЕГГОГ	и через 5с счета получаем сигнал ошибки "ЕГГОГ"

Вот это и есть единственная смысловая диагностика ПМК: "не могу выполнить команду". Что же ПМК не смог выполнить?

F ПРГ | 10 28 40 06 |

Перейдем в режим программирования.

В счетчике адреса — 06, значит ПМК приготовился к выполнению команды по адресу 06, т.е. последняя выполненная команда — это команда по адресу 05 — ее код 10 виден в левом углу индикатора. Этот прием — переход в режим программирования — обычен для выяснения последней выполненной команды. Но нынешние модели имеют один конструктивный недостаток (точнее, ошибку в конструкции): в некоторых случаях при возникновении сигнала ошибки ПМК лишний раз увеличивает счетчик адреса, поэтому надо проверять не только

последнюю, но и предпоследнюю команды. А там видим код 28 — код несуществующей команды (этот эксперимент пройдет только на МК-54 и БЗ-34, у МК-61 и МК-52 этот код соответствует команде). Реакция ПМК на несуществующие команды ясна. Исправляем ошибку.

ШГл	28 40 50 05	Как обычно, убираем ошибочный код с индикатора.
ШГл	40 50 0L 04	
F x <sup>2</sup>	22 40 50 05	Вводим правильную команду, и не читая программу дальше, переходом в автономный режим.
F АВТ	4.	

Заметим, что в РХ то значение, которое там и было до того, как возникла ошибка (ЕГГОГ). Продолжаем эксперименты:

В/О	4.	Готовим программу к выполнению. Опять набираем значение $a$ .
3	3.	
С/П	-9.	После останова набираем значение $b$ , при останове на индикаторе значние $b^2 - a^2$ . Набираем значение $c$ , ждем результата, но опять получаем ошибку.
4	4.	
С/П	7.	
5	5.	
С/П	ЕГГОГ	

F ПРГ	50 13 12 17	Тот же прием, что и ранее: в режиме программирования смотрим последнюю команду.
-------	-------------	---

Понятно, что команда "С/П" вызвать ошибку ("ЕГГОГ") не могла, значит ошибку дала команда деления:

F АВТ	0.	В РХ находится ноль, т.е. пытались поделить на ноль
-------	----	---

Поскольку мы вносили ошибки умышленно, то причина ясна: значение  $c$  попало в Р2, а делим мы на содержимое Р1, умноженное на что-то. Как легко убедиться, при нажатии клавиш "П→х 1" в Р1 появляется ноль. На вопрос: "Неужели ПМК не видит, что там ничего нет?" Ответ прост: "Конечно, не видит. Потому что там не ничего, а ноль, т.е. вполне правильное число, которое может участвовать почти во всех операциях". Таким образом, ПМК не может заметить отсутствие значения. Исправляем ошибку:

F ПРГ	50 13 12 17	Отходим до места ошибки и вводим правильную команду.
ШГл	10 раз	
х → П1	41 50 10 08	

F АВТ	0.	Эти действия нам уже знакомы.
В/О		

3 С/П	-9.		Аналогично повторяем ввод данных,
4 С/П	7.		
5 С/П	8.	-01	и получаем результат.
<hr/>			
В/О	8.	-01	Еще раз готовим программу к выполнению,
<hr/>			
3 С/П	-9.		значение $a$ взяли такое же,
С/П	81.8		и нажимаем клавишу "С/П", забыв набрать $b$ ,
5 С/П	-1.1866666		но результат получим все равно.

Получили значение косинуса, по абсолютному значению больше единицы, чего быть не может. Что же произошло? Когда мы набираем значение  $b$ , значение  $-a^2$  переходит в РУ. Поскольку мы это значение не набрали, то ПМК с содержимым РХ стал работать как со значением  $b$ , в РУ ничего не перешло, а еще при наборе  $a$  туда попал результат предыдущего расчета, но ПМК этого, конечно, понять не мог, он просто работал с содержимым регистров.

Продолжаем делать ошибки. Выполним программу с  $a = 1$ ,  $b = 3$ ,  $c = 9$ . Результат — 1.6481481, косинус получился больше единицы. Дело в том, что треугольника с такими сторонами быть не может, но ПМК не знает, что он вычисляет косинус: он просто берет  $a$ , возводит в квадрат, меняет знак и т.д. — что при этом получится, то и выводится на индикатор. И такой же результат даст любая ЭВМ.

А теперь "забудем" нажать клавишу "В/О". Наберем значение  $a = 1$  и нажмем клавишу "С/П". Индикатор начинает мигать, мигает значительно дольше, чем обычно: ПМК как будто и не собирается останавливаться. Остановим его принудительно, еще раз нажав клавишу "С/П" — на индикаторе видим 10000000. Что же произошло?

После выполнения программы в счетчике адреса осталось значение 17. При нажатии клавиши "С/П" ПМК начал выполнять команды дальше, т.е. команду по адресу 17, по адресу 18 и т.д. Как мы видели, в этих ячейках есть команды с кодом 00, но это — обычная команда, выполняющая занесение цифры ноль на индикатор. Эти команды и стали выполняться. После занесения семи нулей дальнейшие цифры уже не заносятся, и число 10000000 остается на индикаторе. Если теперь нажмем клавишу "F ПРГ", то увидим на индикаторе "00 00 00 цц", где "цц" — две цифры, показывающие текущее значение счетчика адреса, по ним можно определить, как далеко успел "уйти" ПМК, выполняя команды. Итак, ПМК не замечает и то, что он "вышел" за пределы программы. Программная память также пустой не бывает — там всегда есть какие-то команды. В частности,

могут остаться команды от ранее занесенной программы. И команда с кодом 00, как и число ноль — это не "ничего", а обычная команда.

Кстати отметим, что наиболее близка к понятию "ничего" команда "К НОП" (нет операции) (НОП — совмещенная функция клавиши "0") — она имеет код 54 и не производит никаких изменений в регистрах. С ее использованием встретимся в дальнейшем.

Вернемся в режим вычислений ("F АВТ"), нажмем клавишу "С/П", пустив ПМК выполнять "программу" дальше, и посмотрим, что из этого выйдет. Приблизительно через 15 с он остановится с результатом на индикаторе  $-1 \cdot 10^{14}$ . Посмотрим, где он остановился ("F ПРГ"), и увидим в счетчике адреса странное значение  $-1$ , и последние три команды с кодами 50, 0L и 22, напоминающими начало нашей программы (такой эффект даст МК-54, на других моделях он может отличаться в деталях). Что же произошло? Просто, выполнив все команды до команды с адресом 97 включительно, ПМК пытается выполнить команды с адресами 98, 99, 100 и т.д. Таких адресов нет, но блокировки на все случаи жизни предусмотреть невозможно, и ПМК берет команды из каких-то (трудно понять каких) ячеек памяти — правило вычисления по несуществующему адресу того адреса, откуда будет взята команда, довольно запутано. Вообще говоря, иногда выполняя с ПМК какие-то не предусмотренные инструкцией действия, можно получить вполне осмысленные результаты. На некоторых подобных действиях даже построены программы, приведенные в справочниках по расчетам на ПМК. Однако ни исследовать подобные действия, ни тем более строить на них программы не следует, так как в любой момент с завода могут начать выходить ПМК, внешне ничем не отличающиеся от своих предшественников, но в схему которых внесены изменения, поэтому все, что им положено, они будут выполнять, а за то, что в инструкции не описано, завод ответственности не несет.

И наконец, отметим некоторые особенности выполнения расчетов на ПМК, которые характерны для любых ЭВМ. Выполним нашу программу при  $a = 1$ ,  $b = 10^{20}$ ,  $c = 10^{20}$  (напомним, что значения  $b$  и  $c$  набирают клавишами "1", "ВП", "2", "0") получим результат 1, но такому косинусу соответствует нулевой угол, а наш треугольник имеет угол малый, но все же не нулевой. Дело в том, что косинус нужного угла столь мало отличается от единицы, что это невозможно отразить на восьмиразрядном индикаторе. И хотя кажется, что восемь разрядов — это очень много, но погрешности вычислений могут вызывать

серьезные проблемы, о чем подробнее сказано в гл. 6. Выполнение программы при  $a = b = c = 10^{50}$  даст ошибку ("ЕГГОГ"). Хотя итоговый результат должен получиться равным 0.5, промежуточные значения оказываются слишком велики, и несмотря на то, что и программа правильна, и данные корректны, ПМК отвечает "Не могу". Сообщение об ошибке "ЕГГОГ" получаем после ввода  $b$ , хотя оно должно было получиться уже после ввода  $a$ . Здесь следует отметить еще одну ошибку в конструкции ныне выпускаемых ПМК: при возникновении ошибки "ЕГГОГ" при умножении ПМК не останавливается и, хотя на индикаторе горит "ЕГГОГ", в памяти хранится число с порядком, большим 100, которое можно использовать в дальнейших операциях. Но если при этом получается число с порядком, большим 300, индикатор гаснет, ПМК перестает реагировать на нажатие клавиш, и единственное, что можно с ним сделать, — это выключить, а затем снова включить. Можно надеяться, что эта ошибка будет исправлена в следующих модификациях микрокалькулятора, но пока пользователь ПМК должен об этой особенности знать, чтобы не растеряться, встретившись с ней.

Итак, мы рассмотрели, как реагирует ПМК на ошибки, и увидели, что микрокалькулятор не может заметить практически никаких ошибок, даже попытка выполнить не занесенную в память команду или обработать число, которое не записалось в регистр, с его точки зрения — вполне выполнимое действие. Поэтому и контроль за правильностью программы и за корректностью исходных данных возлагается на программиста. Кроме того, ПМК, как и любая ЭВМ приносит дополнительные заботы о правильности результата вследствие ограниченных точности и диапазона допустимых чисел.

Обратим также внимание на то, как в процессе экспериментов мы переходили из режима программирования в режим вычислений и обратно. Это можно делать в любой момент и сколько угодно раз, при этом нигде ничего не изменяется. На индикаторе мы можем видеть либо содержимое  $R_X$ , либо содержимое части программной памяти. Переключением в режим программирования переводим индикатор на обзор программной памяти, но в  $R_X$ , как и других регистрах, при этом ничего не меняется, и при возврате в режим вычислений мы увидим в  $R_X$  то же самое, что было в нем до перехода в режим программирования.

Любые действия в режиме программирования затрагивают только программную память и не производят никаких изменений в регистрах данных.

Любые действия в режиме вычислений затрагивают только регистры данных и не могут ничего изменить в программной памяти.

Работу со счетчиком адреса можно выполнять в любом режиме клавишами "ШГл" и "ШГп", а в режиме вычислений и клавишей "В/О" (хотя в режиме вычислений содержимое счетчика адреса и не индицируется, но нужные изменения в нем все равно происходят).

## Глава 3. ПРОГРАММИРОВАНИЕ

### 1. УСЛОВИЯ И ВЕТВЛЕНИЯ

Алгоритмы, в которых последовательность действий всегда одинакова и predetermined заранее, встречаются на практике достаточно редко: это самые простые алгоритмы. Например, если мы попытаемся описать в виде алгоритма процесс покупки хлеба, то увидим, что одними командами (фразами в повелительном наклонении "зайти...", "взять...", "уплатить...") обойтись нельзя — нам необходим некоторый анализ обстановки, и действия будут зависеть от результатов этого анализа: если есть ваш любимый батон, то берем его, если нет — берем две булочки, если и их нет, то ... и так далее вплоть до того, что при каких-то условиях мы уходим без покупки.

Таким образом, для описания более сложных алгоритмов необходимы средства для *анализа данных и принятия решений* о дальнейших действиях. Этим средством в А-языке является *условный оператор*, который имеет следующий вид:

*если условие то действия 1 иначе действия 2 все*

Здесь *условие* — это некоторое выражение, значением которого является либо "да" либо "нет", а *действия 1* и *действия 2* — это последовательность любых операторов. Смысл такого оператора полностью соответствует смыслу этой фразы на русском языке: проверяется *условие*, если оно истинно (т.е. имеет значение "да"), то выполняются *действия 1*, а *действие 2* пропускаются, а если ложно (т.е. имеет значение "нет"), то выполняют *действия 2*, а *действия 1* пропускаются. Например:

если батон есть то взять батон; уплатить 15 коп  
иначе взять 2 булки; уплатить 20 коп все

Здесь проверка *батон есть* может дать результат "да" или "нет",

и в зависимости от ответа на этот вопрос мы будем по-разному действовать дальше. Разберемся с функциями слова **все**, которое во фразе на русском языке мы употреблять бы не стали. Оно показывает, где заканчиваются те действия, которые надо выполнить при ложном условии. Например, в записи

**если условие то Д1; Д2 иначе Д3; Д4; Д5;**

не ясно, какие из действий Д3, Д4, Д5 надо выполнить только при ложном условии, а какие действия — и при истинном и при ложном. В то же время в записи

**если условие то Д1 ; Д2 иначе Д3 все ; Д4 ; Д5 ;**

ясно, что к **иначе** относится только действие Д3, и Д4 и Д5 — это самостоятельные действия, которые выполняются независимо от результатов проверки условия. Таким образом, если условие истинно, будут выполнены действия Д1, Д2, Д4, Д5, а если ложно — действия Д3, Д4, Д5.

Условный оператор дает нам средство для ветвления процесса исполнения алгоритма, что наглядно изображается с помощью *схем* (см. рис. 5, *а*): в прямоугольниках на схемах записываются действия, стрелки показывают переход от одного блока к другому, в ромбах записываются условия, и выход из ромба выполняется по одной из двух стрелок в зависимости от результата проверки условия.

Несколько слов о записи. Вообще говоря, условный оператор, как и все другие, можно записывать любым способом, делая переходы со строки на строку в произвольном месте, но

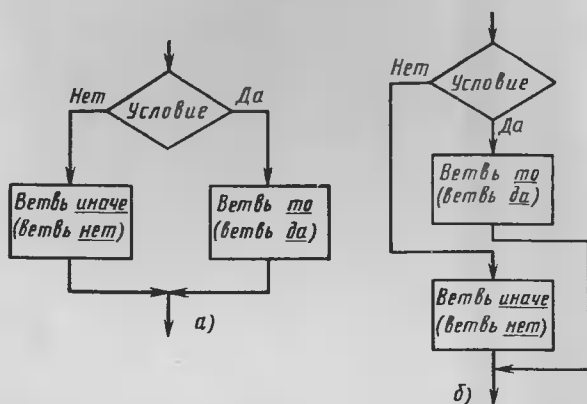


Рис. 5. Схема работы условного оператора:

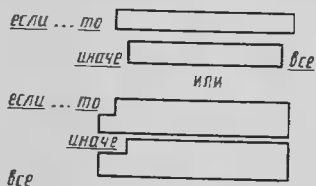
*а* — типовое размещение блоков; *б* — линейное размещение блоков

для удобства чтения программы применяют уже упоминавшуюся запись "лесенкой". Для условного оператора правила записи "лесенкой" таковы:

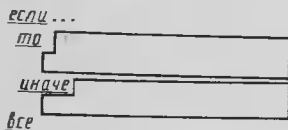
а) если можно весь оператор записать в строку, то это и следует делать:

если . . . то . . . иначе, . . . все

б) в более длинных операторах размещают иначе под то, а все размещают под если или в конце последней строки ветви иначе:



в) в очень длинных операторах начинают с новой строки и часть то, смещая ее на 2—3 позиции вправо:



Применительно к программированию на микрокалькуляторах в условных операторах должны стоять условия, выражающие некоторые отношения между числами (других данных ПМК не обрабатывает), т.е. сравнение чисел, а в качестве действий могут выступать присваивания, ввод, вывод, другой условный оператор (пока мы других действий не знаем). Примером алгоритма числовой обработки с применением условного оператора может служить алгоритм определения минимального из двух чисел, тело которого имеет вид

если  $a < b$  то  $\text{мин} := a$  иначе  $\text{мин} := b$  все

Прежде чем перейти к рассмотрению реализации ветвлений на ПМК, обратим внимание на то, что в условном операторе



порядок выполнения действий не совпадает с порядком их записи: одно из действий пропускается. На схеме обе ветви ветвления располагаются параллельно, однако соответствующие команды в программе для микрокалькулятора мы вынуждены будем расположить последовательно. Но тогда нам нужны средства для того, чтобы пропустить выполнение части команд, а кроме того, средства для анализа данных.

Соответствующими средствами являются команды переходов: "БП", " $F X \geq 0$ ", " $F X < 0$ ", " $F X = 0$ " и " $F X \neq 0$ ". Каждая команда перехода занимает две ячейки программной памяти: в первой находится код команды, а во второй — адрес некоторой ячейки программной памяти. Например:

Адрес	Код	Команда	Комментарий
12	51	БП	В ячейке 12 находится код команды "БП", а в следующей ячейке — адрес 64
13	64	64	

По команде "БП" (безусловный переход) ПМК работает следующим образом: адрес, записанный за кодом команды засылается в счетчик адреса, стирая в нем то значение, которое было. По приведенной выше команде "БП" в счетчик засылается значение 64. Когда ПМК выполнит команду "БП" и приступит к выполнению следующей команды, он будет брать ту команду, адрес которой записан в счетчике адреса. Но при выполнении команды "БП 64" туда попало значение 64, следовательно, сразу после команды с адресом 12 будет выполнена команда с адресом 64. После выполнения этой команды будет выполнена команда с адресом 65, затем с адресом 66 и т.д. Таким образом будет пропущено выполнение части команд. Такое изменение порядка выполнения команд именуется *переходом*: программисты употребляют термины "*переход на адрес 64*", "*переход по адресу 64*", а также "*передача управления на команду (по адресу)...*".

Команду "БП" можно выполнить и в автономном режиме: после нажатия клавиши "БП" и двух цифровых клавиш на индикаторе ничего не меняется, но в счетчик адреса поступит новое значение, в чем можно убедиться, нажав клавишу "FПРГ" (этот прием используется для быстрого выхода к месту ошибочной команды, если клавишами "ШГл" и "ШГп" это делать долго).

По команде "БП" выполняется переход без анализа каких-либо данных, и потому она называется командой *безусловного перехода*. При остальных четырех командах ПМК начинает свою

работу с анализа содержимого РХ, и переход осуществляется только тогда, когда содержимое РХ не удовлетворяет записанному в названии команды условию. Например, команда " $F X \neq 0$ " изменит содержимое счетчика адреса только в том случае, если в РХ находится нуль. Эти команды называют командами *условного перехода*, так как они выполняют переход только при некоторых условиях. Хотя правильнее было бы называть их командами *условного продолжения*, так как при выполнении условия реализуется не переход, а продолжение исполнения команд в естественном порядке.

Для занесения этих команд в программную память нужно нажать клавишу с названием команды (на индикаторе появляется код команды), а затем две цифровые клавиши (на индикаторе после нажатия второй клавиши появится адрес). При необходимости исправить адрес нужно *повторно занести и саму команду*: нажатие двух цифровых клавиш без предварительного нажатия клавиш команд переходов ПМК поймет как команды занесения цифр на индикатор. В программной памяти адрес внешне ничем не отличается от кода операции, ПМК их также не различает. Просто при выполнении команд переходов ПМК берет число из следующей за кодом команды ячейки и считает его адресом. Например, если нажать клавиши "F", " $X \geq 0$ ", "F", " $X \geq 0$ ", "6", "1", то в память ПМК запишутся коды: 59 59 61. Нетрудно проверить, что первая команда " $F X \geq 0$ " исправно выполняет переход на адрес 59, а адрес 61 исправно работает как команда " $\Pi \rightarrow x 1$ ".

Теперь с помощью этих команд будем программировать ветвление. Расположим элементы схемы, приведенной на рис. 5, а, линейно в таком же порядке, в каком следуют соответствующие им элементы оператора на А-языке (рис. 5, б). Очевидно, ромбу, из которого можно выйти либо дальше вниз, либо в обход линейного блока должна соответствовать команда условного перехода, работающая именно таким образом. Программирование линейных блоков, изображенных прямоугольниками, нам знакомо. А ломаной линии, обходящей ветвь иначе, должна соответствовать команда " $\text{Б П}$ ".

Из возможных сравнений ПМК располагает только сравнениями с нулем, поэтому условия следует преобразовать так, чтобы получилось сравнение с нулем. А поскольку на ПМК нет сравнений  $X \leq 0$  и  $X > 0$ , то вместо них приходится использовать сравнения  $-X \geq 0$  и  $-X < 0$  соответственно.

В качестве примера запрограммируем упомянутый выше фрагмент — нахождение минимального из двух значений. Пусть

$a$  размещается в  $P0$ ,  $b$  — в  $P1$ ,  $мин$  — в  $P2$ . Фрагмент программы разместим с адреса 20.

Адрес	Код	Команда	А-язык	Комментарий
20	60	$P \rightarrow x 0$	если	Программируем вычисление условия, преобразованного к имеющемуся сравнению с нулем: $a < b \sim a - b < 0$ ;
21	61	$P \rightarrow x 1$		
22	11	—	$a - b$	
23	5C	$F X < 0$	$< 0$	пишем команду условного перехода, а адрес пока не пишем, так как пока не ясно, куда надо перейти.
24			то	
25	60	$P \rightarrow x 0$	$мин := a$	Программируем ветвь то, она сработает при выполнении условия.
26	42	$x \rightarrow P 2$		
27	51	БП	иначе	Для обхода ветви иначе ставим команду "БП" и опять пропускаем неизвестный пока адрес.
28				
29	61	$P \rightarrow x 1$	$мин := b$	Ветвь иначе программируем так же, как и ветвь то.
30	42	$x \rightarrow P 2$		
31	...	...	все	Заполняем пропущенные адреса: в ячейку № 24 пишем "29", в № 28 — "31"

Параллельная запись программы и текста на А-языке отражает приблизительное соответствие их элементов. Заметим, что служебным словам *если* и *все* команды в программе не соответствуют. Слово *все* отмечает конец программирования условного оператора, в это время уже известно, с каких адресов начинаются фрагменты, соответствующие ветвям, и где они кончаются, и в команду условного перехода можно поставить адрес начала ветви *иначе*, а в команду "БП" — адрес, следующий за концом ветви *иначе*, что мы и делали в нашем примере.

Проследим за работой этого фрагмента программы, выполняя его в пошаговом режиме. Включаем ПМК.

Клавиши	Индикатор	Комментарии
БП 20	0.	Заносим в счетчик адреса 20 и переходим
Ф ПРГ	00 00 00 20	к вводу фрагмента программы в память, начиная не с адреса 00, а с адреса 20
$P \rightarrow x 0$	60 00 00 21	
...	...	Занесение программы окончено
$x \rightarrow P 2$	42 61 31 31	
Ф АВТ	0.	Готовимся к выполнения команд, начиная с адреса 20
БП 20	0.	

1 x → П0	1.	Заносим значения $a=1$ $b=3$ .
3 x → П1	3.	
ПП ПП ПП	-2.	Выполнили три команды, в РХ — значение $a - b$ . в счетчике адреса — 23.
F ПРГ	11 61 60 23	
F АВТ	-2.	
ПП	-2.	Выполнили команду "F x < 0"
F ПРГ	29 5C 11 25	и на индикаторе видим значение счетчика адреса 25: перехода не произошло, так как $X < 0$
F АВТ	-2.	
ПП ПП	1.	Выполнили ветвь то, и сейчас готовы к выполнению команды "БП", адрес которой видим на индикаторе в счетчике адреса.
F ПРГ	42 60 29 27	
F АВТ	1.	
ПП	1.	Выполняем команду "БП" и убедимся,
F ПРГ	42 61 31 31	что выполнен переход по адресу 31.
...	...	

Если повторить этот эксперимент при  $a = 5$ , а  $b$  оставить прежним, то после выполнения команды "F X < 0" в счетчике адреса увидим 29: при положительном значении X переход выполняется.

В записи программы рекомендуется оставлять сплошные стрелки, показывающие направление переходов. Штриховые стрелки нужны были лишь для иллюстрации процесса составления программы.

Сформулируем правила перевода ветвления с А-языка в программу для ПМК.

1. Программируется вычисление выражения из условия, приведенного к имеющемуся на ПМК сравнению с нулем.
2. Ставится нужная команда условного перехода и пропускается одна ячейка под адрес.
3. Программируется ветвь то.
4. Ставится команда "БП" и пропускается ячейка под адрес.
5. Программируется ветвь иначе.
6. Пропущенная на шаге 2 ячейка заполняется адресом начала ветви иначе, а пропущенная на шаге 4 — адресом очередной еще не занятой ячейки.

Так же, как ранее мы позволили употребление конструкций ввод и вывод внутри операторов присваивания, можно позволить использование внутри выражения и конструкции если — то — иначе — все, но в этом случае на ветвях то и иначе должны стоять не операторы, а выражения. Например, можно написать

$мин := \text{если } a < b \text{ то } a \text{ иначе } b \text{ все}$

и понимать это следует так: при  $a < b$  работает то, что стоит перед если, т.е. " $мин :=$ ", и то, что стоит между то и иначе, т.е. " $a$ " — вместе получаем " $мин := a$ "; аналогично, при  $a \geq b$  получаем " $мин := b$ ". Иными словами, присваивание как бы вынесено за скобки. Программируется такая конструкция (она называется *условное выражение*) так же, как и условный оператор, только вычисленное на ветви то или иначе значение выражения просто остается на РХ. Программа получается на одну команду короче:

Адрес	Команды	Комментарий
20	$\Pi \rightarrow x \ 0 \quad \Pi \rightarrow x \ 1 \quad -$	
23	$F \ X < 0 \quad 28$	
25	$\Pi \rightarrow x \ 0$	Вызываем в РХ значение $a$ , и переходим к команде " $x \rightarrow \Pi \ 2$ ". А здесь вызываем значение $b$ . Запись содержимого РХ в Р2.
26	$\text{Б} \ \Pi \ 29$	
28	$\Pi \rightarrow x \ 1$	
29	$x \rightarrow \Pi \ 2$	

К команде " $x \rightarrow \Pi \ 2$ " приходим либо с адреса 26 со значением  $a$ , либо с адреса 28 со значением  $b$  в РХ — это значение и записывается в Р2.

Понятно, что с условными выражениями можно строить и более сложные операторы, например:

$p := (\text{если } a < b \text{ то } e^2 - x \text{ иначе } y \text{ все}) \cdot (c - b);$

программирование которых выполняется по общим правилам. При этом условное выражение должно быть первым операндом в выражении.

**Замечание.** Запрет использовать условное выражение в качестве первого операнда связан с некорректной реализацией команд условного перехода: по правилам ПОЛИЗа они должны выбрасывать из стека содержимое X, сдвигая весь стек в сторону РХ. Однако они это не делают, поэтому при программировании по общим правилам оператора

$m := (\text{если } a \geq b \text{ то } a \text{ иначе } b \text{ все}) * c;$

получим правильный результат, а для оператора

$m := c \cdot (\text{если } a \geq b \text{ то } a \text{ иначе } b \text{ все})$

к моменту выполнения операции умножения в стеке будет:  $c$  — в РZ,  $a - b$  — в РY, значение  $a$  или  $b$  — в РХ. Понятно, что невыброшенное

значение  $a - b$  будет мешать. Строго говоря, можно запрограммировать и такой оператор, но для условного выражения, стоящего не первым операндом, программирование ветвей то и иначе должно начинаться командой "FO", выбрасывающей значение условия из стека (как видим, опять эту команду приходится применять только для компенсации ошибок в конструкции ПМК).

В ряде случаев одна из ветвей ветвления оказывается пустой, т.е. при выполнении или не выполнении условия надо просто ничего не делать. Если пустой оказывается ветвь иначе, то в записи на А-языке можно не писать и само слово иначе. Такое неполное ветвление называется *обход*:

если условие то действия все

При невыполнении условия действия обходятся.

Использование обхода позволяет, например, запрограммировать вычисление минимального значения на две команды короче:

$мин := a$ ; если  $a > b$  то  $мин := b$  все

Адрес	Команды	А-язык	Комментарий
20	$П \rightarrow x 0 \quad x \rightarrow П 2$	$мин := a$	Сначала считаем, что минимум $-a$
22	$П \rightarrow x 0 \quad П \rightarrow x 1$	если	Если $a \geq b$ , то оставляем в <i>мин</i> значение $a$
24	-	$a - b$	
25	$F X \geq 0 \quad 29$	$\geq 0$ то	и более ничего не делаем,
27	$П \rightarrow x 1 \quad x \rightarrow П 2$	$мин := b$	в противном случае пересылаем в <i>мин</i> значение $b$ , стирая значение $a$
29	...	все	

Если же использовать возможность присваивания в выражении, то алгоритм

если  $(мин := a) \geq b$  то  $мин := b$  все

даст еще более короткую программу: становится ненужной команда по адресу 22, повторно вызывающая в  $PX$  значение, которое там и так уже есть.

Еще один пример использования обхода дает такая задача: если  $a > b$ , то поменять их значения местами, т.е. сделать так, чтобы в  $a$  находилось минимальное, а в  $b$  — максимальное из двух значений. Тело алгоритма имеет вид

если  $a > b$  то  $c := a$ ;  $a := b$ ;  $b := c$  все

а программу для ПМК мы предоставим написать читателям.

Задачи

1. Бидон представляет собой цилиндр диаметром  $D$  и высотой  $H$  с горловиной в форме усеченного конуса с диаметром горла  $d$  и высотой

горловины  $h$ . В бидон до высоты  $H_0$  налита вода. Определить объем воды (т.е. написать алгоритм и программу, которые по значениям  $D$ ,  $d$ ,  $H$ ,  $h$  и  $H_0$  выдадут значение объема воды, определяемое по формулам:

$$V = \begin{cases} \frac{\pi D^2}{4} H_0 & \text{при } H_0 \leq H \\ \frac{\pi D^2}{4} H + \frac{\pi D^2}{12} \cdot \frac{Dh}{D-d} \left(1 - \left[1 - \frac{(H_0 - H)(D-d)}{Dh}\right]^3\right) & \text{при } H_0 > H. \end{cases}$$

2. Автобус отправился в путь в  $ЧЧ$  часов и  $ММ$  минут и будет в пути  $Ч$  часов и  $М$  минут (время в пути не более суток). Определить время его прибытия: сутки (0 – те же сутки, 1 – на следующие сутки), часы и минуты.

## 2. ЦИКЛЫ

Наиболее подходящей для переложения на плечи ЭВМ работой является многократное выполнение одинаковых действий. Необходимость такого выполнения придает смысл программированию: один раз написанная последовательность команд может быть выполнена многократно, как мы это делали в п. 3 гл. 1. Но даже в пределах одного расчета часто бывает необходимо повторить несколько раз одинаковую последовательность команд. Простейшее решение этой проблемы — выписать эту последовательность столько раз, сколько нужно, но это, во-первых, неэкономично, а во-вторых, не всегда осуществимо, так как число повторений нередко зависит от исходных данных. Поэтому во всех языках программирования имеются специальные конструкции, описывающие многократное повторение некоторых действий. Эти конструкции называются *операторами цикла*, а сам процесс повторения действий называется *циклом*.

Циклы имеют множество различных форм, самая простая из которых соответствует фразе на русском языке, начинающейся словами "повторять столько-то раз...". Оператор такого цикла имеет вид

**повторять количество раз действия *кпвт***

и смысл его соответствует смыслу фразы на русском языке. Служебное слово *кпвт* (конец повторения) служит для указания конца повторяемых действий (как все в операторе *если*).

4. *Количество* может быть любым выражением, если его значение оказывается дробным, то в качестве числа повторений берется

целая часть. Сама фраза "повторять ... раз" называется *заголовком цикла*, а повторяемые действия — *телом цикла*. Работа оператора цикла заключается в том, что тело цикла повторяется заданное количество раз.

В языках программирования в аналогичных конструкциях вместо слова **повторять** обычно употребляют слово **цикл** или **нц** (начало цикла), а вместо **кпвт** — **кц** (конец цикла) (например, **нц** и **кц** можно встретить в школьном учебнике информатики), но мы выберем вариант с русской лексикой, наиболее близкий подходящий к обычной фразе на русском языке.

В качестве примера использования цикла рассмотрим задачу вычисления  $M^k$  при целых  $M$  и  $k$ . Применение для этой цели операции " $F x^y$ " дает результат с довольно значительной погрешностью (например, при вычислении  $3^6$  получаем 728.99942 вместо 729 — погрешность заходит даже в предпоследний разряд). А для получения точного результата придется вычислять  $M^k$ , используя определение понятия степени:  $k$ -кратным умножением. Тело алгоритма можно записать:

степ: = 1;

повторять  $k$  раз степ: = степ  $\cdot M$  **кпвт**

Проследим работу этого алгоритма при  $M = 3$ ,  $k = 6$ . Изначально переменная *степ* имеет значение, равное единице; при первом выполнении тела цикла она принимает значение  $M$ , при втором —  $M^2$  и т.д. После того как тело цикла выполнится  $k$  раз, *степ* получает значение  $M^k$ , что и требовалось.

Для реализации таких циклов на ПМК имеются специальные команды — команды циклов с общим названием " $F Li$ ", где  $i$  может быть равным 0, 1, 2, 3, т.е. команды " $F L0$ ", " $F L1$ ", " $F L2$ ", " $F L3$ ". Эти команды также занимают по две ячейки программной памяти, в первой из которых хранится код команды, а во второй — адрес (при исправлении адреса необходимо повторно занести и команды, исправить только адрес нельзя). Цифры 0, 1, 2, 3 указывают номер регистра, с которым работает команда. А работают эти команды так: содержимое упомянутого в команде регистра уменьшается на единицу, если в результате получается ноль, далее выполняется следующая команда, а если не ноль, происходит переход по заданному в команде адресу. Если в регистре, с которым работает команда, было не целое число, то после первого же применения команды " $F Li$ " в нем остается уменьшенная на единицу целая часть числа.

Понятно, что эти команды являются удобным средством для обеспечения действий цикла **повторять ... раз**. Правила программирования таких циклов следующие.



1. Если программа содержит циклы **повторять ... раз**, то распределение регистров начинается с отведения одного из регистров  $P0...P3$  под *счетчик цикла* (иногда нужно отвести регистры и под несколько счетчиков), затем распределяется память для остальных переменных.

2. Заголовок **"повторять...раз"** программируется командами вычисления стоящего в заголовке выражения с засылкой результата в регистр-счетчик.

3. Программируется тело цикла.

4. Служебное слово **кпвт** программируется командой **"F Li"**, где  $i$  — номер регистра-счетчика этого цикла, с адресом начала тела цикла.

Проиллюстрируем правила программирования на примере цикла вычисления  $M^k$ .

Адрес	Команда	А-язык	Комментарий
Счетчик — $P0$ $M$ — $P1$ $k$ — $P2$ степ — $P3$		числ $M$ , $k$ , степ;	$P0$ отводим для счетчика, остальные регистры распределяем по описаниям переменных, как обычно
...	...	...	Программируем только нужный фрагмент
10	1	степ: = 1;	
11	$x \rightarrow P3$		
12	$P1 \rightarrow x 2$	повторять	Вычисление количества повторений
13	$x \rightarrow P0$	$k$ раз	сводится к пересылке значения $k$ в счетчик.
14	$P1 \rightarrow x 3$	степ ·	С адреса 14 начинается тело цикла, состоящее из одного оператора присваивания
15	$P1 \rightarrow x 1$	$M$	
16	$x$		
17	$x \rightarrow P3$	$\rightarrow$ степ;	
18	С/П		Эту команду добавим для изучения работы команд цикла.
19	F L0	кпвт	Слово кпвт программируется командой "F L0" с адресом начала тела цикла.
	└14		

Посмотрим, как будет выполняться наша программа. Зане-  
сем в  $P1$  и  $P2$  соответственно значения  $M = 4, k = 3$ , в счетчик  
адреса — значение 10.

Клавиши	Индикатор	Комментарии
С/П	4.	Выполнили первый проход тела цикла,
F ПРГ	50 43 12 19	стоим перед командой "F L0".
F АВТ	4.	
П → x 0	3.	В счетчике — число проходов цикла, включая текущий.
ПП	3.	Выполняем команду цикла,
F ПРГ	40 62 43 14	и видим, что произошел переход на адрес 14.
F АВТ	3.	
П → x 0	00000002.	В счетчике — число оставшихся проходов цикла (в несколько необычном виде).
С/П	16.	Выполняем следующий проход цикла,
ПП	16.	выполняем команду цикла,
F ПРГ	40 62 43 14	убеждаемся, что опять произошел переход на адрес 14.
F АВТ	16.	
С/П	64.	Третий проход цикла,
П → x 0	00000001.	перед выполнением "F L0" в счетчике — 1.
ПП	1	Выполняем команду цикла,
F ПРГ	14 5Г 50 21	Счетчик адреса показывает, что произошел выход из цикла: команда цикла перехода не выполнила.
F АВТ	1.	
П → x 0	00000001.	В счетчике по-прежнему единица.

Обратите внимание на то, что при выходе из цикла уменьшенное на единицу значение счетчика, т.е. ноль, в счетчик не записывается, и в нем остается единица — это конструктивный недостаток ПМК. В дальнейшем мы увидим, что это обстоятельство создает некоторые сложности — по счетчику не ясно, произошел ли выход из цикла или еще выполняется последний проход цикла.

Результат нашей программы оказывается в R3, он же виден и на индикаторе (если не проводить дополнительных действий, чтобы посмотреть содержимое счетчика). Заметим, однако, что эти действия не мешают получению результата, так как мы вызывали какие-то значения в RХ тогда, когда содержимое RХ и всего стека было не нужно.

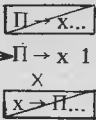

Точно такие же результаты получатся, если взять значение  $k = 3.95$ . После первого же выполнения команды "F L0" и P0 останется то же значение 00000002., т.е. выполнение команды цикла начинается с отбрасывания дробной части числа.

При отрицательных значениях  $k$  цикл повторять  $k$  раз не должен работать вообще, т.е. тело цикла не должно выполняться ни разу. Но команды цикла на ПМК при отрицательном значении счетчика изменяют его значение довольно сложно объяснимым способом (по крайней мере, дают не то, что интуитивно можно ожидать), поэтому цикл с отрицательным числом повторений будем считать ошибкой, точнее смысловой ошибкой 2-го рода: цикл выполняется очень большое число раз.

На примере программы изучим еще один очень полезный прием организации циклических программ. Обратим внимание, что перед выполнением команды цикла на РХ находится значение переменной *step*, однако при выполнении нового прохода цикла это значение опять вызывается в РХ, хотя оно там уже есть. Как избавиться от лишних действий? Для этого вспомним, что *регистры стека могут хранить данные так же, как и адресуемые регистры*. И под одну, иногда под две переменные можно отвести не адресуемый регистр, а РХ, иногда и РУ. Поскольку стек работает при всех операциях, то большее количество переменных в стеке разместить, как правило, не удастся. Для переменной, хранящейся в РХ, не нужны команды " $\Pi \rightarrow x$ " и " $x \rightarrow \Pi$ " для вызова или записи ее значения. Однако иногда приходится выполнять некоторые нестандартные действия для того, чтобы вернуть в РХ ее значение, которое ушло вглубь стека в результате работы с другими переменными. Естественно, что в стеке имеет смысл хранить только активно используемые переменные. Помимо выигрыша в длине и скорости программы на этом достигается экономия адресуемых регистров, которых на ПМК не так много, как хотелось бы.

В программе вычисления  $M^k$  такой активно используемой переменной является *step*. Отведем для ее хранения РХ, прочие переменные оставим в тех же регистрах. Получаем программу:

Адрес	Команда	А-язык	Комментарий
10	1 <div style="border: 1px solid black; padding: 2px; display: inline-block;"><math>x \rightarrow \Pi \dots</math></div>	$step := 1$ ;	Команда " $x \rightarrow \Pi$ " становится ненужной
11	$\Pi \rightarrow x$ 2	повтор	(Мы часто будем сокращать "повторить")
12	$x \rightarrow \Pi$ 0	$k$ раз	А здесь придется добавить команду,
13	$\leftrightarrow$		чтобы вернуть в РХ значение <i>step</i>

Адрес	Команда	А-язык	Комментарий
			Команда вызова становится лишней.
14	$\bar{П} \rightarrow x \ 1$	ступ: =	В теле цикла программируется только умножение на $M$ , команда записи также не нужна
15	$x$	ступ: $M$ ;	
			
16	$FLO$	кпвт	Работа этой команды не затрагивает содержимого $PX$
17	14		

Фрагмент сократился примерно на треть, и работает почти в 2 раза быстрее, так как в цикле повторяется меньше команд.

Стековую команду " $\leftrightarrow$ " нам пришлось применить опять потому, что команды " $x \rightarrow П$ " работают со стеком не так, как положено по теории: здесь видно, что сдвиг стека в сторону  $PX$  при засылке значения в адресуемый регистр был бы для нас удобнее. Но поскольку работать приходится с тем ПМК, который есть, более эффективно решить задачу возврата значения *ступ* в  $PX$  по-другому: поменять местами команды с адресами 11 и 12 и команду с адресом 10. Этот прием в дальнейшем будет использоваться довольно часто: начальные присваивания переменной, хранимой в  $PX$ , выполняются после занесения числа повторений в счетчик цикла, т.е. несколько нарушая порядок записи алгоритма на А-языке.

В завершение отметим, что в  $PX$  можно хранить только такую переменную  $p$ , которая в цикле участвует только в операторах вида

$$p := p \circ \dots \quad \text{или} \quad p := f(p) \circ \dots$$

где  $\circ$  — это некоторая двуместная операция, а  $f$  — некоторая элементарная функция, т.е. одноместная операция с точки зрения ПМК.

На схемах цикл повторять  $M$  раз показывается так, как это сделано на рис. 6: шестиугольник соответствует заголовку цикла, а маленький ромб — слову **кпвт**. Правила записи "лесенкой" для циклов таковы:

а) если весь оператор помещается в одну строку, то его следует размещать в строке

повторять ... раз ... **кпвт** ;

б) в более длинных циклах тело цикла записывается со смещением на две — три позиции вправо, а слово **кпвт** пишется в

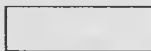
отдельной строке под повторять или в конце последней строки тела цикла:

*повторять ... раз*



*к л в т*

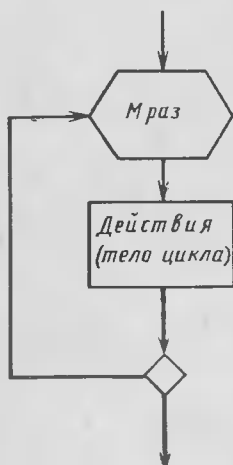
*повторять ... раз*



*к л в т*

Смещение позволяет увидеть, что входит в тело цикла.

Программы, не содержащие циклов, встречаются редко. И изучив понятие цикла, можно перейти к рассмотрению первой прикладной программы, применяемой для решения действительно часто встречающейся в практике задачи, относящейся как раз к тому классу задач, решение которых быстрее получить на ПМК, чем дойти до комнаты, где стоит ЭВМ. Эта задача, математическая формулировка которой звучит так: вычислить определенный интеграл от функции  $f(x)$  на отрезке  $[a, b]$ :  $\int_a^b f(x) dx$ .

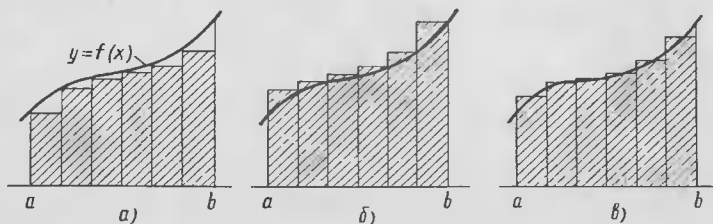


Геометрически интеграл представляет собой площадь фигуры, ограниченной осью  $x$ , графиком функции  $f(x)$  и прямыми  $y = a$  и  $y = b$  (рис. 7). К такой математической формулировке сводится множество разнообразных задач: если  $f(x)$  — это зависимость скорости автомобиля от времени  $x$ , то интеграл —

Рис. 6. Схема цикла

Рис. 7. К задаче вычисления определенного интеграла:

$a$  — формула левых прямоугольников;  $b$  — формула правых прямоугольников;  $в$  — формула средних прямоугольников



это пройденный в течение времени с момента  $a$  до момента  $b$  путь; если  $f(x)$  описывает изменение электрического или водяного тока, то интеграл даст перенесенное количество электричества (заряд) или общий расход воды; и т.д. Получить значение интеграла в *аналитическом* виде, т.е. в виде формулы, включающей значения  $a$  и  $b$ , удастся только для простейших функций. Чаще значение интеграла приходится вычислять *численно*, т.е. по конкретным значениям  $a$  и  $b$  для конкретной функции вычислять *приближенное числовое значение*.

Простейший способ получения приближенного числового значения заключается в том, что отрезок  $[a, b]$  разбивается на  $N$  частей, и в качестве результата берется сумма площадей прямоугольников, заштрихованных на рис. 7, а. Одна из сторон каждого прямоугольника равна шагу разбиения  $h$ , где  $h = (b - a)/N$ , а вторая — значению  $f(x_i)$ , где  $x_i$  — некоторая точка соответствующей части отрезка  $[a, b]$ . Если все  $x_i$  взять в левых концах отрезков, на которые делится отрезок  $[a, b]$ , то получим формулу *левых прямоугольников* (как на рис. 7, а), если в правых концах — формулу *правых прямоугольников*, а если посередине (рис. 7, б) — формулу *средних прямоугольников* (рис. 7, в). Все три варианта равноценны, но для монотонных функций формула средних прямоугольников дает меньшую погрешность.

Рассмотрим алгоритм вычисления интеграла по формуле левых прямоугольников:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} (f(a + ih) \cdot h), \text{ где } h = (b - a)/N.$$

Понятно, что при заданной конкретной функции исходными данными для алгоритма будут значения  $a$ ,  $b$  и  $N$  (значение  $N$  должно быть достаточно большим, чтобы получить результат, близкий к истинному; оценку получающейся погрешности можно найти в любом курсе математического анализа или численных методов), а результатом — значение накопленной суммы, для которой возьмем переменную с именем *инт*. Здесь мы фактически уже приступили к разработке алгоритма методом пошаговой детализации. На основе сказанного выше можем записать:

```
алг интеграл ;
    числ  $a, b, N, \text{инт}, \dots$ 
    ввод  $a, b, N$ ;
    вычисление значения интеграла;
    вывод инт ;
```

**кон**

Вычисление значения интеграла заключается в  $N$ -кратном повторении одинаковых действий — в данном случае вычислении площади очередного прямоугольника — и в добавлении полученного значения площади к накопленной сумме, что можно записать так:

**повторять  $N$  раз**

    вычислить площадь очередного прямоугольника;

    добавить результат к *инт*

**кпвт**

Площадь очередного прямоугольника равна  $f(a + ih)h$ , для его вычисления нужно ввести еще одну переменную  $i$ , которую на каждом шаге надо увеличивать на единицу, или же переменную  $x$ , которую на каждом шаге надо увеличивать на  $h$ . Выберем второй вариант. Значение  $h$  тоже надо вычислить. Все значения площадей прямоугольников можно присваивать одной и той же переменной; после того как они добавляются к значению *инт*, их можно стереть. Получаем

**повторять  $N$  раз**

$\text{плоч} := f(x) \cdot h; x := x + h;$

$\text{инт} := \text{инт} + \text{плоч}$

**кпвт**

и в строку описания переменных вместо многоточия надо добавить имена *плоч* и  $h$ . Но где-то переменные *инт* и  $x$  должны получить начальные значения, к которым будут добавляться значения *плоч* и  $h$  на первом шаге цикла. Эти значения они должны получить перед началом выполнения цикла:

$x := a; \text{инт} := 0;$

Почему начальные значения именно такие — вполне очевидно.

Итак, получаем алгоритм:

**алг интеграл ;**

**числ**  $a, b, N, h, x, \text{плоч}, \text{инт} ;$

**ввод**  $a, b, N;$

$h := (b - a)/N; x := a; \text{инт} := 0 ;$

**повторять  $N$  раз**

$\text{плоч} := f(x) \cdot h; x := x + h;$

$\text{инт} := \text{инт} + \text{плоч}$

**кпвт;**

**вывод** *инт* ;

**кон**

Обратите внимание на то, что даже простейший алгоритм редко удается писать сразу в окончательном виде. Строку описания переменных мы дописывали по мере того, как выяснялось, какие вспомогательные переменные нам нужны; сначала написали тело цикла, потом выяснилось, что перед циклом необходимо сделать несколько присваиваний. Это типично для программирования. Кстати заметим, что обычно каждый цикл сопровождается несколькими *начальными присваиваниями* перед ним или *конечными присваиваниями* за ним (в нашем случае их нет), иногда — и теми и другими присваиваниями.

Но работа с алгоритмом не закончена. Пока получен простейший алгоритм — лобовое решение задачи, которое хорошей программы не даст. Приступаем к улучшению алгоритма. Во-первых, заметим, что значение *площ* используется только один раз, следовательно, это значение нет смысла запоминать в переменной: его можно сразу использовать для добавления к *инт*:

$$\text{инт} := \text{инт} + f(x) \cdot h;$$

И переменная *площ* становится ненужной. Значение *a* нужно дважды, но если сначала его присвоить переменной *x*, то затем можно вместо значения *a* использовать значение *x*. Итак, переменная *a* тоже не нужна, как и *b*, значение которой используется однажды. Наконец, обратим внимание на то, что умножение на *h* можно вынести за скобки в формуле

$$\sum_{i=0}^{N-1} (f(a + i \cdot h) \cdot h) = \left[ \sum_{i=0}^{N-1} f(a + i \cdot h) \right] \cdot h$$

и вынести за цикл в программе: незачем многократно повторять одно и то же. В результате получаем гораздо лучший вариант:

алг интеграл ;

числ *N*, *h*, *x*, инт ;

ввод {*a*} *x*;

инт := 0 ; *h* := (ввод {*b*} - *x*) / ввод *N*;

повторять *N* раз

инт := инт + *f*(*x*); *x* := *x* + *h*;

кпвт;

вывод инт · *h* ;

кон



Данный пример иллюстрирует типичный процесс построения алгоритма. Хотя опытный программист может написать второй вариант сразу, но все же первый вариант он "прокручивает" в уме. Приступаем к программированию. По формальным правилам перевода программа получается однозначно. Но имеется еще один неформальный момент: *распределение памяти*. Во-первых, у нас имеется две активно используемые в цикле переменные, которые можно хранить в РХ: *инт* и *х*. Поэтому напомним и сравним три варианта программы: без хранения переменных в РХ, с хранением в РХ *инт* и с хранением *х*. Во-вторых, покажем на этой программе еще один прием эффективного программирования: *совмещение переменных в памяти*. Заметим, что значение *N* нужно нам только до заголовка цикла, а после заголовка нужен счетчик — это позволяет отвести под *N* и счетчик *один и тот же регистр*. А так как начальное значение счетчика равно *N*, это сэкономит нам еще и две команды.

В качестве функции возьмем  $f(x) = x^2/2$ . Интеграл от нее легко вычисляется аналитически — это позволит нам оценить правильность программы и точность вычислений.

Адрес	Команды	А-язык	Комментарий
		алг интеграл;	Первый вариант: без хранения переменных в РХ.
	Счетчик ← Р0		Совмещаем в регистрах <i>N</i> и счетчик.
	<i>N</i> ← Р0	числ <i>N</i> ,	Дальше регистры распределяем как обычно: подряд
	<i>h</i> ← Р1	<i>h</i> ,	
	<i>x</i> ← Р2	<i>x</i>	
	инт ← Р3	инт;	
00	<i>x</i> → П 2	ввод { <i>a</i> } <i>x</i> ;	
01	0 <i>x</i> → П 3	инт := 0 ;	
03	С/П      П → <i>x</i> 2 —	(ввод { <i>b</i> } — <i>x</i> )	
06	С/П <i>x</i> → П 0 ÷	/ввод <i>N</i>	
09	<i>x</i> → П      1	→ <i>h</i>	
	<div style="border: 1px solid black; padding: 2px; display: inline-block;">П → <i>x</i>      <i>x</i> → П 0</div>	повторять <i>N</i> раз	Эти команды не нужны вследствие совмещения счетчика и <i>N</i> в Р0
10	П → <i>x</i> 3	инт +	
11	П → <i>x</i> 2	$F \ x^2$	$x^2/2$
13	2 ÷	+	
16	<i>x</i> → П 3	→ инт ;	Тело цикла программируется как обычно

Адрес	Команды	А-язык	Комментарий
17	$\Pi \rightarrow x \ 2$	$\Pi \rightarrow x \ 1$	$x + h$
19	$+$	$x \rightarrow \Pi \ 2$	$\rightarrow x;$
21	F L0	10	кпвт
23	$\Pi \rightarrow x \ 3$	$\Pi \rightarrow x \ 1$	вывод
25	$x$	C/Π	инт·h ;
кон			

Теперь посмотрим, что получается, если хранить в РХ инт.

Адрес	Команды	А-язык	Комментарий
...	...	алг интеграл ; числ ...	Второй вариант. Все регистры, кроме РЗ, распределены так же, как и ранее
00	$x \rightarrow \Pi \ 2$	ввод $\{a\} x ;$	
01	C/Π $\Pi \rightarrow x \ 2 -$	(ввод $\{b\} - x)$	
04	C/Π $x \rightarrow \Pi \ 0 \div$	/ввод $N$	
07	$x \rightarrow \Pi \ 1$	$\rightarrow h$	Начальное присваивание переменной, хранящейся в РХ, приходится перенести за заголовков цикла
повторять N раз			
08	0	инт: = 0 ;	Одну команду уже сэкономили
09	$\Pi \rightarrow x \ 2$	F $x^2$	
11	2 $\div +$	инт: = инт+ $x^2/2 ;$	Сэкономили еще две команды в теле цикла.
14	$\Pi \rightarrow x \ 2$	$\Pi \rightarrow x \ 1$	
16	$+$	$x \rightarrow \Pi \ 2$	
18	$\leftrightarrow$		Но сейчас значение инт ушло в РУ, приходится его "доставать" из РУ
19	F L0 09	кпвт;	
21	$\Pi \rightarrow x \ 1 \times$	C/Π	вывод инт·h;
кон			

Итак, мы сэкономили три команды из 26 — это немало. Но тело цикла уменьшилось только на одну команду, так что выигрыш в скорости будет небольшой.

Посмотрим третий вариант — хранение в  $PX$  значения  $x$ . Распределение регистров прежнее, но  $P2$  просто не используется.

Адрес	Команды	А-язык	Комментарий
		алг интеграл числ...	Третий вариант.
		ввод $\{a\} x$ ;	Набранное значение просто остается в $PX$ , следующая команда проталкивает его в $PY$ .
00	$B \uparrow$	$0 \rightarrow P3$ инт: = 0 ;	
03	$\leftrightarrow$		Возвращаем значение $x$ в $PX$ .
04	$B \uparrow$		Снимаем копию с $x$ , так как $x$ участвует в операции.
05	$C/P$	— (ввод $\{b\}$	Операнды не там, где надо, — пришлось поменять знак у результата.
07	$/- /$	— $x$ )	
08	$C/P$	$x \rightarrow P0 \div$ /ввод $N$	
11	$x \rightarrow P1$	$\leftrightarrow \rightarrow h$	И опять возвращаем $x$ в $PX$ .
		повторять $N$ раз	
13	$B \uparrow$		Сняли копию с $x$ для ис- пользования в операции.
14	$F x^2$	$2 \div x^2/2 +$	
17	$P \rightarrow x$	$3 +$ инт	
19	$x \rightarrow P3$	$\leftrightarrow \rightarrow$ инт ;	И опять возвращаем $x$ в $PX$ .
21	$P \rightarrow x$	$1 + x := x + h$	Только здесь экономим две команды.
23	$F L0$	$13$ кпвт ;	
25	$P \rightarrow x$	$3 P \rightarrow x$ 1	вывод
27	$x$	$C/P$ инт $\cdot h$ ;	
		кон	

Как видим, и программа получилась длиннее, и программировать было очень неудобно: все время приходилось следить за тем, куда (в какой регистр стека) ушел  $x$ , снимать копии  $x$ . Дело в том, что хранение переменной в  $PX$  выгодно тогда, когда переменная участвует *только* в операциях вида

переменная: = переменная ○ ...

и в теле цикла кроме оператора приведенного выше вида имеется *не более одного оператора*: это связано с тем, что каждый оператор "проталкивает" содержимое  $PX$  вглубь стека, откуда его затем надо доставать.

Выполнение этих программ при  $a = 1$ ,  $b = 2$ ,  $N = 10$  даст результат 1.0925, причем первый вариант программы будет работать 42 с, а второй — 36 с\* (третий вариант рассматривать не будем, так как он явно хуже). Точный результат — 1.1666666. Понятно, что значение получилось с недостатком (см. рис. 7), так как функция монотонно возрастает. Если взять  $N = 20$ , то получим более точный результат: 1.129375, но первый вариант программы будет работать 1 мин 25 с, а второй — 1 мин 12 с (время считают от нажатия клавиши "С/П" после набора значения  $N$  и до высвечивания результата).

Если в теле цикла поменять местами операторы: сначала вычислять новое значение  $x$ , а затем наращивать *инт*, то получим расчет по формуле *правых прямоугольников*, который по такой функции дает результат *с избытком*.

В процессе работы цикла в счетчике цикла находится некоторое целое число, показывающее число оставшихся проходов цикла. Хотя оно и записано в несколько необычном виде, но его вполне можно использовать в любых операциях. И потребность использования значения счетчика цикла в вычислениях возникает достаточно часто. Чтобы это сделать, нужно рассматривать счетчик как обычную переменную и дать ей некоторое имя, а в заголовке цикла указать, какая переменная будет использоваться в качестве счетчика. Заголовок цикла принимает вид

**меня переменная повторять количество раз**

При программировании таких циклов вместо отведения одного из регистров  $P0 \dots P3$  под счетчик, отводим его под упомянутую переменную.

Рассмотрим алгоритм вычисления  $M! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot M$  (запись " $M!$ " читается " $M$  факториал"). Первый и простейший вариант тела алгоритма имеет вид:

факт: = 1 ;  $k$ : = 1 ;

повторять  $M$  раз факт: = факт  $\cdot k$  ;  $k$ : =  $k + 1$  кпвт

---

\* На разных экземплярах микрокалькуляторов время работы может быть различным.

Однако понятно, что перемножать числа можно и в обратном порядке:  $M! = M \cdot (M - 1) \cdot (M - 2) \cdot \dots \cdot 2 \cdot 1$ . Тогда алгоритм принимает вид:

факт: = 1 ;  $k := M$ ;

повторять  $M$  раз факт: = факт· $k$  ;  $k := k - 1$  кпвт

В этом варианте нетрудно заметить, что значение  $k$  равно как раз значению счетчика цикла, а следовательно, вместо  $k$  можно использовать значение счетчика. Записывается это так:

факт: = 1 ;

меняя  $k$  повторять  $M$  раз факт: = факт· $k$  кпвт

Запрограммируем этот алгоритм. Сразу отметим, что переменная *факт* удовлетворяет всем требованиям, которые делают выгодным ее хранение в РХ.

Адрес	Команды	А-язык	Комментарий
	$k - P0$ $M - P1$ факт - РХ	алг факториал ; числ $k$ , $M$ , факт ;	Для $k$ берем $P0$ , а для счетчика теперь регистр не нужен.
00	$x \rightarrow P1$	ввод $M$ ;	
01	$P \rightarrow x1$ $x \rightarrow P0$	меняя $k$ , повторять $M$ раз	Заголовок цикла программируем как обычно, только теперь счетчик имеет свое имя.
03	1	факт: = 1 ;	Хранимая в РХ переменная получает начальное значение после заголовка цикла.
04	$\rightarrow P \rightarrow x0$ $x$	факт: = факт· $k$	В теле цикла значение $k$ используется как обычная переменная.
06	$F L0$ 04	кпвт	
08	С/П	вывод факт; кон	Нужное значение уже в РХ.

Посмотреть работу этой небольшой программы удобно в пошаговом режиме, нажимая клавишу "ПП".

Отметим, что допустима и запись вида

меняя  $M$  повторять  $M$  раз

В этом случае в качестве счетчика используется сама переменная

$M$ . Понятно, что после цикла хранившееся в ней значение теряется, и после окончания работы цикла значение  $M$  будет равно единице. Такому заголовку в программе не соответствует ни одна команда. В нашей программе станут лишними команды с адресами 01 и 02, и команду "F L0" надо будет заменить на "F L1".

В приведенной выше программе для переменной  $k$  необходимо было обязательно взять один из регистров R0 ... R3. Однако по тексту алгоритма этого не видно. Чтобы отразить данный факт, нужно описать переменную так, чтобы показать ее специальное использование. Для этого введем новый описатель счетчик. Начало алгоритма должно быть записано так:

алг факториал ;

числ  $M$ , факт ; счетчик  $k$  ;

Переменные с таким описанием учитываются при распределении памяти в первую очередь и им отводятся только регистры R0 ... R3.

**Замечание.** Эта конструкция цикла введена в А-язык для того, чтобы отразить особенности программирования на ПМК рассматриваемого семейства. В языках программирования обычно встречается более общая конструкция, соответствующая русской фразе

*меня переменная от нач.значение до конеч.значение шагом знач.  
шага повторять ...*

(в силу исторических причин записывается она обычно в виде  
*для переменная от ... до ... шаг ... цикл*

— почти в таком виде эта конструкция вошла в школьный учебник). Это очень удобная конструкция, позволяющая ясно и кратко записать многие алгоритмы. Например, с ее помощью тело алгоритма вычисления интеграла можно записать

инт: = 0 ;

меня  $x$  от  $a$  до  $b$  шагом  $(b - a)/N$  повторять

инт: = инт +  $f(x)$

кпвт

Здесь заголовок цикла вбирает в себя и установку начального значения  $x$ , и изменение  $x$  на каждом шаге цикла, и определение числа повторений цикла.

Но на ПМК нет команд, позволяющих реализовать такой цикл в общем виде. Введенные нами конструкции циклов соответствуют случаю

меня  $x$  от  $M$  до 1 шагом  $-1$  повторять

где  $M$  должно быть целым числом. Так как ПМК позволяет непосредственно реализовать только цикл с фиксированным конечным значением и шагом, то их можно не указывать.

### Задачи

1. Для построения графика функций нужно вычислить их значения в ряде точек. Написать программу, которая на отрезке  $[a, b]$  в точках, отстоящих на шаг  $h$  выдает значения функций  $f(x)$  и  $g(x)$  в следующем порядке: значение  $x$ , значение  $f$ , значение  $g$ .

2. Существенно более точное значение интеграла получается по формуле Симпсона при  $h = (b - a)/N$ :

$$\int_a^b f(x)dx \approx [f(a) + 4f(a + h) + 2f(a + 2h) + 4f(a + 3h) + 2 \dots + 4f(b - h) + f(b)] \frac{h}{3}.$$

Написать программу, реализующую интегрирование этим методом при заданных  $a$  и  $b$  и числе разбиений  $N$  ( $N$  — четное).

Для сравнения результатов взять ту же функцию и те же исходные данные, что и в рассмотренных программах для метода прямоугольников.

## 3. ИТЕРАЦИОННЫЕ ЦИКЛЫ

В приведенных выше примерах к моменту начала выполнения цикла было *известно*, сколько раз должно выполниться тело цикла. Однако такая ситуация имеет место далеко не всегда. Часто момент окончания повторений тела цикла определяется уже после того, как цикл начал выполняться. Вернемся к алгоритму вычисления интеграла и поставим другую задачу: определить значение  $T$ , при котором  $\int_0^T f(t)dt = S$ , где  $S$  задано. Физи-

ческий смысл этой задачи может быть весьма разнообразным: если  $f(t)$  — зависимость расхода жидкости от времени, то  $T$  — время, за которое наполнится (опустошится) данная емкость; если  $f(t)$  — изменение вертикальной скорости самолета (темпа набора высоты) в зависимости от времени, то  $T$  — время, за которое он выйдет на заданную высоту. Здесь мы уже не можем задаться некоторым  $N$  и набрать сумму площадей  $N$  прямоугольников, так как не ясно, сколько их нужно брать. В этом случае необходим другой подход: задавшись точностью  $\Delta T$ , с которой мы хотим получить результат, нужно прямоугольники шириной  $\Delta T$  брать до тех пор пока сумма их площадей не превысит  $T$ . Проверять, не пора ли завершать цикл, придется на каждом шаге.

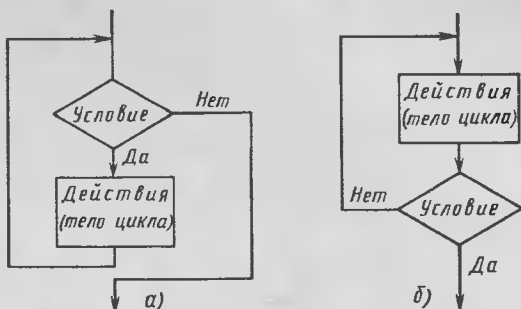


Рис. 8. Конструкция циклов :  
а – цикл пока; б – цикл до

Для построения подобных алгоритмов нам понадобятся еще две конструкции циклов. Первая из них имеет вид

**пока условие повторять действия кпвт**

Называется такая конструкция "цикл пока", схема ее показана на рис. 8, а, и работает она в точном соответствии со значением фразы на русском языке: проверяется условие, если оно истинно, то выполняются действия, затем опять проверяется условие и, если оно истинно, выполняются действия, и т.д. пока условие не станет ложным. А когда условие становится ложным, то происходит переход к выполнению оператора, расположенного за кпвт. Отметим, что если условие оказывается ложным при первой же проверке, то тело цикла *не выполнится ни разу*.

Вторая конструкция имеет вид

**повторять действия до условие кпвт**

Называется она "цикл до", схема ее показана на рис. 8, б, работает она также в соответствии со смыслом русской фразы, и отличается от предыдущей конструкции двумя моментами: во-первых проверка условия производится *после* выполнения тела цикла, вследствие чего тело цикла выполняется *не менее одного раза*, а во-вторых, в цикле до задается *условие окончания* выполнения цикла (при истинном условии цикл *завершается*), в то время как в цикле *пока* задается *условие продолжения* (при истинном условии цикл *продолжается*). Слово кпвт в цикле до можно не ставить, ибо и так понятно, где заканчиваются повторяемые действия.



С использованием введенных конструкций поставленную задачу определения верхней границы интеграла можно решить так:

алг граница интеграла 1 ;

числ  $S, \Delta T, T, \text{инт}$  ;

ввод  $S, \Delta T$ ;

инт: =  $T$ : = 0 ;

пока  $\text{инт} < S$  повторять

$\text{инт} := \text{инт} + f(T) \cdot \Delta T$ ;  $T := T + \Delta T$ ;

кпвт;

вывод  $T$ ;

кон

Здесь был использован цикл **пока**, но в этом алгоритме можно было применить и цикл **до**, тогда изменился бы только сам цикл:

повторять

$\text{инт} := \text{инт} + f(T) \cdot \Delta T$ ;  $T := T + \Delta T$

до  $\text{инт} \geq S$  кпвт ;

Различие в работе этих вариантов алгоритма заключается только в том, что при  $S = 0$  вариант с циклом **пока** даст правильный результат  $T = 0$ , а вариант с циклом **до** и в этом случае один раз выполнит тело цикла и даст результат  $T = \Delta T$ . (Обратите внимание, что после **до** и после **пока** стоят взаимно обратные условия).

Правила программирования этих циклов на ПМК легко вывести при рассмотрении схем циклов, аналогично их выводу для программирования конструкции **если**.

Правила программирования цикла **пока** таковы.

1. Программируется условие, приведенное к сравнению с нулем.
2. Ставится соответствующая команда условного перехода и пропускается ячейка под адрес.
3. Программируется тело цикла.
4. Служебное слово **кпвт** программируется командой "БП" с адресом начала проверки условия, после чего пропущенная на шаге 2 ячейка заполняется адресом очередной свободной ячейки.

# Правила программирования цикла до еще проще.

1. Программируется тело цикла.
2. Программируется стоящее после до условие, приведенное к сравнению с нулем.
3. Ставится соответствующая команда условного перехода с адресом начала тела цикла.

Запрограммируем алгоритм *граница интеграла 1*.

Адрес	Команды	А-язык	Комментарий
		алг граница интеграла 1 ;	
	$S \leftarrow P0$	числ $S$ ,	
	$\Delta T \leftarrow P1$	$\Delta T$ ,	
	$T \leftarrow P2$	$T$ ,	
	$инт \leftarrow PX$	$инт$ ;	Ясно, что переменную <i>инт</i> удобно хранить в $PX$
00	$x \rightarrow P0$	ввод $S$ ,	
01	С/П $x \rightarrow P1$	$\Delta T$ ,	Ввод и присваивание програм- мируем как обычно.
03	0 $x \rightarrow P2$	$инт := T := 0$ ;	
05	$\rightarrow B \uparrow$	пока	Сняли копию с переменной <i>инт</i> для операции.
06	$P \rightarrow x0 -$	$инт - S$	Программируем цикл: вычисляем левую часть условия, приведенного к сравнению с нулем, и ставим команду условного перехода с пока пустым адресом.
08	$F X < 0$	$< 0$	
09		повторять	
10	$\leftrightarrow P \rightarrow x2 \quad F x^2$	$инт +$	Тело цикла (два присваивания) программируем как обычно, функцию взяли прежнюю $f(x) = x^2/2$ .
13	$2 \div \quad P \rightarrow x1$	$f(T) \cdot \Delta T$	
16	$x +$	$\rightarrow инт$ ;	
18	$P \rightarrow x2 \quad P \rightarrow x1$	$T + \Delta T$	
20	$+ \quad x \rightarrow P2 \leftrightarrow$	$\rightarrow T$ ;	
23	БП	кпвт	Программируем слово <i>кпвт</i> : пишем "БП", адрес начала проверки условия;
24	05		Очередной свободный адрес — 25 заполняет пропущенную ячейку
25	...		
25	$\rightarrow P \rightarrow x2 \quad С/П$	вывод $T$ ; кoi	

Обратите внимание на то, что хотя переменная *инт* используется еще и при проверке условия, но все же ее хранение в РХ оказалось выгодным: пришлось лишний раз снять копию (команда по адресу 05). Впрочем, за движением хранимой в РХ переменной нужно следить внимательно: она "задвигается" в РУ как при проверке условия, так и при выполнении второго оператора тела цикла, поэтому возвращать ее в РХ приходится дважды (команды по адресам 10 и 22). Так что в количестве команд мы не выиграли, а сэкономили только один адресуемый регистр. В данной программе это не имеет никакого значения, здесь лишь иллюстрируется прием, с помощью которого можно сэкономить регистр, если регистров не хватает.

Теперь программируем вариант с циклом до.

Адрес	Команды	А-язык	Комментарий
...	...	алг граница интеграла 2 ; Регистры прежние. Первые пять команд без изменений.	
05	V↑		Об этой команде рассказано ниже.
06	↔ П→х 2 F x <sup>2</sup>	повторять	Тело цикла программируется без изменений.
09	2 ÷ П→х 1	инт + f(T).	
12	х +	ΔT → инт ;	
14	П→х 2 П→х 1	T + ΔT	
16	+ х→Н 2 ↔	→ T ;	
19	V↑ П→х 0 -	до инт - S,	Проверяем выполнение условия и делаем условный переход на начало тела цикла.
22	F X ≥ 0 06	≥ 0 кпвт ;	
24	П→х 2 С/П	вывод T ; кон	

Команду "V↑" по адресу 05 пришлось добавить потому, что после выполнения тела цикла и проверки условия значение *инт* оказывается в РУ и выполнение тела цикла приходится начинать с возвращения его в РХ. Чтобы так же можно было работать и на первом проходе цикла, приходится искусственным приемом — командой "V↑" — заслать значение *инт* в РУ. Эти примеры окончательно очерчивают круг ситуаций, в которых выгодно хранить переменную в РХ. Как видим, циклы до и пока этот круг сузили, а причина прежняя — не вполне корректная работа со стеком команд условного перехода.

Выполним эти программы, взяв  $S = 1$ ,  $\Delta T = 0.2$ . Получим результат 2 через 1 мин счета у первого варианта программы и

через 55 с — у второго (во втором варианте в цикле выполняется на одну команду меньше). Точное значение должно быть  $\sqrt[3]{6} \approx 1.8171205$ .

Совершим небольшой экскурс в математику, связанный с вопросами точности. Выше у нас получился результат, отличающийся от точного менее чем на шаг. Но если взять, например,  $\Delta T = 0.5$ , получим результат 2.5, отличающийся от точного более чем на шаг. При этом значение *инт* (оно находится в PZ, его можно увидеть на индикаторе, выполнив дважды команду "FO") равно 1.875, т.е. гораздо больше, чем надо. Дело в том, что когда  $T = 2$  и значение интеграла уже больше, чем  $S$ , значение переменной *инт*, равное сумме площадей прямоугольников еще меньше, чем  $S$  (см. рис. 7). Поэтому сказать, что получаемый по такому алгоритму результат имеет погрешность  $\Delta T$ , нельзя, для оценки получаемой погрешности необходимы некоторые математические расчеты, которые выходят за рамки нашего рассмотрения.

В данном примере можно было использовать как цикл до, так и цикл пока, однако эти циклы не взаимозаменяемы. Как правило, выбор одного из них однозначен. Рассмотрим следующую задачу. Имеется некоторая функция, непрерывная на отрезке  $[a, b]$ ; значения  $f(a)$  и  $f(b)$  имеют разные знаки и  $f''(x)$  на  $[a, b]$  не меняет знака. Нужно найти корень этой функции (в данных условиях он обязательно существует и он единственный). Один из способов приближенного численного решения уравнения  $f(x) = 0$  носит название "метод хорд" и заключается в следующем. Из точек  $a$  и  $b$  выберем ту, в которой знак функции совпадает со знаком второй производной. Пусть это будет точка  $a$ . Точнее, будем считать точкой  $a$  ту точку, где  $f(a)$  и  $f''(a)$  имеют одинаковые знаки; то, что при этом точка  $a$  может оказаться правее точки  $b$ , роли не играет — итоговая формула во всех случаях получается одинаковой. В качестве первого приближения к корню  $x_0$  возьмем  $b$ . Проводим хорду (откуда и название метода) между точками  $(a, f(a))$  и  $(x_0, f(x_0))$ . Пересечение этой хорды с осью  $X$  дает следующее приближение  $x_1$  (рис. 9). Проводим хорду между точками  $(a, f(a))$  и  $(x_1, f(x_1))$  и получаем следующее приближение  $x_2$  и т.д. Будем продолжать этот процесс, пока расстояние между очередными приближениями не станет достаточно малым:  $|x_i - x_{i-1}| < \epsilon$ , где  $\epsilon$  задано. Как и в предыдущем примере это не означает, что расстояние между  $x_i$  и истинным значением корня меньше  $\epsilon$ , однако дальнейшие расчеты уже не дадут более точного результата, а точная оценка погрешности результата требует некоторых математических

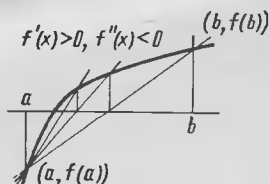
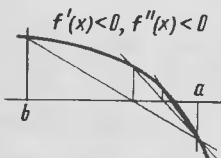
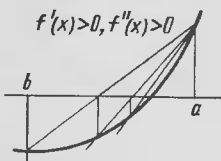
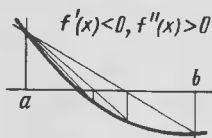


Рис. 9. Различные варианты сочетания знаков 1-й и 2-й производных при решении уравнения методом хорд



выкладок. Формула вычисления следующего приближения через предыдущее имеет вид:

$$x_{i+1} = a + f(a) \frac{a - x_i}{f(a) - f(x_i)}.$$

В алгоритме достаточно иметь две переменных, одна из которых имеет значение предпоследнего приближения (ее назовем  $x_n$ ), а другая — значение очередного приближения (эту переменную назовем  $x$ ). Тело алгоритма можно записать следующим образом:

**ввод**  $a, b, \epsilon; x := b;$

**повторять**  $x_n := x; x := a + (f(a) \cdot (a - x)) / (f(x) - f(a))$

**до**  $|x_n - x| < \epsilon;$

Сразу отметим, что просто заменить здесь цикл **до** циклом **пока** не удастся, так как для проверки условия нужно иметь и значение  $x$  и значение  $x_n$ , но для получения значения  $x_n$ , надо, чтобы тело цикла выполнилось хотя бы один раз.

Применяя те приемы оптимизации алгоритма, которые рассматривались ранее, можно увидеть следующее: значение  $f(a)$  целесообразно вычислить один раз до начала цикла и запомнить во вспомогательной переменной; значение  $b$  используется только один раз, а значит, его можно ввести непосредственно в переменную  $x$ . В итоге получаем усовершенствованный вариант алгоритма:

$fa := f(\text{ввод } a); \text{ ввод } \{b\} x; \epsilon;$

**повторять**  $x_n := x; x := a + (fa \cdot (a - x)) / (f(x) - fa)$

**до**  $|x_n - x| < \epsilon;$

Используя возможность присваивания внутри выражения еще улучшаем работу цикла:

**повторять**

до  $| (x := (a + (fa \cdot (a - (xn := x)) / (f(x) - fa))) - xn | < \epsilon;$

Здесь уже все тело цикла фактически вошло в выражение, стоящее в условии и после до. Но это длинное выражение записано не лучшим образом, так как если в функции много операций, то может не хватить стека. Его следует переписать так, чтобы по возможности операции выполнялись в порядке их записи, мы это и сделаем в окончательном варианте алгоритма:

**алг метод хорд ;**

**числ**  $a, x, \epsilon, xn, fa ;$

$fa :=$  **ввод**  $a ;$  **ввод**  $\{b\} x, \epsilon;$

**повторять**

до  $| (x := (fa \cdot (a - (xn := x)) / (f(x) - fa)) + a) - xn | < \epsilon;$

**вывод**  $x ;$

**кoi**

Здесь еще раз наглядно проиллюстрирована основная идея построения хороших программ: начинаем с простейшего, наиболее очевидного алгоритма, а затем "чистим" его; в итоге еще на уровне записи на А-языке получаем качественный алгоритм, который затем естественно превращается в качественную программу.

Этот алгоритм является типичным представителем так называемых *итерационных* алгоритмов, реализующих *итерационные* методы (методы последовательных приближений), когда решение получается путем постепенного приближения к нему с контролем на каждом шаге степени приближения. Реализация этих алгоритмов, как правило, осуществляется с помощью циклов **до** и **пока**, поэтому такие циклы называют итерационными.

Программируем этот алгоритм, взяв функцию  $f(x) = e^x - 5$ . Для этой функции также можно легко получить решение вручную:  $x = \ln 5 = 1.65...$  Как обычно, для иллюстрации работы алгоритма берется простая функция.

Адрес	Команды	А-язык	Комментарий
		алг метод хорд ;	
	$a - P0$	<b>числ</b> $a,$	Регистры распределяем подряд, по мере чтения описаний переменных.
	$x - P1$	$x,$	
	$\epsilon - P2$	$\epsilon,$	

Адрес	Команды	А-язык	Комментарий
	$xn - P3$ $fa - P4$	$xn,$ $fa ;$	
00	$x \rightarrow P0$	$F e^x$	$e^{\text{ВВод } a}$
02	5 — $x \rightarrow P4$	$-5 \rightarrow fa ;$	Введенное значение $a$ запоминается и используется в операции.
05	C/П	$x \rightarrow P1$	$\text{ВВод } \{b\} x,$
07	C/П	$x \rightarrow P2$	$\epsilon ;$ Значение $b$ вводится в регистр, отведенный переменной $x$ .
09		повторять	С адреса 09 начинается тело цикла, которое фактически пусто.
09	$\rightarrow P \rightarrow x 0$	до $((a -$	
10	$P \rightarrow x 1$	$x \rightarrow P3$	$(xn := x)$
12	— $P \rightarrow x 4$	$x$	$\cdot fa) /$
15	$P \rightarrow x 1$	$F e^x$	$(e^x$
17	5 —	$-5$	
19	$P \rightarrow x 4$	— $\div$	$-fa)$
22	$P \rightarrow x 0$	+	$+a$
24	$x \rightarrow P1$	C/П	$\rightarrow x)$
26	$P \rightarrow x3$	—	Добавили команду останова для контроля программы.
28	$F x^2$	$F \sqrt{\phantom{x}}$	$-xn $ Самый быстрый способ получить модуль: возвести в квадрат и извлечь корень.
30	$P \rightarrow x 2$	—	Завершаем программирование цикла.
32	$F X < 0$	09	
34	$P \rightarrow x 1$	C/П	Вывод $x ;$ кон

Команда "C/П", добавленная по адресу 25, остановит программу в момент, когда на РХ будет значение очередного приближения. Фактически мы запрограммировали выражение

$$\text{Вывод } (x := ((a - (xn := x) \cdot fa) / (f(x) - fa)) + a) - xn| < \epsilon$$

Зададим теперь  $a = 2$  (после нажатия клавиши "C/П" на индикаторе увидим значение  $fa = 2.3890557$ ),  $b = 0$ ,  $\epsilon = 0.05$ . При остановах по адресу 25 на индикаторе последовательно увидим значения 1.2521412, 1.5408455, 1.5967882, 1.6071238 — последнее отличается от предыдущего менее чем на 0.05, и после следующего нажатия клавиши "C/П" на индикаторе высветится опять оно же: проверка счетчика адреса показывает, что останов произошел по адресу 35, т.е. на индикаторе результат. Если теперь вычислим значение функции от находящегося в РХ числа, получим  $-0.0115573$ , т.е. достаточно близкий к нулю результат.

Возвращаясь к вопросу точности, отметим два момента. Во-первых, корень вычислялся с точностью 0.05, поэтому выписывать все цифры с индикатора нет смысла: можно сказать, что корень приблизительно равен 1.60 и при этом последняя цифра уже будет неточной. Во-вторых, из того, что разность между соседними приближениями стала меньше 0.01, не следует, что от значения 1.60... до истинного значения корня разность тоже меньше 0.01 или хотя бы меньше 0.05 (повторяем это еще раз). К этому вопросу мы еще вернемся.

Теперь можно убрать команду "С/П" (заменить ее командой "К НОП") и пользоваться программой. Работа с программой при разных значениях  $a$ ,  $b$  и  $\epsilon$  будет естественно давать почти одинаковые результаты. Более интересна эксплуатация программы с заменой команд вычисления функции (адреса 01 ... 03 и 16 ... 18) командами вычисления какой-либо другой функции. Однако здесь мы ограничены только такими функциями, вычисление которых укладывается в три команды. Как уйти от этого ограничения, будет рассмотрено в п. 8 гл. 3.

Проведем еще один эксперимент, не меняя программу (и не убирая команды "С/П"). Если при  $\epsilon = 0.05$  взять  $a = 0$  и  $b = 2$ , т.е. в качестве точки  $a$  взять такую, где знаки  $f(a)$  и  $f''(a)$  не совпадают, то при остановках будем получать следующие результаты: 1.25214212, 2.0051709, 1.2478965, 2.0102975 и т.д. Как видим, значения  $x$  колеблются вокруг корня все больше и больше удаляясь от него. Если убрать команду "С/П" и запустить программу с этими исходными данными, то она *зациклится*: условие никогда не выполнится, а значит, программа будет работать бесконечно. Когда программа работает слишком долго и есть подозрение, что она зациклилась, ее можно остановить нажатием клавиши "С/П". Если на рис. 9 начать проводить хорды из неподходящей точки, где  $f(a) \cdot f''(a) < 0$ , то на рисунке можно увидеть, что при этом происходит. Можно подобрать и такую функцию, для которой при определенных значениях  $a$ ,  $b$  и  $\epsilon$  мы все же получим результат, не имеющий никакого отношения к корню. Это еще раз показывает, что обеспечение правильности исходных данных, корректного использования программы и интерпретация результата *всегда остаются за человеком*.

Последний пример иллюстрирует одно из ключевых отличий итерационных циклов от цикла повторять ... раз: *итерационные циклы могут зацикливаться, а цикл повторять ... раз — никогда*.

Разбирая разные виды циклов, мы видели, что они отличаются способом определения завершения выполнения цикла.



Существует еще один вид циклов — *бесконечный* цикл, который, как следует из его названия, сам по себе не заканчивается никогда. Конструкция такого цикла имеет вид

**повторять действия кпвт**

Правила его программирования чрезвычайно просты.

1. Программируется тело цикла.
2. Служебное слово **кпвт** программируется командой "БП" с адресом начала тела цикла.

Чтобы понять, где применяются такие циклы, вспомним самую первую программу — вычисление параметров цилиндра. Эта программа многократно выполнялась для получения результатов по различным исходным данным. Но ею удобнее пользоваться, если ее алгоритм записать в виде

**повторять**

**вывод (вывод ( $S := \text{ввод } \{h\} \cdot \pi \text{ ввод } d) \cdot d/4$ )**

**кпвт**

Программа тогда принимает следующий вид:

Адрес	Команды		А-язык	Комментарий
00	Ф π	х	<b>повторять</b>	Перед работой программы набираем первое значение $h$ , при останове — значение $d$ . Считываем значение площади, при следующем останове считываем значение объема и набираем новое значение $h$ .
02	С/П	х → П 0	( $\text{ввод } \{h\} \cdot \pi$	
05	х → П 1	С/П	$\cdot \text{ввод } d$	
07	П → х 0	х	$\rightarrow S) \rightarrow \text{вывод}$	
09	4	С/П	$\cdot d/4 \rightarrow$	
12	БП	00	<b>вывод</b> <b>кпвт</b>	

При останове для считывания последнего результата набираем новое значение первого исходного данного  $h$ , и после нажатия клавиши "С/П" программа повторяет расчет для новых значений данных. Когда все данные обработаны, микрокалькулятор просто выключается или же в него вводится другая программа. Позже увидим и другие применения бесконечных циклов.

Справедливости ради отметим, что в этой программе несколько нарушены строгие правила программирования: **ввод** не является первой исполняемой командой в алгоритме (перед ним стоит слово **повторять**), поэтому первой должна была бы

стоять команда "С/П", которую мы "сэкономили" за счет того, что при останове по адресу 11 и читали результат, и набирали новое исходное значение. Но не будем вводить специальные средства, чтобы отразить в записи на А-языке эти возможности ПМК, так как такая программа предназначается только для иллюстрации применения бесконечных циклов.

#### Задачи

1. Для вычисления приближенных значений функций часто используют их разложение в ряды. Вычислить с помощью ряда значение функции интегральный гиперболический синус  $\text{shi}(x)$ , который определяется формулой

$\text{shi}(x) = \int_0^x \frac{\text{sh}t}{t} dt$  и вычисляется с помощью ряда

$$\text{shi}(x) \approx \frac{x}{1 \cdot 1!} + \frac{x^3}{3 \cdot 3!} + \dots + \frac{x^{2n+1}}{(2n+1)(2n+1)!} + \dots$$

Вычисление вести до тех пор, пока очередной член ряда  $\frac{x^{2n+1}}{(2n+1)(2n+1)!}$  не станет по модулю меньше заданного  $\epsilon$ .

2. Известно, что  $f(a) > 0$ . Переходя к точкам  $a + H, a + 2H, \dots$  найти точку, в которой  $f(x) < 0$ , если известно, что функция при больших  $x$  становится отрицательной. Такой прием может быть применен для выделения отрезка, на котором затем ищут корень, например, тем же методом хорд.

## 4. ОБРАБОТКА ТАБЛИЦ

Одной из широко распространенных на практике ситуаций является ситуация, когда имеется некоторое количество однородных данных, над которыми нужно выполнить одинаковые операции. Например, при покупке в магазине нескольких товаров для каждого товара нужно умножить стоимость единицы товара на количество купленных единиц и суммировать полученные результаты. Однородные данные, например те же цены на товары, обычно сводят в таблицы. Фактически можно сказать, что таблица — это несколько переменных, каждая из которых имеет свое значение, и все эти переменные имеют одно и то же имя, в нашем примере это имя — "цена". Для выделения конкретной переменной применяют дополнительное указание, которое называют *индексом*. Чаще всего индекс — это целое число, т.е. переменные перенумеровываются и вместе с именем переменной указывается, какая именно из переменных с данным именем имеется в виду.

В языках программирования таблицы называют *массивами*. Для описания массива на А-языке вслед за его именем задается *диапазон изменения индекса*. Например:

числ цена [1:10]

означает, что у нас имеется 10 переменных, называемых ЦЕНА, с номерами от 1 до 10. Для указания конкретной переменной выражение, задающее значение индекса, пишем вслед за именем в квадратных скобках:

цена [K - 1]    цена [4]

Диапазон индексов не всегда удобно начинать с единицы, например, таблицу каких-то данных за последнюю неделю удобно описать

числ температура [-6:0]

где температура [-5] — это значение температуры пятидневной давности, а температура [0] — значение за сегодняшний день.

Вообще говоря, часто удобно использовать не один, а два или более индексов: элемент в матрице определяется номером строки и номером столбца, стул в кинотеатре имеет номер ряда и номер места и т.д. Такие таблицы называются двухмерными, трехмерными и т.д. — в зависимости от количества индексов. Но малая емкость памяти ПМК практически не позволяет работать с многомерными массивами (таблицами): в памяти ПМК можно разместить двухмерную таблицу размером не более чем  $3 \times 4$  (3 строки, 4 столбца) поэтому многомерные массивы рассматриваться не будут.

Обработка таблиц в основном заключается в выполнении одинаковых действий над элементами таблицы (массива), индексы (номера) которых изменяются по определенному закону. Это означает, что обработка массива есть цикл, внутри которого изменяется индекс, а также встречается *индексированная переменная*, т.е. имя массива с указанием индекса. Простейший пример обработки массива — получение суммы его элементов:

сумма: = 0 ; k: = 1 ;

повторять M раз сумма: = сумма + A [k]; k: = k + 1 кпвт

В этом примере роль индекса играет переменная k. Тело цикла повторяется M раз (где M — число элементов в таблице-массиве). При первом выполнении тела цикла k имеет значение 1, следовательно, к значению переменной сумма будет добавляться значение A [1]. Далее k принимает значение 2, и на следующем

проходе цикла к значению *сумма* будет добавляться значение *A* [2], и т.д.

В памяти ПМК массив будет занимать несколько адресуемых регистров. Практически всегда это несколько рядом расположенных регистров. При организации программы нужно сделать так, чтобы одна и та же команда при разных проходах цикла вызывала значение из разных регистров. Если массив занимает регистры P5, P6, P7 ..., то при первом исполнении тела цикла нужно вызвать значение из P5, при втором — из P6, при третьем — из P7 и т.д. Коды команд "П → х" и "х → П" таковы, что их вторая цифра — это номер регистра, с которым работает команда (—, L, C, G, и E — это соответственно 10, 11, 12, 13 и 14: читателям, знакомым с шестнадцатеричной системой счисления, это должно быть понятно). И если бы можно было работать с кодом команды, как с числом, то достичь желаемого результата можно, изменяя саму команду путем добавления к ее коду по единице. Но в программной памяти изменить команду, когда ПМК работает в автоматическом режиме, невозможно. Следовательно, номер регистра, из которого нужно читать или в который нужно писать значение, должен задаваться не в команде, а одним из регистров, доступных для изменения при работе в автоматическом режиме. Нужные средства на ПМК есть — это команды с *косвенной адресацией*.

В командах "П → х м" и "х → П м" адрес нужного регистра задается прямо (непосредственно) в команде, поэтому такие команды называют командами с прямой (непосредственной) адресацией. Но на ПМК имеется еще две группы команд: "К П → х м" и "К х → П м", где адрес регистра, из которого идет чтение или в который идет запись, задается не прямо, а косвенно, не в самой команде, а в другом регистре, номер последнего задан в команде, т.е. в регистре № м. Эти команды называют соответственно командами косвенного чтения и косвенной записи. Для их исполнения или запоминания в программной памяти требуется нажатие трех клавиш: "К", "П → х" или "х → П", а затем клавиши с номером регистра. Исполнение этих команд включает два шага:

модификация адреса в заданном в команде регистре;  
выполнение собственно чтения или записи.

Способ модификации адреса зависит от регистра: содержимое P0 ... P3 уменьшается на единицу, содержимое P4 ... P6 увеличивается на единицу, а содержимое прочих регистров не изменяется. Если в регистре было дробное положительное число, то

от него отбрасывается дробная часть, а оставшаяся целая часть подвергается модификации.

Модифицированное содержимое заданного в команде регистра рассматривается как номер другого регистра (10 соответствует RA, 11 — RB, 12 — RC, 13 — PD, 14 — PE) и именно этот регистр участвует в процедуре чтения или записи. Если в указанном в команде регистре находится отрицательное или слишком большое число, то возникает ситуация "эффект не определен": ПМК команду выполняет и чтение-запись производится с каким-то регистром, номер которого определить трудно.

Рассмотрим работу команд с косвенной адресацией на примерах. Запишем в R0 значение 0, в R1 — значение 1 (в каждый регистр попадает число, равное номеру регистра) и т.д.

Клавиши	Индикатор	Комментарий
К П → x 4	5.	Число 4, бывшее в R4, увеличилось на 1 и значение из R5 было считано в RX.
К П → x 4	6.	По той же команде теперь происходит чтение из R6, так как в R4 было значение 5, а после модификации стало 6, в чем легко убедиться.
П → x 4	00000006.	
К П → x 4	7.	По той же команде теперь идет чтение из R7.
К x → П 3	7.	Так как в R3 было 3, то после модификации получим 2: запись пойдет в R2, а по следующей команде — в R1.
П → x 2	7.	Теперь в R2 — значение 7,
П → x 1	7.	в R1 — тоже 7,
П → x 3	00000001.	а в R3 — номер регистра, с которым работали последним.
К П → x 8	00000008.	Чтение произведено из R8, но при модификации значения в R8 не изменились.
П → x 8	00000008.	
3.4	3.4	
x → П В	3.4	В RB зашли дробное число, по команде "К П → x В" прочитано значение из R3, а при модификации от значения RB отбрасывается дробная часть.
К П → x В	00000001.	
П → x В	00000003.	
/ — /	—3.	Теперь засылаем в RB отрицательное число, чтобы посмотреть, что будет.
x → П В	—3.	
К П → x В	13	Прочитали 13, вероятно — из PD, но почему регистр № — 3 — это PD?
П → x В	—99999993.	

Работа по командам косвенной адресации изучена, но выполнение этих команд не дает нам непосредственного решения поставленной задачи, так как они не соответствуют тому, что в языках программирования называется индексом. Индекс — это

позиция элемента, отсчитываемая *от начала массива*, а для команд косвенной адресации нужен адрес, т.е. номер регистра, отсчитываемый *от начала всех регистров*, а не только тех, которые отведены массиву. Поэтому в наш А-язык придется ввести средство, которое не характерно для языков высокого уровня (хотя его аналоги можно найти и там), а является специфическим "калькуляторным" средством. Конструкция

### адрес $A$

означает номер того регистра, который отведен переменной  $A$ . В качестве  $A$  может употребляться также имя переменной-массива с индексом-константой. Если переменная  $C$  занимает РЗ, а массив  $P[2:7]$  занимает регистры с 5-го по 10-й включительно, то можно написать, например:

$k := \text{адрес } C; k := \text{адрес } P[4]; k := \text{адрес } P[1]; k := \text{адрес } P[8];$

При этом переменная  $k$  получит соответственно значения: 3, 7, 4 и 11. Обратите внимание на то, что элементов  $P[1]$  и  $P[8]$  в массиве не существует, но если бы они были, то их адресами были бы значения 4 и 11 соответственно — именно такие значения и получила переменная  $k$ .

Переменная, получившая значение адреса, как правило, используется в конструкциях косвенной адресации, имеющих вид

$[k] \quad [^+k] \quad [-k]$

и означающих соответственно косвенную адресацию без модификации, с предварительным увеличением адреса (*с автоувеличением*) и с предварительным уменьшением адреса (*автоуменьшением*). Эти конструкции могут использоваться и в качестве операндов в выражении и в качестве переменных, которым присваивается значение. Одна переменная может использоваться только в одном виде косвенной адресации (например, всегда без модификации, или всегда с автоувеличением). Описывать такие переменные будем с описателями *косв*, *косв—* и *косв+* (в зависимости от того, в каком виде адресации она используется).

С применением новых средств А-языка алгоритм суммирования таблицы можно записать в разных вариантах, используя разные виды адресации.

алг сумма таблицы 1 ;

числ сумма ; косв  $k$  ; числ  $A$  [1:9] ;

Рввод  $A$  ; сумма: = 0 ;  $k$ : = адрес  $A$  [1] ;

повторять 9 раз сумма: = сумма +  $[k]$  ;  $k$ : =  $k + 1$  кпвт ;

вывод сумма ;

кон

Здесь применена ручная операция Рввод, чтобы не отвлекаться на программирование ввода массива. Заметим, что для алгоритма несущественно, какие именно регистры занимают элементы массива, существенно лишь то, что они занимают соседние регистры.

Сформулируем правила программирования новых конструкций.

1. Для переменных с описателями косв, косв+ и косв- регистры отводятся в первую очередь, как и для счетчиков, так как для них подходят не все регистры.

2. Память под массивы отводится в последнюю очередь.

3. Конструкция адрес программируется набором числа, соответствующего номеру регистра, занимаемого стоящей после слова адрес переменной (или элементом массива).

4. Конструкции  $[k]$ ,  $[*k]$  и  $[-k]$  программируются командой "К П  $\rightarrow x$   $i$ ", если они используются в выражении, и командой "К  $x \rightarrow$  П  $i$ ", если они используются в качестве той переменной, которой присваивается значение.

При отведении памяти для переменной с описателем косв обычно берут регистр PD или PC (регистр "с края" множества регистров), так как если взять, например, P7, то остается мало смежных регистров, которые можно было бы отвести под массив.

Программируем алгоритм.

Адрес	Команды	А-язык	Комментарий
		алг сумма таблицы 1 ;	
	счетчик — P0		Сначала берем регистр для счетчика цикла, затем для переменной с описателем косв, далее отводим регистры для остальных переменных.
	$k$ — PD	косв $k$ ;	
	сумма — PX	числ сумма,	далее отводим регистры для остальных переменных.
	$A$ [1:9] — P2 ÷ PA	$A$ [1:9] ;	

Адрес	Команды	А-язык	Комментарий
		Рввод $A$ ;	Для этого оператора нет команд в программе.
00	2	$x \rightarrow П D$	$k := \text{адрес } A [1]$ ; $A [1]$ лежит в $P2$ .
02	9	$x \rightarrow П 0$	повторять 9 раз
04	0	сумма: = 0;	переноса после него оператор "сумма: = 0".
05		$К П \rightarrow x D$	На каждом проходе цикла из памяти будут браться разные данные. Увеличиваем значение $k$ и возвращаем в $PX$ значение переменной $сумма$ .
06		$+$	
07		$П \rightarrow x D$	$k := k + 1$
10		$х \rightarrow П D$	
12	$F L 0$	05	кпвт
14	С/П	вывод сумма; кон	Завершаем программирование цикла.

Если в этом алгоритме заменить константу 1, которую мы на каждом шаге прибавляем к значению  $k$ , на 2, и соответственно заменить число повторений цикла на 5, то получим алгоритм, суммирующий элементы с нечетными номерами. Аналогичными заменами можно получить суммирование элементов через 2, от конца массива к началу и т.д. Но для случая, когда нужно просуммировать или обработать другим способом каждый из элементов таблицы-массива, можно существенно улучшить программу, применив косвенную адресацию с автоувеличением или автоуменьшением.

алг сумма таблицы 2;

числ сумма ; косв+  $k$  ; числ  $A [1:9]$  ;

Рввод  $A$  ; сумма: = 0 ;  $k := \text{адрес } A [0]$  ;

повторять 9 раз сумма: = сумма + [ $^+k$ ] кпвт ;

вывод сумма

кон

Сразу видно, что программа будет работать быстрее, так как в теле цикла остался один оператор.



Адрес	Команды	А-язык	Комментарий
	счетчик — P0 $k$ — P4	алг сумма таблицы 2 ; косв+ $k$ ;	Описатель косв+ требует одного из регистров P4 ... P6.
	сумма — PX A [1:9] — P5 ÷ PD	числ сумма, A [1:9]	
		Рввод A;	
00	4 $x \rightarrow П 4$	$k := \text{адрес}$ A [0];	
02	9 $x \rightarrow П 0$	повторять 9 раз	
04	0	сумма: = 0 ;	
05	K П $\rightarrow x 4$ +	сумма: = сумма+ [ $+k$ ]	
07	F L0      05	кпвт	
09	С/П	вывод сумма ; кон	

Использование косвенной адресации с автоувеличением сэкономило не только четыре команды увеличения  $k$ , но и пятую команду, возвращавшую значение суммы в PX. Впрочем, кроме преимуществ эта программа имеет и недостатки; если в предыдущем варианте можно было обработать массив из 12 элементов, расположив их в P1 ... PC (и соответственно изменив константы в программе), то в последнем варианте программы взят предельный размер массива. Требование разместить  $k$  в одном из регистров P4 ... P6 фактически приводит к тому, что мы не можем использовать P1, P2 и P3. С этой точки зрения выгоднее применить адресацию с автоуменьшением, так как она позволяет взять регистры, расположенные ближе к краю массива регистров и иметь больше смежных регистров для размещения массива. Здесь можно пойти двумя путями: во-первых, воспользоваться тем, что алгоритм допускает обработку элементов от конца к началу, а во-вторых, расположить элементы массива в обратном порядке. Соответственно, получаем:

алг сумма таблицы 3 ;

числ сумма, A [1:9] ; косв —  $k$  ;

Рввод A; сумма: = 0 ;  $k := \text{адрес A [10]}$  ;

повторять 9 раз сумма: = сумма +  $[-k]$  кпвт ;

вывод сумма;

кон

алг сумма таблицы 4 ;

числ сумма, А [9:1] ; косв —  $k$  ;

Рввод А ; сумма: = 0 ;  $k$ : = адрес А [0] ;

повторять 9 раз сумма: = сумма +  $[-k]$  кпвт ;

вывод сумма;

кон

В алгоритме *сумма таблицы 4*, чтобы подчеркнуть, что элементы располагаются в обратном порядке, поменяли местами границы диапазона изменения индекса. Оба варианта дают одинаковую программу

Адрес	Команды		А-язык	
			Вариант 1	Вариант 2
			алг сумма таблицы 3	алг сумма таблицы 4
	счетчик — P0			
	$k$ — P1		косв — $k$ ;	косв — $k$ ;
	сумма — PX		числ сумма,	числ сумма,
	$A - P2 \div PA$		А [1:9];	А [9:1];
			Рввод А ;	Рввод А ;
00	1 1	$x \rightarrow П 1$	$k$ : = адрес А [10];	$k$ : = адрес А [0] ;
03	9	$x \rightarrow П 0$	повторять 9 раз	повторять 9 раз
05	0		сумма: = 0 ;	сумма: = 0 ;
06	К П $\rightarrow x 1$	+	сумма: = сумма + $[-k]$ ;	сумма: = сумма + $[-k]$ ;
08	F L0	06	кпвт	кпвт
10	С/П		вывод сумма; кон	вывод сумма; кон

В этих вариантах ничто не мешает использовать регистры РВ ... РD и довести размер обрабатываемого массива до 12 элементов. Сравним характеристики полученных вариантов программы.

Характеристика	Вариант 1	Вариант 2	Вариант 3	Вариант 4
Длина программы	14	9	10	10
Время исполнения, с	26	12	12	12
Максимальный размер массива	11	9	12	12

Как видим, преимущества адресации с автоизменением очевидны и по-возможности следует использовать адресацию с автоуменьшением.

К сожалению, при необходимости на одном проходе цикла использовать один и тот же элемент массива более одного раза адресация с автоизменением неприменима. Например, пусть надо каждый из  $M$  элементов массива умножить на 2. Тело алгоритма

$k := \text{адрес } A [0] ;$

повторять  $M$  раз  $[^+k] := 2 \cdot [^+k]$  кпвт

результата не даст, так как значение  $k$  будет увеличиваться и перед чтением значения, и перед записью, в результате удвоенное значение будет записываться в соседний элемент массива. А вариант

$k := \text{адрес } A [0] ;$

повторять  $M$  раз  $[^+k] := 2 \cdot [k]$  кпвт

к сожалению, невозможен: одну и ту же переменную  $k$  нельзя использовать в косвенной адресации и с автоувеличением, и без автоувеличения.

**Замечание.** Было бы гораздо удобнее, чтобы наличие или отсутствие автоизменения было не функцией регистра, а функцией команды, т.е., чтобы с каждым регистром можно было выполнить команду "К П  $\rightarrow$  х  $i$ " без автоизменения и команды "К + П  $\rightarrow$  х  $i$ " и "К - П  $\rightarrow$  х  $i$ " с автоизменением любого вида.

Наилучший выход из положения в подобных случаях — это взять две переменные:

косв +  $k, k1$ ;

$k := k1 := \text{адрес } A [0] ;$

повторять  $M$  раз  $[^+k1] := 2 \cdot [^+k]$  кпвт

Если же один и тот же элемент массива используется на каждом шаге цикла многократно, то надо снять копию с этого элемента либо с указывающей на него косвенной переменной:

косв+  $k$  ; числ  $C$  ;

повторять  $M$  раз

$C := [^+k]$  ; *< работа с переменной  $C$  >*

кпвт

(этот вариант годится только в случае, если элемент массива не должен измениться в результате обработки) или

косв+  $k$  ; косв  $k1$  ;

повторять  $M$  раз

$k1 := k$  ;

*< многократная работа с  $[k1]$  и однократная — с  $[^+k]$  >*

кпвт

Например, при необходимости получить

$$\sum_{i=1}^M ((a_i + 1) \cdot a_{i-1} \cdot (a_{i-1} - 2))$$

можно записать тело алгоритма так:

косв+  $k$  ; косв  $k1$  ; ...

...

$k := \text{адрес } A[0]$  ;

повторять  $M$  раз

$k1 := k$  ; сумма := сумма +  $([^+k] + 1) \cdot ([k1]) \cdot ([k1] - 2)$

кпвт

Программы обработки массивов обычно построены так, что они почти не зависят от размеров обрабатываемого массива. Единственное место программы, зависящее от этого размера — это число повторений цикла. Для повышения универсальности программы это число можно задать в качестве вводимых данных. Тогда в программе оказывается массив с неизвестным заранее диапазоном изменения индекса:

алг сумма массива;

числ сумма,  $M, A[M:1]$  ; косв—  $k$  ;

Рввод  $A$  ; ввод  $M$  ;

сумма := 0 ;  $k := \text{адрес } A[0]$  ;

повторять  $M$  раз  $\text{сумма} := \text{сумма} + [-k]$  ;  $\text{кпвт}$  ;

вывод  $\text{сумма}$  ;

**кон**

При распределении регистров фиксируется только регистр, в который попадает  $A[1]$ , а второй конец массива "плавает": его место выясняется только при вводе  $M$  (разумеется,  $M$  должно быть таким, чтобы второй конец массива не занял регистры, отведенные другим переменным).

Адрес	Команды	А-язык	Комментарий
		алг $\text{сумма}$ массива ;	
	счетчик — P0		
	$k$ — P1	косв — $k$ ;	
	$M$ — P0	числ $M$ ,	$M$ совмещаем в памяти со счетчиком.
	сумма — PX	сумма,	
	$A[M:1] - P? \div PD$	$A[M:1]$ ;	Массив начинается от PD и занимает $M$ ячеек.
		Рывод $A$ ;	
00	$x \rightarrow P0$	ввод $M$ ;	
01	$\uparrow 4 \quad x \rightarrow P1$	$k := \text{адрес}$ $A[0]$ ;	
04		повторять $M$ раз	
04	0	$\text{сумма} := 0$ ;	
05	$K P \rightarrow x1 \quad +$	$\text{сумма} :=$ $\text{сумма} + [-k]$	
07	F L0 05	кпвт	
09	C/P	вывод $\text{сумма}$ ; кон	

В этой программе можно брать  $M$  не более 12.

В принципе в работе алгоритма может участвовать несколько массивов, но практически малая память ПМК позволяет разместить не более двух массивов. Если оба массива имеют переменную границу индекса, то размещать их следует на свободном участке регистров навстречу друг другу:

алг  $\text{сумма}$  покупки ;

косв —  $kK$  ; косв+  $kC$  ; числ сум,  $M$ , ЦЕНА  $[1:M]$ , КОЛ  $[M:1]$  ;

**Рввод ЦЕНА, КОЛ ; ввод  $M$ ;**

**сум:=0 ;  $\kappa K:=\text{адрес КОЛ } [0]$  ;  $\kappa Ц:=\text{адрес ЦЕНА } [0]$  ;**

**повторять  $M$  раз сум:=сум+[- $\kappa K$ ]·[+ $\kappa Ц$ ] ;  $\kappa пвт$  ;**

**вывод сум ;**

**кон**

Адрес	Команды	А-язык	Комментарий
	Счетчик – P0 $\kappa K$ – P1 $\kappa Ц$ – P4 сум – PX $M$ – P0 ЦЕНА [1:M] – P5 ÷ P? КОЛ [M:1] – P? ÷ PD	алг сумма покупки;  косв– $\kappa K$ ; косв+ $\kappa Ц$ ; числ сум, $M$ , ЦЕНА [1:M], КОЛ [M:1];	Для каждого массива своя косвенная переменная.  На участке P5 ... PD массивы направлены навстречу друг другу.
		Рввод ЦЕНА, КОЛ ;	
00	x → П 0	ввод $M$ ;	
01	1 4 x → П 1	$\kappa K:=\text{адрес КОЛ } [0]$ ;	
04	4 x → П 4	$\kappa K:=\text{адрес ЦЕНА } [0]$ ;	
06		повторять $M$ раз	
06	0	сум:=0 ;	
07	→ К П → x 1	сум + [- $\kappa K$ ].	
08	К П → x 4	[+ $\kappa Ц$ ]	
09	x +	→ сум ;	
11	└ F L0 07	$\kappa пвт$ ;	
13	С/П	вывод сум ; кон	

В этой программе максимальное значение  $M = 4$ . Ценой отказа от адресации с автоувеличением и удлинения программы это значение можно довести до 5.

Приведенные выше примеры иллюстрируют наиболее часто применяемую схему алгоритмов обработки массивов.

1. Организуется цикл повторять  $M$  раз, где  $M$  — число подлежащих обработке элементов.

2. Перед циклом переменной с описателем *косв*, *косв+* или *косв-* присваивается значение адреса первого обрабатываемого элемента или значение соседнего с ним адреса при использовании адресации с автоизменением.

3. В теле цикла используется косвенная адресация.

#### Задачи

1. Дана таблица из  $M$  чисел, где  $M$  — четное. Вычислить значение  $a_1a_2 + a_3a_4 + a_5a_6 + \dots + a_{M-1}a_M$ .

2. Пусть таблица  $a$  [1: $M$ ] содержит результаты некоторого эксперимента. Такая таблица называется выборкой. Значение  $M = \frac{1}{N} \sum_{i=1}^N a_i$

называется оценкой математического ожидания выборки, а значение

$D = \frac{1}{N-1} \sum_{i=1}^N (a_i - M)^2$  — оценкой дисперсии. Написать программу

вычисления этих оценок. Попробовать выполнить вычисление обеих оценок в одном цикле.

## 5. СОЧЕТАНИЯ ВЕТВЛЕНИЙ И ЦИКЛОВ

Каждая из рассмотренных выше программ содержала только по одному ветвлению или по одному циклу, причем условия этих ветвлений и циклов были простейшими — в виде отношения (сравнения) двух выражений. Но в более сложных задачах нам придется встретиться с ситуациями, требующими комбинированного использования ветвлений и циклов разных видов, а также более сложных условий.

Заметим, что как ветвление, так и цикл имеют *только по одному входу и выходу*, а также одну или две внутренние части (тело цикла или ветви ветвления), которые также содержат *только по одному входу и выходу* каждая. Отсюда ясно, что сочетать их можно только двумя способами:

*следованием* друг за другом, когда выход одной конструкции соединяется со входом другой (рис. 10, а, б);

*вложением* одной конструкции в другую, когда одна конструкция подставляется в качестве внутренней части другой (рис. 10, в, г).

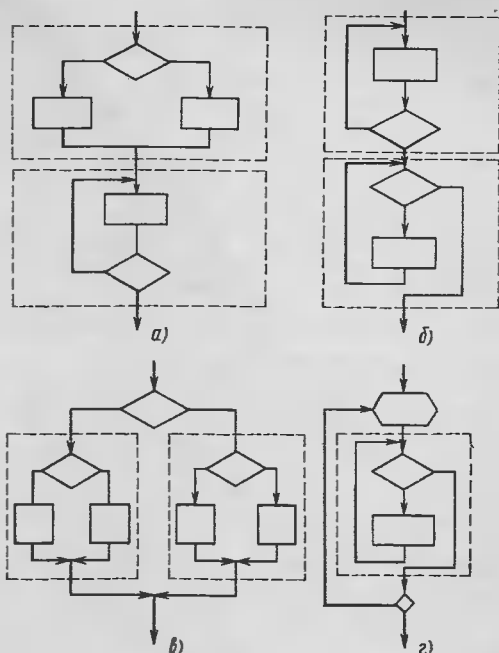


Рис. 10. Следование и вложение конструкций:

*а* — следование ветвления и цикла; *б* — следование цикла до и цикла пока; *в* — вложение ветвления в ветвление; *г* — вложение цикла пока в цикл раз

Комбинированная конструкция также имеет один вход и один выход и может выступать как одна конструкция, которую можно сочетать путем следования или вложения с любой другой конструкцией, и так до бесконечности.

Программирование сочетаний конструкций на ПМК подчиняется единственному простому правилу:

*каждая конструкция программируется по своим правилам (описанным в предыдущих параграфах) независимо от того, с чем и как она сочетается.*

Следование не создает никаких проблем при программировании: кончили программирование одной конструкции — начинаем программирование следующей. Единственный достойный упоминания нюанс заключается в том, что

*для следующих друг за другом циклов повторять ... раз можно использовать для счетчика один и тот же регистр.*










Рассмотрим программирование вложенных конструкций. Начнем с вложения ветвлений. Вычисление максимального из трех чисел выполняется алгоритмом:

если  $a \geq b$  то если  $a \geq c$  то макс:= $a$  иначе макс:= $c$  все  
иначе если  $b \geq c$  то макс:= $b$  иначе макс:= $c$  все

все

Дополнительные подчеркивания показывают соответствия между если, то, иначе и все. (Здесь можно провести некоторую оптимизацию, вынеся "за скобки" "макс:=", но мы сейчас делать этого не будем.) Програмируем этот фрагмент с адреса 20, считая, что  $a$ ,  $b$  и  $c$  размещены в RA, RB и RC соответственно, а максимальное число (макс) размещено в R0.

Адрес	Команды	А-язык	Комментарий
20	П→х А    П→х В	если $a - b$	Условие первого если программируем по правилам программирования ветвлений.
23	F X ≥ 0	0	
24		то	
25	П→х А    П→х С	если $a - c$	На ветви то первого если встречается второе если.
28	F X ≥ 0	≥ 0	
29		то	
30	П→х А    х→П 0	макс: = $a$	Временно прервав программирование первого если, выполняем программирование второго если по тем же правилам.
32	БП	иначе	
33			
34	П→х С    х→П 0	макс: = $c$	После обработки второго иначе получаются уже три пустые ячейки.
36	→...	все	
			Первое все завершает второе если, и можно заполнить пропущенные при программировании этого если ячейки.
			
36	БП	иначе	Продолжаем программирование первого если.
37			
38	П→х В    П→х С	если $b - c$	И опять приходится отложить работу с первым если и заняться третьим если, которое программируется по тем же правилам.
41	F X ≥ 0	≥ 0	
42		то	
43	П→х В    х→П 0	макс: = $b$	

1 2

34

Адрес	Команды	А-язык	Комментарий
45	БП		
46		иначе	
47	$\Pi \rightarrow x \quad x \rightarrow \Pi \quad 0$	макс: = c	
49	...	все	При обработке этого все можно заполнить ячейки, относящиеся к соответствующему если.
49	...	все	И, наконец, обработка последнего все позволяет заполнить оставшиеся пустые ячейки.

Заметим, что по команде "БП" по адресу 32–33 выполняется переход на другую команду "БП". Здесь можно сэкономить полсекунды выполнения программы, записав в этой команде сразу адрес 49. Однако, чтобы не ошибаться, можно порекомендовать сначала записывать программу по формальным правилам, как это сделано выше, а затем при обнаружении переходов на команды "БП" заменять в них адреса тем адресом, который стоит в команде "БП". В противном случае при попытке написать сразу наилучший вариант легко запутаться в перекрещивающихся переходах (см. стрелки в программе).

В целом программирование вложенных конструкций выполняется по тому же принципу, что и работа со стеком: незавершенная конструкция как бы уходит вглубь "стека" нашей памяти, и мы начинаем работать с вложенной конструкцией, а когда закончим работу с ней, из "стека" памяти "выскакивает" незавершенная конструкция.

Для иллюстрации сочетания ветвления с циклом вернемся к задаче определения корня функции на отрезке  $[a, b]$  при условии, что  $f(a)$  и  $f(b)$  имеют разные знаки. Рассмотренный ранее метод хорд накладывает на функцию довольно жесткие требования. Однако имеется и другой метод, в котором требуется только, чтобы функция была непрерывна и на отрезке  $[a, b]$  существовал ровно один корень (не требуется даже дифференцируемость). Например, для монотонной на отрезке  $[a, b]$  непрерывной функции единственность корня гарантирована. Впрочем и требование единственности можно ослабить: при наличии нескольких корней метод дает какой-то один из них, хотя заранее

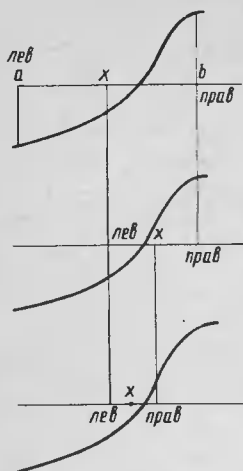


Рис. 11. К задаче решения уравнения методом деления пополам

не ясно, какой именно. Этот метод называется методом *деления пополам* (*половинного деления*, *дихотомии*, *двоичного поиска*) и заключается в следующем. Исследуется знак функции  $f(c)$  где  $c$  — это середина отрезка  $[a, b]$ . Если знаки  $f(a)$  и  $f(c)$  совпадают, то корень находится на отрезке  $[c, b]$ , в противном случае — на отрезке  $[a, c]$ . Возьмем тот из отрезков, где находится корень (рис. 11), опять берем его середину и т.д. На каждом шаге ограничивающий корень отрезок становится в 2 раза меньше. Когда его длина станет меньше заданного

отрезка  $\epsilon$ , любую точку отрезка можно считать корнем. Отрезок в алгоритме будем задавать его границами — переменными *лев* и *прав*, значения которых приближаются друг к другу. Тело алгоритма имеет вид

ввод  $a, b, \epsilon$ ;

лев :=  $a$  ; прав :=  $b$  ;

повторять  $x := (\text{лев} + \text{прав}) / 2$  ;

если  $f(x) \cdot f(\text{лев}) \geq 0$  то лев :=  $x$  иначе прав :=  $x$  все

до прав — лев  $< \epsilon$  ;

вывод  $x$  ;

Заметим, что условие после *если* проверяет одинаковость знаков у сомножителей. При этом знак  $f(\text{лев})$  всегда одинаков, хотя *лев* и меняет свое значение, поэтому вычисление  $f(\text{лев})$  можно вынести перед циклом (нужно не само значение, а только его знак) и запомнить это значение в переменной, которую назовем *флев*. Применим также все другие известные по предыдущим программам способы оптимизации: ввод  $a$  и  $b$  непосредственно в *лев* и *прав*; внесение присваивания в выражение и в итоге получим:

числ прав, лев,  $\epsilon$ , флев,  $x$  ;

Рввод лев, прав,  $\epsilon$  ;

флев :=  $f(\text{лев})$  ;

**повторять**

если  $f(x := (\text{прав} + \text{лев})/2) \cdot \text{флев} \geq 0$  то  $\text{лев} := x$   
иначе  $\text{прав} := x$  все

до  $\text{прав} - \text{лев} < \epsilon$  ;

**вывод**  $x$  ;

(Рывод взят для того, чтобы не отвлекаться на программирование знакомых конструкций.)

Этот алгоритм можно построить и с помощью цикла пока:

**пока**  $\text{прав} - \text{лев} \geq \epsilon$  **повторять**

если  $f(x := (\text{прав} + \text{лев})/2) \cdot \text{флев} \geq 0$  то  $\text{лев} := x$   
иначе  $\text{прав} := x$  все

**кпвт** ;

**вывод**  $\text{лев}$  ;

Если в этом варианте *прав* и *лев* с самого начала задают отрезок с длиной менее  $\epsilon$ , то тело цикла не выполнится ни разу и  $x$  не получит никакого значения, поэтому выводиться  $x$  нельзя и выведено *лев* (можно ведь взять любую точку отрезка).

Хотя это и не сразу очевидно, но тот же алгоритм можно построить и циклом **повторять** ... раз: чтобы отрезок длиной  $b - a$  превратился путем деления пополам в отрезок с длиной меньшей  $\epsilon$ , деление надо выполнить  $\lceil \log_2 \frac{b - a}{\epsilon} \rceil + 1$  раз.

Соответственно можно алгоритм записать в виде (записан только цикл):

**повторять**  $\ln((\text{прав} - \text{лев})/\epsilon)/\ln 2 + 1$  раз

если  $f(x := (\text{прав} + \text{лев})/2) \cdot \text{флев} \geq 0$  то  $\text{лев} := x$   
иначе  $\text{прав} := x$  все

**кпвт**

(в заголовке цикла использована формула  $\log_2 a = \ln a / \ln 2$ ). Заметим, что число повторений цикла определяется только один раз и изменение *лев* и *прав* в теле цикла на число повторений не влияет.

Как обычно, для проведения машинного эксперимента возьмем простейшую функцию  $f(x) = \sin x$  и запрограммируем все три варианта, чтобы сравнить их. Распределение регистров одинаковое; переменные занимают следующие регистры: P0 — счетчик (только в третьем варианте); P1 — *прав*; P2 — *лев*; P3 —  $\epsilon$ ; P4 — *флев*; P5 —  $x$ .

Адреса	Команды	А-язык	Комментарий
00	$\Pi \rightarrow x 2$	$F \sin$	$f$ (лев)
02	$x \rightarrow \Pi 4$	$\rightarrow \text{флев}$ ;	
03		повторять	Программируем тело цикла до.
03	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2 +$	если $f(x) =$
06	$2 \div$	$x \rightarrow \Pi 5$	(прав +
09	$C/\Pi$	$F \sin$	лев) / 2)
11	$\Pi \rightarrow x 4$	$x$	$\cdot \text{флев}$
13	$F X \geq 0$	19	$\geq 0$ то
15	$\Pi \rightarrow x 5$	$x \rightarrow \Pi 2$	лев; $=x$
17	БП	21	иначе
19	$\Pi \rightarrow x 5$	$x \rightarrow \Pi 1$	прав; $=x$
21	$\rightarrow \dots$		все
21	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2 -$	до прав -
24	$\Pi \rightarrow x 3$	--	лев $< \epsilon$ ;
26	$F X < 0$	03	
28	$\Pi \rightarrow x 5$	$C/\Pi$	вывод $x$ ;

При программировании цикла до не остаются пустые ячейки, поэтому и его сочетание с другими конструкциями программируется проще. Реализуем теперь вариант с циклом пока.

Адреса	Команды	А-язык	Комментарий
Команды по адресам 00 ... 02 не меняются.			
03	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2 -$	пока прав
06	$\Pi \rightarrow x 3$	--	--лев - $\epsilon$
08	$F X \geq 0$		$\geq 0$ повтор
10	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2 +$	если $f($
13	$2 \div$	$x \rightarrow \Pi 5$	$x := (\text{прав} +$
16	$C/\Pi$		лев) / 2)
17	$F \sin$	$\Pi \rightarrow x 4$	$\cdot \text{флев}$
20	$F X \geq 0$		$\geq 0$ то
22	$\Pi \rightarrow x 5$	$x \rightarrow \Pi 2$	лев; $=x$

Адрес	Команды	А-язык	Комментарий
24	БП		иначе
26	П → x 5    x → П 1		прав: = x
28	...		все
28	БП	03	кпвт
30	П → x 5    С/П		вывод x ;

Программирование ветвления закончено и можно заполнить относящиеся к ветвлению ячейки.

Теперь закончено программирование цикла и заполняется ячейка, относящаяся к циклу.

Для единообразия и в этом варианте сделали вывод  $x$ , так как вряд ли придется задавать такие исходные данные, чтобы цикл не работал ни разу.

Программируем третий вариант — с циклом раз.

Адреса	Команды	А-язык	Комментарий
00	Первые три команды без изменений		
03	П → x 1	П → x 2 – повторять	Программируем цикл раз: вычисляем число повторений и засылаем в счетчик.
06	П → x 3	÷ Fln ln ((прав	
09	2 Fln	÷ –лев) /ε/	
12	1 +	x → П 0 ln 2 + 1 раз	
15	Команды 15 ... 32 повторяют команды 10 ... 27 предыдущего варианта с изменением только адресов у команд переходов.		
33	FL0	15	кпвт ;
35	П → x 5	С/П	вывод x ;

Этот вариант получился длиннее, но программа будет выполняться быстрее, так как в цикле работает меньше команд.

Приступаем к машинным экспериментам. Найдём корень функции  $\sin x$  на отрезке  $[6, 7]$  (весь расчёт выполняем в радианах: поставьте переключатель "Г—ГРД—Р" в положение "Р") с точностью 0.01. Должно получиться  $2\pi \approx 6.2832$ . При исполнении любого варианта программы на промежуточном останове

увидим значения 6.5, затем 6.25, 6.375 (пока очередное приближение всякий раз оказывается с другой стороны от точного значения и ветви ветвления работают по очереди), 6.3125 (новое приближение с той же стороны, что и предыдущее), 6.28125, 6.296875 (теперь два приближения с другой стороны), 6.2890625 — точность достигнута: следующий останов уже вне цикла, на индикаторе 6.2890625. В качестве результатов можно записать только 6.28, так как последующие цифры уже неточные: корень искали с точностью 0.01.

Если теперь переведем переключатель "Г—ГРД—Р" в положение "Г" и повторим вычисления, то получим результат 6.99 ...: на отрезке  $[6^\circ, 7^\circ]$  корня нет, но алгоритм все же даст результат. Применение алгоритма к неподходящим данным все равно может дать результат, ответственность за использование которого лежит на пользователе алгоритма.

Попробуем применить алгоритм к отрезку  $[6, 14]$  (в радианах), на котором имеется три корня. Первое приближение 10, второе 12. Ясно, что на первом же шаге мы выбросили отрезок  $[6, 10]$ , на котором имеется два корня, и вышли на отрезок с единственным корнем  $4\pi \approx 12.567$ , к которому и приближаемся. Но нельзя задавать отрезок с четным числом корней, так как тогда можно потерять все корни и получить результат, не совпадающий ни с одним из корней.

Сравним теперь скоростные качества алгоритмов. Выполним, убрав из программ промежуточный останов, все варианты на данных первого примера: отрезок  $[6, 7]$ , точность 0.01. Вариант с циклом **до** работает 57 с, с циклом **пока** — 62 с, а с циклом **раз** — 53 с. И чем меньше будет взято  $\epsilon$ , т.е. чем больше раз будет выполняться тело цикла, тем яснее будет, что он работает примерно в  $23/16$  раз быстрее варианта с циклом **пока** (23 и 16 — это число команд, работающих на каждом проходе цикла в данных вариантах; заметим, что это не число команд, входящих в тело цикла, а число команд, выполняемых в нем).

Вариант с циклом **пока** можно слегка ускорить, заменив адрес перехода в ячейке 25 адресом 03. Это даст около 0.5 с выигрыша на каждом выполнении ветви **то**, т.е. около 0.25 с на каждом проходе цикла (ветвь **то** работает в среднем через раз). Но делать это следует только после полного написания программы, иначе есть риск допустить ошибку.

При программировании непосредственно в кодах ПМК или при нарушении правил перевода возможны рассуждения такого рода: так как после выполнения оператора "**лев**: =  $x$ " необходимые на данном проходе цикла действия завершены, то можно вернуться на начало цикла, следовательно, ставим команду

"БП 03". Это рассуждение не приводит к ошибке в варианте с циклом **пока**, но в варианте с циклом **раз** таким способом пропускается изменение состояния счетчика цикла, а значит, число повторений цикла будет неправильным (вплоть до заикливания).

Для иллюстрации вложения двух циклов рассмотрим такую задачу. Пусть надо получить таблицу значений некоторой функции в  $N$  точках, отстоящих друг от друга на расстояние  $h$ , а первая точка имеет координату  $a$ . Цикл вычислений должен иметь вид:

$x := a$  ;

**повторять**  $N$  **раз**

$y := \text{значение функции в точке } x$  ;

**вывод**  $y$  ;  $x := x + h$  ;

**кпвт**

Теперь в качестве функции возьмем такую, вычисление которой в свою очередь требует цикла, а именно многочлен  $y = \sum_{i=0}^M x^i c_{M-i}$ . Наиболее эффективный способ вычисления многочлена — это так называемая схема Горнера:

$$\sum_{i=0}^M x^i c_{M-i} = ( \dots (((c_0 x + c_1) x + c_2) x +$$

$$+ c_3) x + \dots + c_{M-1}) x + c_M,$$

реализация которой имеет вид

$y := c[0]$  ;  $k := \text{адрес } c[0]$  ;

**повторять**  $M$  **раз**  $y := y \cdot x + [^k]$  **кпвт**

Подставив в первый цикл детализацию вычисления значения функции, получаем полный алгоритм:

**алг** таблица значений многочлена;

**числ**  $x, h, y$  ; **косв** —  $k$  ; **числ**  $N, M, c[M:0]$  ;

**Рввод**  $h, M, c, a, x$  ;

**повторять**  $N$  **раз**

$y := c[0]$  ;  $k := \text{адрес } c[0]$  ;



повторять  $M$  раз  $y := y \cdot x + [-k]$  кпвт ;

вывод  $y$ ;  $x := x + h$  ;

кпвт;

кон

Пишем программу.

Адрес	Команды	А-язык	Комментарий
		алг таблица значений многочлена ;	
	Счетчик1 – P0		Для каждого из вложенных
	Счетчик2 – P1		циклов берем свой счетчик.
	$k$ – P2	косв – $k$ ;	Дальше отводим память для
	$x$ – P3	числ $x$ ,	косвенных переменных, а за-
	$h$ – P4	$h$	тем и для всех остальных,
	$y$ – PX	$y$ ,	причем $y$ будем хранить в PX,
	$N$ – P0	$N$	а $N$ совместим со счетчиком;
	$M$ – P5	$M$ ,	все остатки памяти в конце
$c[M:0]$	$- P? \div PD$	$c[M:0]$ ;	отведем для массива $c$
Рзвод $h, M, c, N, \{a\} x$ ;			
повторять $N$ раз			
00	→ 1 3 $x \rightarrow П 2$	$k := \text{адрес}$ $c[0]$ ;	Начинаем тело цикла.
03	$П \rightarrow x 5$	повторять $M$ раз	Число повторений внутренне-
04	$x \rightarrow П 1$		
05	$П \rightarrow x D$	$y := c[0]$ ;	го цикла засылаем в другой
06	→ $П \rightarrow x 3$ $x$	$y := y \cdot x$ $+ [-k]$ ;	Программируем тело вложен-
08	$K П \rightarrow x 2 +$		
10	← F L1 06	кпвт ;	Завершен внутренний цикл, и
12	C/П	вывод $y$ ;	в команде "F Li" используется
13	$П \rightarrow x 3$ $П \rightarrow x 4$	$x :=$ $x + h$ ;	счетчик этого цикла и адре-
15	$+$ $x \rightarrow П 3$		
17	← F L0 00	кпвт ;	Это кпвт программируется с
19	C/П	кон	использованием другого счет-
			чика и другого адреса.

Мы видели, что два слова **кпвт** программировались по-разному: менялись используемые счетчики и адреса в зависимости от того, к какому циклу относится данное **кпвт**. В общем случае разные **кпвт** могут относиться к разным типам циклов, например: внутренний цикл — типа **пока**, а внешний — типа **раз**, тогда **кпвт** внутреннего цикла программируется командой **"БП"** и заполнением пропущенного адреса, а **кпвт** внешнего цикла — командой **"F Li"**.

#### Задачи

1. Дана таблица из  $M$  чисел. Определить в ней значение максимального элемента.

2. Определить в таблице из  $M$  элементов номер максимального элемента.

3. Для оценки точности вычисления интеграла можно применить такой прием: отрезок разбивается на  $M$  частей и вычисляется интеграл при таком разбиении, затем интеграл вычисляется при разбиении отрезка на  $2M$  частей и результаты сравниваются. Написать программу, по которой вычисляется интеграл на отрезке  $[a, b]$  от функции  $f(x)$ . Вычисление начинается с деления отрезка на  $M$  частей, а затем  $M$  увеличивается в 2 раза и это повторяется до тех пор, пока разница между двумя значениями интеграла не станет меньше заданного.

## 6. СЛОЖНЫЕ УСЛОВИЯ

Перейдем к рассмотрению комбинирования условий. Иногда условие невозможно сформулировать в виде одного сравнения, а нужно сделать несколько проверок и принять решение, зависящее от результатов всех проверок. При этом итоговое решение можно получить из результатов проверок всего двумя способами: "если все проверки дали положительный результат" и "если хотя бы одна проверка дала положительный результат". Все остальные варианты сводятся к этим двум. В записи на А-языке соответствующие сравнения (проверки) соединяются союзом **и**, когда нужен вариант "если все...", либо союзом **или**, когда нужен вариант "если хотя бы одна...". В целом эти союзы вполне соответствуют своему смыслу на русском языке. Например, проверка принадлежности точки  $x$  отрезку  $[a, b]$  записывается так:

если  $a \leq x$  и  $x \leq b$  то принадлежит иначе не принадлежит все или

если  $a > x$  или  $x > b$  то не принадлежит иначе принадлежит все.

Есть, однако, один нюанс: союз **и** полностью сохраняет свое значение, имеющееся в русском языке, а союз **или** его несколько изменяет. Фраза "А или В" на русском языке означает "либо А, либо В, но не оба вместе" (так называемое *исключающее*

"или"), а в программировании обычно используют *включающие* "или": запись "А или В" означает "либо А, либо В, либо и А и В", т.е. "хотя бы одно из А и В".

Все остальные варианты легко сводятся к этим двум. Когда нужна проверка типа "если условие не выполнено...", то она равносильна проверке "если выполнено обратное условие...".  
Конструкция

если А и В и С то...

равносильна

если (А и В) и С то...

где союзом и объединены два условия: первое — (А и В) в свою очередь является комбинированным, второе — С. Так можно усложнять условие сколько угодно. Например, запись

пока (А и В) или ( $\bar{C}$  и Е) или Р повторять...

предполагает, что тело цикла будет выполнено, если удовлетворен один или более пунктов из числа следующих:

выполнены оба условия А и В;

выполнено Е и не выполнено С (черта над С означает условие, обратное к С);

выполнено Р.

На ПМК МК-61 и МК-52 есть специальные команды, реализующие операции и и или, но реализованы они так, что использовать их для организации проверки комбинированных условий невозможно (вообще сложно понять, для чего их можно было бы использовать). Однако реализация таких условий легко выполняется и с помощью уже известных команд.

1. Каждое из условий, соединенных союзом и программируется так, как будто оно единственное.

2. Все соединенные союзом или условия, кроме последнего, программируются так: используется команда "F X?0" с противоположным условием и адресом, следующим за последним адресом программной памяти, занимаемым командами проверки последнего из соединенных союзом или условий.

3. Последнее из соединенных или условий программируется так, как будто оно единственное.

Как простейший пример запрограммируем оба варианта определения принадлежности точки отрезку. В качестве признака принадлежности будем присваивать переменной *pr* значение 1, если точка принадлежит отрезку, и 0 противном случае.

пр: = если  $a \leq x$  и  $x \leq b$  то 1 иначе 0 все ;

Распределение регистров  $a - P0, b - P1, x - P2, пр - P3$ .

Адрес	Команды	А-язык	Комментарий
10	$\Pi \rightarrow x 2$	$\Pi \rightarrow x 0$	если $x - a \geq 0$ и
13	$F X \geq 0$		
15	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2$	$b - x \geq 0$ то
18	$F X \geq 0$		
20	1	1	Программируем ветвь то.
21	БП		
23	0	0	Программируем ветвь иначе.
24	...	все	
			Теперь заполняем адреса: их получилось больше чем обычно.
24	$x \rightarrow \Pi 3$	$\rightarrow пр$	

При том же распределении регистров второй вариант:

пр: = если  $a > x$  или  $x > b$  то 0 иначе 1 все ;

Адрес	Команды	А-язык	Комментарий
10	$\Pi \rightarrow x 2$	$\Pi \rightarrow x 0$	если $x - a < 0$ или
13	$F X \geq 0$		
15	$\Pi \rightarrow x 1$	$\Pi \rightarrow x 2$	$b - x < 0$ то
18	$F X < 0$		
20	...		После завершения условий можно заполнить некоторые адреса.
20	0	0	Дальше все программируется как обычно.
21	БП		
23	1	1	Заполняем адреса.
24	...	все	
24	$x \rightarrow \Pi 3$	$\rightarrow пр$	

Проверка каждого из соединенных союзом и условий "пропускает нас вниз", если условие выполнено. Если выполнены все условия, то мы "доберемся до самого низа" и будет выполняться ветвь то. Но как только очередная проверка даст ответ "нет", можно остальные условия уже не проверять и сразу переходить к выполнению ветви иначе (или выходить из цикла, если это условие в цикле пока, и т.д.). Работа соединенных союзом или условий выглядит в каком-то смысле наоборот: "пропускает вниз" невыполненное условие, а при первом же выполненном условии остальные уже можно не проверять, а сразу переходить к ветви то (к телу цикла пока, к выходу из цикла до и т.д.).

Рассмотрим теперь пример сложной комбинации условий:

пока ( $A = 0$  или  $B = 0$ ) и ( $C \geq 0$  или  $E \geq 0$ ) и  $P = 0$

повторять ...

Распределение регистров:  $A - R0, B - R1, C - R2, E - R3, P - R4$ .

Адрес	Команды	А-язык	Комментарий
10	$P \rightarrow x 0$	пока ( $A = 0$	Вместо условия, стоящего перед или, программируем обратное условие.
11	$F X \neq 0$		
13	$P \rightarrow x 1$	или $B = 0$ ) и	Последнее в данной "группе или" условие программируем как обычно.
14	$F X = 0$		
16	...		"Группа или" закончена, все пропущенные адреса, кроме последнего можно заполнить очередным адресом.
16	$P \rightarrow x 2$	( $C \geq 0$ или	Вторая "группа или" программируется так же: для не последнего условия берется обратное, последнее условие берем как таковое.
17	$F X < 0$		
19	$P \rightarrow x 3$	$E \geq 0$ ) и	
20	$F X \geq 0$		
22	...		Заполняем пропущенный адрес.
22	$P \rightarrow x 4$	$P = 0$ повтор	Условие, соединенное союзом и, программируется как обычно.
23	$F X = 0$		
25	...		Программируем тело цикла.
...			
50	БП	кпвт	А при программировании кпвт придется заполнить несколько адресов.
52	...		

Отметим, что условия проверяются до тех пор, пока какая-либо проверка не позволит однозначно выбрать последующее действие, после чего оставшиеся условия уже не проверяются.

В разных конструкциях циклов (а их мы изучили три варианта) проверки выполняются по-разному: условия стоят в разных местах. Но бывают ситуации, когда два условия выхода из цикла нельзя объединить в одно союзами и и или, так как они должны проверяться в разных местах: одно до тела цикла, а второе — после. Для удобства комбинирования условий в циклах, введем в А-язык общую конструкцию цикла, для которой рассмотренные ранее случаи являются частными. Такой цикл имеет вид:

пока условие1 повторять  $M$  раз

тело цикла

до условие2 кпвт

Этот цикл имеет три условия окончания его работы: либо тело выполнено  $M$  раз, либо не удовлетворено условие1 (проверяемое до тела цикла), либо удовлетворено условие2 (проверяемое после тела). Любая из проверок, а также любая их комбинация может отсутствовать. Если отсутствуют все проверки, то получаем бесконечный цикл, а если присутствует только одна из проверок, то получаем знакомые конструкции циклов.

Случаи, когда нужны циклы со всеми тремя проверками, весьма редки, а примером алгоритма, в котором удобно применить цикл с двумя проверками, является алгоритм определения номера элемента массива, обладающего некоторыми свойствами (положительного, нулевого, с заданным значением и т.д.), при условии, что нужного элемента может и не быть вообще (в этом случае надо выдать нулевой номер). Тело алгоритма можно записать так:

$k := \text{адрес } A [M] ;$

пока  $[k] \neq 0$  повторять  $M$  раз  $k := k - 1$  кпвт;

$k := k - \text{адрес } A [0] ;$

По этому алгоритму отыскивается первый с конца нулевой элемент в массиве из  $M$  элементов. Если такового в массиве нет, то тело цикла выполнится  $M$  раз, после чего  $k$  будет указывать на элемент, стоящий перед первым элементом массива; если же нужный элемент есть, то условие после пока прервет выполнение цикла, в переменной  $k$  при этом останется адрес нужного элемента. Стоящий после цикла оператора превращает адрес элемента в номер.

**Замечание.** Казалось бы, номер элемента массива можно получить и проще: он находится в счетчике цикла, но этому мешает небольшая ошибка конструкторов ПМК. Если нужного элемента нет и цикл выполняется все  $M$  раз, то в счетчике остается не ноль (как следовало ожидать), а единица, и не ясно: то ли нужный элемент стоит на первом месте, то ли его вообще нет.

Если в первом операторе взять "адрес  $A[0]$ ", а в теле цикла увеличивать  $k$ , то получим номер первого подходящего элемента, считая от начала массива.

Правила программирования общей конструкции цикла объединяют элементы программирования частных вариантов цикла.

1. Если есть часть " $M$  раз", то для цикла отводится счетчик, вычисляется значение  $M$  и заносится в счетчик (хотя эта часть и располагается после части "пока условие", но программируется она первой).
2. Если есть часть "пока условие", то она программируется так же, как и в простом цикле пока.
3. Программируется тело цикла.
4. Если есть часть до и нет части " $M$  раз", то часть до программируется так же, как в простом цикле до, но при наличии части пока условный переход делается на начало проверки стоящего после пока условия, а не на начало тела цикла.
5. Если есть и часть до и часть раз, то программируется условие, обратное указанному после до, и в команде условного перехода оставляется пустой адрес.
6. Слово **кпвт** (которое после части до может отсутствовать, но все равно подразумевается присутствующим) программируется командой "**F Li**", если есть часть раз, и командой "**БП**", если части раз нет. В обоих случаях в командах ставится адрес начала проверки условия, стоящего после пока, если оно есть, или начала тела цикла в противном случае. Если имеется часть до, а части раз нет, то **кпвт** не программируется.
7. Пропущенные на шагах 2 и 5 адреса (если они есть) заполняются очередным свободным адресом.

Программируем алгоритм поиска номера нулевого элемента.

Адрес	Команды	А-язык	Комментарий
алг поиск нулевого;			
	счетчик — P0		
	$k$ — PD	косв $k$ ;	
	$M$ — P1	числ $M$ ,	
	$A [1:M]$ — P?—PC	$A [1:M]$ ;	
Рзвод $A, M$ ;			
00	1 2 $x \rightarrow \text{PD}$	$k := \text{адрес}$ $A [M]$ ;	
03	$\text{П} \rightarrow x 1$ $x \rightarrow \text{П} 0$	цикл $M$ раз	Сначала программируется часть "М раз", а затем часть "пока условие".
05	$\text{К П} \rightarrow x D$	пока $[k]$	Адрес заполнится только при обработке кпвт.
06	$\text{F X} \neq 0 14$	$\neq 0$ повторять	
08	$\text{П} \rightarrow x D 1$	$k - 1$	
10	— $x \rightarrow \text{П} D$	$\rightarrow k$	
12	$\text{F L} 0 05$	кпвт	Есть части раз и пока, поэтому ставится команда "F Li" и адрес начала условия после пока.
14	$\text{П} \rightarrow x D$	$k := k -$	
15	1 2 $\text{П} \rightarrow x 1$	адрес $A [0]$	Адрес $A [0]$ приходится вычислять как "адрес $A [M] - M$ "
18	— — $x \rightarrow \text{П} D$		
...	...		

### Задачи

1. В точке с координатами  $(x_0, y_0)$  находится локатор, который видит объекты, находящиеся на высоте не менее  $H$ , и расстояние до которых, измеренное вдоль луча локатора (так называемая наклонная дальность), не превышает  $D$ . Определить, виден ли на локаторе объект с координатами  $(X, Y)$ , находящийся на высоте  $h$ . Программа должна дать результаты: 0, если объект не виден; 1, если виден.

2. Дана функция  $f(x)$ . Двигаясь от точки  $a$ , где  $a < 0$ , шагом  $H$  ( $H > 0$ ) определить с точностью  $H$  координаты точки, в которой график функции  $f(x)$  пересекает одну из осей координат, а именно ту, которую он пересечет раньше.



## 7. НЕСТАНДАРТНЫЕ УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

Можно с математической строгостью доказать, что рассмотренные выше конструкции ветвлений и циклов достаточны для построения любого алгоритма. Более того, достаточно иметь только одну конструкцию цикла: либо цикл *до*, либо цикл *пока* (но не цикл *раз*). Например; если нам доступен только цикл *пока*, а алгоритм требует цикла *до*, то вместо

*повторять действия до условие;*

можно записать

*действия; пока обратное условие повторять действия* *кпвт* т.е. еще раз запрограммировать тело цикла перед циклом *пока*. Понятно, что такая возможность замены одного вида цикла другим представляет лишь теоретический интерес, так как двойное программирование тела цикла (или другие искусственные приемы, которыми иногда можно заменить один вид цикла другим) накладны. При программировании на ПМК, где и программная память и память для данных весьма ограничены, часто возникает задача добиться предельной эффективности программы по памяти (т.е. запрограммировать ее наиболее коротким способом, иначе она не помещается в памяти) или по времени (иначе она работает недопустимо долго). И здесь часто обнаруживается, что нужной эффективности можно добиться только путем изобретения какого-то специфичного приема, учитывающего особенности конкретной задачи. В частности, иногда можно получить существенный эффект, если использовать управляющую конструкцию, отличную от изученных ветвлений и циклов.

В качестве простейшего примера вернемся к задаче определения максимального из трех чисел, что делалось по алгоритму

если  $a \geq b$  то если  $a \geq c$  то макс: =  $a$  иначе макс: =  $c$  все

иначе если  $b \geq c$  то макс: =  $b$  иначе макс: =  $c$  все

все

Здесь дважды встречается фрагмент "*макс:=с*", который при программировании требует двух команд. Однако представим себе, что максимальный элемент требует более сложной обработки и обработка эта различается для случая, когда максимален  $a$ , когда максимален  $b$  и когда максимален  $c$ . Тогда дважды запрограммировать обработку  $c$  становится неприятным. Можно выйти из положения так:

если  $a \geq b$  и  $a \geq c$

то  $\text{макс} := a$

иначе если  $a \geq b$  и  $c \geq b$  то  $\text{макс} := c$  иначе  $\text{макс} := b$  все

но здесь придется программировать дважды сравнение  $a$  с  $b$ . Фактически нам нужно получить управляющую конструкцию, показанную на рис. 12. В целом она ничем не хуже известных конструкций ветвлений и циклов: так же имеет один вход и один выход, но содержит три проверки и три блока обработки. Ясно, что такая конструкция требуется не часто и вводить специальную запись в А-язык для нее нецелесообразно. Проще ввести средства, позволяющие построить при необходимости такую, равно как и другую, конструкцию.

Мы легко получили бы эту конструкцию из вложенных ветвлений, если бы смогли явно указать, какую группу операторов надо выполнить (во всех рассмотренных до сих пор конструкциях нужные операторы выбирались неявно: по их расположению после *кпвт* или между *то* и *иначе* и т.д.). Для такого указания нам нужны, во-первых, средства именования операторов и, во-вторых, средства явного указания, какие операторы должны выполняться далее.

Имя оператора называется в программировании *меткой*. Выглядит оно так же, как и имя переменной, размещается перед оператором и отделяется от него двоеточием. Например:

М4:макс:=с;

Здесь М4 — это имя (метка) оператора присваивания. Метка может стоять перед любым оператором, кроме описаний переменных. Понятно, что все метки в программе должны быть различны.

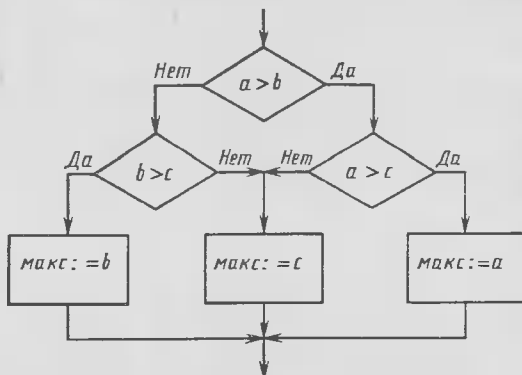


Рис. 12. Нестандартная управляющая конструкция для определения максимального из трех значений

Средство явного указания выполняемого далее оператора называется *оператором перехода* и имеет вид

**перейти к метка;**

После выполнения этого оператора выполняется оператор с указанной меткой, а далее — тот оператор, который должен быть выполнен за оператором с указанной меткой в соответствии со стоящими за ним служебными словами. С применением таких средств наша задача решается следующим образом:

если  $a \geq b$  то если  $a \geq c$  то макс:= $a$  иначе перейти к МС все  
иначе если  $b \geq c$  то макс:= $b$  иначе МС: макс:= $c$  все  
все

или же второй вариант:

если  $a \geq b$  то если  $a > c$  то макс:= $a$  иначе МС: макс:= $c$  все  
иначе если  $b \geq c$  то макс:= $b$  иначе перейти к МС все  
все

Отметим, что во втором варианте после перехода к оператору с меткой МС этот оператор будет выполнен, а далее будут выполняться операторы, размещенные следом. Но следом стоит слово **все**, которое никаких действий не требует, а далее — слово **иначе**, которое требует перехода на завершающее **все**, и этот переход будет осуществлен, т.е. повторно вторая строка алгоритма выполняться не будет.

Правило программирования оператора **перейти к** простое.

1. Оператор **перейти к** программируется командой "БП" с адресом начала оператора, помеченного упомянутой меткой.

2. Адрес команды "БП" может заполняться позже, если упомянутая метка расположена ниже по тексту программы.

Программируем первый вариант алгоритма (распределение регистров прежние).

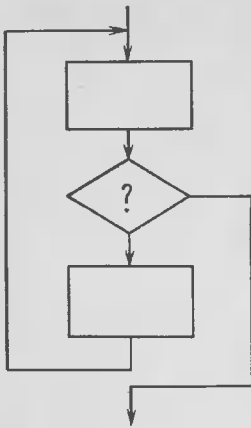
Адрес	Команды	А-язык	Комментарий
10	П → x A П → x B —	если $a - b$	
13	— F X ≥ 0 28	≥ 0 то	
15	П → x A П → x C —	если $a - c$	
18	— F X ≥ 0 24	≥ 0 то	
20	П → x A x → П 0	макс:= $a$	

Адрес	Команды	А-язык	Комментарий
22	БП		
24	БП	иначе перейти к МС все	Адрес заполнится, когда дойдем до оператора с меткой МС.
26	БП	иначе	
28	П → x В	если $v - c$	
31	F X ≥ 0	≥ 0 то	
33	П → x В x → П 0	макс := v	
35	БП	иначе	
37	...	МС:	Теперь заполняем адрес у команды "БП".
37	П → x С x → П 0	макс := c	
39	...	все все	

Пока выигрыша в объеме памяти не видно, так как замененный оператор перехода фрагмент был очень мал. Но и здесь появляются большие возможности улучшения программы. Так как тело ветви **иначе** теперь состоит из одной команды "БП", то переход на него с адреса 19 можно заменить переходом сразу на адрес 37. А так как после выполнения этой замены команда "БП" по адресу 24 становится ненужной, ее можно удалить; после чего сразу видно, что можно удалить и команду "БП" по адресу 22, так как она выполняет переход к следующей команде. В итоге программа сокращается на четыре ячейки из 38, т.е. более чем на 10 %.

Адрес	Команды	А-язык	Комментарий
10	П → x А П → x В	если $a - v$	
13	F X ≥ 0	≥ 0 то	
15	П → x А П → x С	если $a - c$	
18	F X ≥ 33	≥ 0 то	Сразу переходим в нужное место.
20	П → x А x → П 0	макс := a	
22	БП	иначе	Здесь запрограммировано оба <b>иначе</b> сразу. Фактически запрограммировали раньше. Команд не требуется.
		перейти к МС	
		все	
24	П → x В П → x С	иначе	
27	F X ≥ 0 33	если $v - c$	
29	П → x В x → П 0	≥ 0 то	
31	БП 35	макс := v	
33	П → x С x → П 0	иначе	
35	...	МС; макс := c	
		все все	

Рис. 13. Цикл с выходом из середины



Появление возможности подобной оптимизации является обычным при использовании оператора перехода.

Еще один пример полезности нестандартных конструкций дает следующая задача. В цикле вычисляются значения некоторой переменной  $x$  и обрабатываются, вычисление  $x$  нужно продолжать до тех пор, пока не встретится  $x$ , совпадающее с одним из элементов массива  $a$ , это значение обрабатывать уже не нужно. (Допустим, что обработка заключается в занесении нового

значения в массив, т. е. в массиве накапливаются значения до тех пор, пока они не начнут повторяться.) Здесь не удастся применить ни цикл пока (так как прежде, чем выполнять проверку, нужно вычислить  $x$  и проверку нельзя ставить в начало), ни цикл до (так как после проверки нужна обработка и проверку нельзя ставить в конец). Нужен новый вариант цикла — цикл с выходом из середины (рис. 13). Подобная ситуация встречается сравнительно часто, и в некоторых языках программирования имеется специальный оператор **выход**, выполняющий выход из цикла на оператор, следующий за **кпвт**. С применением бесконечного цикла и оператора **выход** задача решается так

**повторять**

*вычисление значения  $x$ ;*

*если  $x$  входит в массив  $a$  то выход все;*

*обработка  $x$ ;*

**кпвт**

Но и с применением оператора перехода можно получить тот же эффект:

**повторять**

*вычисление значения  $x$ ;*

*если  $x$  входит в массив  $a$  то перейти к Вых все;*

*обработка  $x$ ;*

**кпвт ;**

**Вых: ...**

Однако это еще не все: записанная в операторе если проверка не укладывается в одно выражение, а требует цикла. Здесь можно применить комбинированный цикл поиска номера элемента: если номер нулевой, то элемента, равного  $x$ , нет:

повторять

*вычисление значения  $x$ ;*

$k := \text{адрес } a [M];$

пока  $[k] \neq x$  повторять  $M$  раз  $k := k - 1$  кпвт;

если  $k - \text{адрес } a [0] > 0$  то перейти к ВЫХ все;

*обработка  $x$ ;*

кпвт;

ВЫХ: ...

Но это не лучший вариант, поскольку после того как завершен внутренний цикл, выполняется проверка, по какой причине он завершен: то ли нашли нужный элемент, то ли проверили все  $M$  элементов. Но эта проверка лишняя: та же проверка " $[k] \neq x$ ", которая выводит нас из внутреннего цикла, может вывести не за первое, а за второе кпвт, т.е. можно сделать выход сразу из двух циклов (рис. 14, а) .

повторять

*вычисление значения  $x$ ;*

$k := \text{адрес } a [M + 1];$

повторять  $M$  раз

если  $[k] = x$  то перейти к ВЫХ все;

кпвт;

*обработка  $x$ ;*

кпвт;

ВЫХ: ...

Здесь удалось не только убрать лишнюю проверку, но и применить адресацию с автоуменьшением, так как после выхода из внутреннего цикла значение  $k$  нам уже не нужно. По сравнению с предыдущим вариантом в программе для ПМК изменится адрес выхода из цикла после проверки " $[k] \neq x$ ", а в результате будет сэкономлено около десятка команд с соответствующим выигрышем в скорости работы. Показанная на рис. 14, а конст-

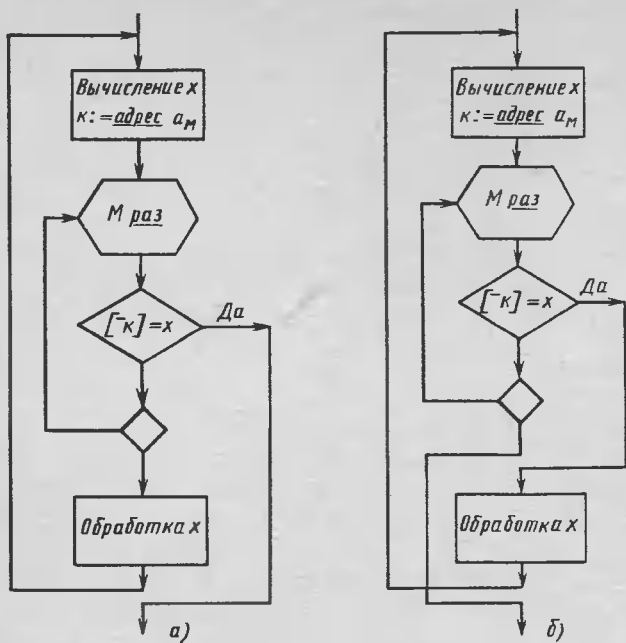


Рис. 14. Нестандартные выходы из циклов

рукция также имеет один вход и один выход и путем следования и вложения может сочетаться с любыми другими схемами.

Еще один пример полезной нестандартной конструкции дает небольшое изменение условий задачи: обрабатываются значения, входящие в массив, а при первом же значении, не входящем в массив, обработка прекращается (это напоминает алгоритм работы кассира: деньги выдаются тем, кто записан в ведомости, а при появлении незаписанного человека кассир отправляется выяснять причины). Схема конструкции показана на рис. 14, б, а на А-языке ее можно записать следующим образом:

повторять

вычисление значения  $x$ ;

$k := \text{адрес } a [M + 1]$ ;

повторять  $M$  раз

если  $[k] = x$  то перейти к ОБРАБ все;

кпвт;

перейти к Вых;

ОБРАБ: обработка х;

кшвт;

Вых: ...

Программирование этих фрагментов на ПМК и машинные эксперименты с ними рекомендуем выполнить самим читателям. Предлагаем также попробовать написать алгоритмы решения приведенных в этом параграфе задач, пользуясь только стандартными конструкциями, т.е. "чистыми" ветвлениями и циклами без операторов перехода, и сравнить полученные программы по скорости и занимаемой памяти.

Понятно, что всегда можно написать программу, пользуясь только нестандартными конструкциями. Есть и такой стиль программирования: в беспорядке записываются обрабатывающие блоки, которые затем объединяются в единое целое с помощью операторов перехода — в итоге вся программа превращается в одну большую нестандартную конструкцию. Но практика показывает, что такой стиль способствует появлению ошибок в программах. Дать четкие рекомендации по использованию в программах нестандартных конструкций очень сложно (если вообще возможно), необходимо помнить только общий принцип — нестандартные конструкции напоминают стрихнин: в малых дозах — это лекарство, а в чуть больших — уже яд.

#### Задача

Переменные  $T_1$  и  $T_2$  содержат значения времен наступления двух событий. Нулевое значение переменной означает, что соответствующее событие не наступило. Получить время более раннего события (или единственного, если наступило одно) при условии, что наличие хотя бы одного наступившего события гарантировано.

### 8. ПОДПРОГРАММЫ

В повседневной практике мы постоянно встречаемся с ситуациями, когда некоторой последовательности действий, которые нужно выполнять неоднократно, придается какое-то название, а затем для выполнения всей последовательности необходимо упомянуть только ее название. Типичный пример — кулинарные рецепты. Например, рецепт супа содержит довольно длинную последовательность действий: "взять столько-то мяса, картофеля, моркови..., мясо варить ... минут, добавить... и т.д.". После того как это описание сделано, для его исполнения достаточно дать одну короткую команду: "Вари суп!"

Аналогичная потребность встречается и в программирова-



нии. Например, в алгоритмах решения уравнений методом хорд и методом деления пополам нам пришлось дважды выполнять алгоритм вычисления функции. В качестве примера были взяты простые функции, но на практике приходится иметь дело с функциями, вычисление которых требует значительного числа команд.

Итак, нам нужно средство, которое позволило бы дать имя целой последовательности операторов (так же, как мы давали имя-метку отдельному оператору), и средство, позволяющее кратко записать "выполнить последовательность оператором с таким-то именем".

Специальным образом оформленная последовательность операторов, снабженная именем, называется *вспомогательным алгоритмом* или *подалгоритмом* или *подпрограммой* (мы не будем различать эти термины), а специальный оператор, требующий выполнения этой последовательности операторов, называется оператором *вызова подпрограммы*, или *обращением к подпрограмме*.

Подпрограммы разделяют на два класса: подпрограммы-*процедуры* (или просто *процедуры*) и подпрограммы-*функции* (или просто *функции*). Начнем с подпрограмм-процедур.

Описание процедуры имеет вид (простейший вариант):

**проц** *имя*;

*описание переменных*

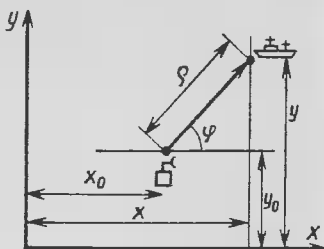
*операторы*

**возврат**

(у нас появились два новых служебных слова **проц** и **возврат**, а все остальное знакомо). Операторы как раз и составляют ту последовательность, которая оформляется в виде подпрограммы. Они могут использовать как переменные, которые описаны в алгоритме за его заголовком, так и переменные, описанные после заголовка процедуры, т.е. после оператора "**проц** *имя*". Эти операторы называются телом процедуры. Описания переменных внутри процедуры может и не быть, если в них нет необходимости.

Чтобы выполнить операторы тела процедуры, нужно в алгоритме (или как еще говорят в *головной программе*) написать оператор, состоящий из одного имени процедуры. При выполнении такого оператора фактически выполняется тело подпрограммы, а когда выполнение тела доходит до оператора **возврат**, то происходит переход назад в головную программу к оператору

Рис. 15. Полярные и декартовы координаты



пу, следующему за оператором обращения к подпрограмме.

Рассмотрим пример. Радиолокационные навигационные станции обычно получают данные с локаторов в полярной системе координат, т.е. в виде дальности  $\rho$  до объекта и его азимута  $\varphi$  (рис. 15). В той же системе координат удобно проводить и некоторые расчеты. Однако штурману корабля или самолета его координаты нужны не в виде расстояния до точки расположения станции, а в виде декартовых координат, т.е. долготы и широты  $x$  и  $y$  (кривизной поверхности Земли пренебрегаем, что допустимо при небольших расстояниях). Поэтому при выдаче штурману координат их следует преобразовать из полярных в декартовы:

$$x = x_0 + \rho \cdot \cos \varphi, \quad y = y_0 + \rho \cdot \sin \varphi,$$

где  $x_0$  и  $y_0$  — координаты радиолокационной станции. Пусть нужно выдать координаты некоторой точки, для которой  $\rho$  и  $\varphi$  задано или вычислено, затем как-то пересчитать  $\rho$  и  $\varphi$  и выдать координаты новой точки. Алгоритм имеет вид

алг ...

...

• вычисление  $\rho$  и  $\varphi$ ;

преобразование в декартовы координаты и вывод;

вычисление новых значений  $\rho$  и  $\varphi$ ;

преобразование в декартовы координаты и вывод;

...

кон

В этом алгоритме дважды приходится делать преобразование координат и вывод, что занимает более десятка команд. Поэтому оформим эти действия как подпрограмму. Для конкретности вычисление  $\rho$  и  $\varphi$  организуем просто как ввод их

значений, а вычислять новые координаты будем путем определения координат точки, получающейся из исходной поворотом на угол  $\alpha$  относительно точки  $(x_0, y_0)$ . Получаем алгоритм

алг координаты;

числ  $\rho, \varphi, x_0, y_0, \alpha$ ;

Рввод  $\rho, \varphi, x_0, y_0$ ; ввод  $\alpha$ ;

преобразование;

$\varphi := \varphi + \alpha$ ;

преобразование;

кон

проц преобразование;

вывод  $x_0 + \rho \cdot \cos \varphi, y_0 + \rho \cdot \sin \varphi$ ;

возврат

Алгоритм состоит из собственно алгоритма (*головного алгоритма, головной программы*) и подпрограммы с именем *преобразование*, которая размещается вслед за головным алгоритмом. В этой подпрограмме нет своих описанных переменных, она использует только переменные головной программы.

При первом обращении к подпрограмме выполняется единственный оператор подпрограммы — **вывод**, а затем выполнение оператора **возврат** вызывает переход "под оператор обращения", т.е. на оператор изменения значения  $\varphi$ . Следующее решение опять влечет за собой переход к выполнению тела процедуры, но на этот раз оператор **возврат** выполнит переход к **кон**.

Для реализации подпрограмм на микрокалькуляторе необходимы специальные команды, так как имеющиеся средства из числа изученных до сих пор нужного результата не дадут: если перейти к выполнению тела подпрограммы можно командой "БП", то микрокалькулятор не сможет определить, куда нужно перейти после выполнения тела подпрограммы. Поэтому на ПМК имеются две команды, специально предназначенные для организации подпрограмм — это команды "ПП" и "В/О", по которым в автоматическом режиме работа осуществляется не так, как в режиме вычислений.

Команда "ПП" (здесь расшифруем это сокращение как "переход к подпрограмме") занимает две ячейки программной памяти и так же, как по команде "БП", по ней выполняется переход к адресу, запомненному в следующей за кодом команды ячейке, но, кроме этого, в специальном месте — в *регистре*

возврата (содержимое которого мы никаким способом увидеть не можем) — запоминается текущее состояние счетчика адреса. Поскольку после прочтения адреса счетчик адреса увеличивает свое значение на единицу, то он указывает как раз на команду, следующую за "ПП", а значит, в регистре возврата запоминается место, куда надо вернуться. Команда "В/О" (расшифруем это как "Возврат к основному алгоритму") выполняет переход по тому адресу, который запомнен в регистре возврата.

С помощью этих команд и организуются подпрограммы и обращения к ним: "ПП" реализует обращение, "В/О" — возврат. Программируем алгоритм координаты.

Адрес	Команды	А-язык	Комментарий
	$\rho - P0$ $\varphi - P1$ $x_0 - P2$ $y_0 - P3$ $\alpha - P4$	алг координаты; числ $\rho$ , $\varphi$ , $x_0$ , $y_0$ , $\alpha$ ;	
Рввод $\rho, \varphi, x_0, y_0$ ;			
00	$x \rightarrow P4$	ввод $\alpha$ ;	
01 ПП		преобразование;	По команде "ПП" выполняется обращение к подпрограмме (адрес 30 взят произвольно).
02 30			
03 $P \rightarrow x1$	$P \rightarrow x4$	$\varphi := \varphi + \alpha$	
05 +	$x \rightarrow P1$		
07 ПП		преобразование;	Еще одно обращение к той же подпрограмме.
08 30			
09 ...			
проц преобразование;			
30 $P \rightarrow x2$	$P \rightarrow x0$	вывод	Программируем тело подпрограммы.
32 $P \rightarrow x1$	$F \cos$	$x_0 + \rho \cdot$	
34 $x$	$+ C/P$	$\cos \varphi$ ,	
37 $P \rightarrow 3$	$P \rightarrow x0$	$y_0 + \rho \cdot$	
39 $P \rightarrow x1$	$F \sin$	$\sin \varphi$	
41 $x$	$+ C/P$		
44 В/О		возврат	Оператор возврат программируется В/О"

Заносим программу в память ПМК. Напомним, что так как в ячейке 09...29 ничего заносить не требуется, то их можно пропустить, нажав либо клавишу "ШГп" 21 раз, либо последовательно клавиши "F АВТ", "БП 30", "F ПРГ".

Посмотрим, как работают команды "ПП" и "В/О". Занесем в регистры значения  $x_0 = 3$ ,  $y_0 = 2$ ,  $\rho = 6$ ,  $\varphi = 60^\circ$  (переключатель "Г-ГРД-Р" — в положении "Г"), набираем значение  $\alpha = 30^\circ$  и начинаем исполнение программы.

Клавиши	Индикатор	Комментарий
ПП	30.	Выполнили первую команду.
F ПРГ	44            01	ПМК готов к выполнению команды "ПП".
F АВТ	30.	
ПП	30.	Выполнили очередную команду из памяти — это команда "ПП"; произошел переход на адрес 30 ("???" — это некоторые коды команд, оставшиеся в ячейках, куда ничего не заносилось).
F ПРГ	?? ?? ?? 30	
F АВТ	30.	
С/П	6.	Выполняем тело подпрограммы и читаем выводимые результаты.
С/П	7.16	
F ПРГ	50 10 12 44	Остановились перед выполнением команды "В/О".
F АВТ	7.16	
ПП	7.16	Выполним ее,
F ПРГ	30 53 44 03	и убедимся, что произошел переход на команду, следующую за "ПП".
F АВТ	7.16	
ПП ПП	30.	Выполним команды до следующей "ПП".
ПП ПП	90.	
F ПРГ	41 10 64 07	Убедимся, что остановились перед выполнением команды "ПП",
F АВТ	90.	выполним ее;
ПП		
F ПРГ	?? ?? ?? 30	опять произошел переход на адрес 30.
F АВТ	90.	
С/П	3.	Выполняем подпрограмму.
С/П	8.	
F ПРГ	50 10 12 44	Опять остановились перед командой "В/О",
F АВТ	8.	
ПП		но на этот раз выполнение "В/О"
F ПРГ	30 53 41 09	вызовет переход уже на адрес 09.

Выделив из основного алгоритма подпрограмму, к которой обращаются более чем из одного места, мы сэкономили около 10 ячеек программной памяти. Это очень важный, но далеко не единственный момент, который делает полезным понятие подпрограммы.

Наша подпрограмма выполняет некоторые заранее предписанные действия под некими *заранее определенными переменными*. Это случай достаточно редкий, чаще приходится выполнять *одни и те же действия, но над разными переменными*. Например, в алгоритмах решения уравнения методом хорд и половинного деления значение функции приходилось вычислять дважды: аргументом были разные переменные, а результат один раз запоминался в переменной, другой раз непосредственно использовался как операнд в выражении.

Подобная ситуация в целом не является новой. Когда, например, записываем алгоритм решения квадратного уравнения  $a \cdot x^2 + b \cdot x + c = 0$  в виде формул для вычисления  $x_1$  и  $x_2$ , а затем требуем решить уравнение  $5x^2 + 8x - 12 = 0$  или  $c \cdot x^2 - e = 0$ , то это никого не смущает. Алгоритм описан применительно к неким абстрактным числам  $a$ ,  $b$ ,  $c$ , и в первом случае надо взять 5 вместо  $a$ , 8 вместо  $b$  и  $-12$  вместо  $c$ , а во втором случае  $-c$  вместо  $a$ , 0 вместо  $b$  и  $e$  вместо  $c$  (обратите внимание, что  $c$  берется в качестве старшего коэффициента). Итак, алгоритм описывается применительно к *абстрактным* числам, а при команде его выполнить еще дополнительно задаются *конкретные* значения, которые следует взять для обработки.

Те абстрактные объекты, с которыми работает подпрограмма, называют *формальными параметрами*, а реальные объекты (константы или переменные), с которыми этой подпрограмме предстоит работать при конкретном обращении к ней называют *фактическими параметрами*. Параметры можно подразделить на две категории: параметры, задающие *исходные данные*, с которыми алгоритм должен работать (они называются *входными* параметрами, или *аргументами*), и параметры, показывающие, куда следует поместить *результаты* работы алгоритма (они называются *выходными* параметрами, или *результатами*). Описание программы с параметрами имеет вид

**проц имя (арг список аргументов, рез список результатов);**

*описания переменных*

*операторы*

**возврат**

где *список аргументов* и *список результатов* представляют собой разделенные запятыми имена переменных вместе с их описателями. Например, подпрограмма решения квадратного уравнения при условии заведомой неотрицательности дискриминанта может иметь вид

проц квадратур (арг числ  $a, b, c$ , рез числ  $x_1, x_2$ );

числ  $kd, a2$ ;

$a2 := 2 \cdot a$ ;  $kd := \sqrt{b^2 - 4 \cdot a \cdot c}$ ;

$x_1 := (-b + kd) / a2$ ;  $x_2 := (-b - kd) / a2$ ;

**возврат**

В списках аргументов и результатов описатели числ можно опустить. В этой процедуре обрабатываются некие три числа называемые здесь  $a, b, c$ , результаты помещаются в некие числовые переменные, называемые  $x_1$  и  $x_2$ . В процессе работы используются промежуточные переменные  $kd$  и  $a2$ . Эти переменные описаны внутри процедуры и более о них "никто не знает". Такие переменные называются *локальными*. В принципе в процедуре могут еще использоваться и переменные головной программы, так называемые *глобальные* переменные, но в этой процедуре они не используются.

**Замечание.** Доступ подпрограммы к переменной головной программы не характерен для языков программирования, однако во вводимом здесь А-языке он необходим, чтобы точнее отразить возможности ПМК и приемы программирования на нем.

Обращение к подпрограмме с параметрами имеет вид

*имя (список фактических параметров)*;

где список фактических параметров представляет собой разделенные запятыми имена переменных, числа и выражения. Количество и типы формальных и фактических параметров должны совпадать. Формальному параметру-аргументу может соответствовать фактический параметр-выражение (в частности, выражение, состоящее только из константы или только из переменной), а параметру-результату должен соответствовать фактический параметр-переменная. Соответствие формальных и фактических параметров определяется по порядку их следования в списках заголовка процедуры и обращения к ней: первый фактический параметр соответствует первому формальному, второй фактический параметр — второму формальному и т.д.

Таким образом, возможны, например, следующие обращения к процедуре *квадатур*:

квадрурав (5,8, - 1 2, x1, x2);  
 квадрурав (с, 0, -е, кор1, кор2);  
 квадрурав (кор1, 0, 5.5, k1, k);

По первым двум обращениям выполняется решение упомянутых выше уравнений, их корни размещаются в переменных с именами x1, x2 и кор1, кор2. По третьему обращению решается уравнение  $k \cdot x^2 + 5.5 = 0$ , где  $k$  — один из корней второго уравнения, полученный при втором обращении к процедуре.

Обратите внимание, что соответствие формальных и фактических параметров идет *только по порядку их следования*; *совпадение имен никакой роли не играет*. Можно сказать, что головная программа и процедура "разговаривают на разных языках", а списки формальных и фактических параметров — это как словарь перевода с одного языка на другой. В наиболее полной мере эту независимость имен иллюстрирует следующая задача: по заданным длинам  $a$ ,  $b$  и  $c$  сторон треугольника определить косинусы всех углов и сами углы. Косинус угла, лежащего против стороны  $a$ , определяется в соответствии с теоремой косинусов по формуле

$$\cos \alpha = \frac{b^2 + c^2 - a^2}{2 \cdot b \cdot c}.$$

Подпрограмма вычисления одного угла и его косинуса имеет вид:

**проц** угол (арг  $a$ ,  $b$ ,  $c$ , рез уг, косинус);

    косинус:=( $b^2 + c^2 - a^2$ )/( $2 \cdot b \cdot c$ ); уг:=arccos (косинус);

**возврат**

а головной алгоритм имеет вид

**алг** углы и косинусы;

    числ  $a$ ,  $b$ ,  $c$ , косА, угА, косВ, угВ, коС, угС;

**ввод**  $a$ ,  $b$ ,  $c$ ;

    угол ( $a$ ,  $b$ ,  $c$ , угА, косА); угол ( $b$ ,  $c$ ,  $a$ , угВ, косВ);

    угол ( $c$ ,  $a$ ,  $b$ , угС, косС);

**вывод** угА, угВ, угС, косА, косВ, косС;

**кон**

Обратите внимание, что здесь происходит тоекратное обращение к одной и той же процедуре с одними и теми же данными, но



эти данные поступают в *разном порядке*, а потому получаются разные результаты: в процедуре вычисляется угол, лежащий против стороны, заданной первой, независимо от имени задающей эту сторону переменной.

Сформулируем правила программирования с параметрами.

1. Под описанные в процедуре переменные, а также под переменные-параметры отводятся регистры памяти, отличные от тех, которые распределены в головной программе (даже если имена некоторых переменных головной программы и подпрограммы совпадают).

2. При программировании обращения к подпрограмме вычисляются значения аргументов и засылаются в стек в порядке их перечисления в списке параметров.

3. Ставится команда "ПП" с адресом начала подпрограммы, который пока остается незаполненным.

4. Значения результатов извлекаются из стека в порядке их перечисления в списке параметров и рассылаются в переменные—фактические параметры.

5. После окончания программирования головной программы программирование подпрограммы начинается с очередного свободного адреса, которым заполняются пропущенные адреса команд "ПП".

6. Подпрограмма начинает работы с извлечения значений аргументов из стека и размещения их в отведенных соответствующим переменным регистрах.

7. Программируется тело подпрограммы.

8. Результаты засылаются в стек в порядке, обратном тому, в котором они перечислены в списке параметров (аргументы извлекаются из стека в обратном порядке).

9. Оператор **возврат** программируется командой "В/О".

### Запрограммируем алгоритм углы и косинусы

Адрес	Команды	А-язык	Комментарий
Переменные		занимают P0 ... P8 по порядку их перечисления	
00...04...		ввод $a, b, c$ ;	
05	$P \rightarrow x\ 0$	угол $(a,$	Значение первого аргумента — в стек, в PZ; значение $b$ — в PY; значение $c$ — в PX; обращаемся к подпрограмме, при возврате достаем уг4 из PX,
06	$P \rightarrow x\ 1$	$b,$	
07	$P \rightarrow x\ 2$	$c,$	
08	ПП 50		
10	$x \rightarrow P\ 4$	угA,	

Адрес	Команды	А-язык	Комментарий
11	F O		
12	x → П 3	косА);	косА — из РУ
13	П → x 1	угол (b,	Теперь засылаем в PZ — b,
14	П → x 2	c,	в РУ — c,
15	П → x 0	a,	в РХ — a;
16	ПП 50		и снова обращаемся к той же
18	x → П 6	угВ,	подпрограмме, но результаты
			теперь засылаем в угВ
19	F O		
20	x → П 5	косВ);	и в косВ.
...	...		Третье обращение аналогично.
	a — РХ	проц угол	Переходим к программирова-
	b — РВ	(arg a,	нию подпрограммы: три аргу-
	c — РС	b,	мента используются однократ-
	уг — РХ	c,	но, поэтому оставляем их в РХ,
	косинус — РХ	уг,	для b и c отводим регистры, от-
		косинус);	личные от занятых в головной
			программе.
50	x → П С		Достаем аргументы из стека, за-
51	F O x → П В		поминаем их в регистрах и сра-
53	F x <sup>2</sup> ↔		зу же используем в вычисле-
			ниях.
55	F x <sup>2</sup>	(b <sup>2</sup> — a <sup>2</sup>	Здесь в вычислениях несколь-
57	П → x С F x <sup>2</sup>	+ c <sup>2</sup> )	ко нарушены строгие правила,
59	+ 2	/(2 · b · c)	но можно записать алгоритм и
61	П → x В x		так, чтобы строгий перевод дал
63	П → x С x		именно такую оптимальную
			программу.
65	+ В†	→ косинус	Второй результат — в стек.
67	F cos <sup>-1</sup>	уг:= cos <sup>-1</sup> (косинус);	и первый — в стек.
68	В/О	возврат	

Чтобы окончательно понять принцип передачи данных через параметры, попробуем ответить на вопрос: при входе в подпрограмму (перед выполнением команды с адресом 50) можно ли определить, из какой переменной значение попало в РХ, РУ и PZ? Разумеется, нет. То, что находится в РХ подпрограмма именует c, то, что в РУ — b и т. д.

Среди подпрограмм выделяется важный частный случай, когда подпрограмма имеет единственный выходной параметр. Такие подпрограммы называют подпрограммами-функциями.

Эти подпрограммы имеют особенности как в описании, так и в обращении к ним. Поскольку у них только один выходной параметр, то все остальные параметры — входные, а значит, в списке параметров нет необходимости указывать, что есть аргументы, а что — результат. В качестве единственного выходного параметра используется имя функции, которому присваивается значение при возврате из подпрограммы. Общий вид описания функции таков

**функция** *имя* (*список аргументов*);

*описания локальных переменных*

*операторы*

**возврат** *выражение*

Слово **функция** можно сокращать до **функ.** Упомянутое после слова **возврат** выражение вычисляется и присваивается имени функции.

Например, использованная в одном из предыдущих примеров функция  $y(x) = e^x - 5$  вычисляется с помощью подпрограммы

**функция**  $\Phi$  (*числ*  $x$ );

**возврат**  $e^x - 5$

Обращение к функции имеет такой же вид, как и обращение к процедуре: задаются имя и фактические параметры, но поскольку имя функции получает значение, то это обращение может быть использовано в выражении на правах простой переменной (называется оно *указателем функции*). С применением функции алгоритм вычисления определенного интеграла можно записать так (используется описанная выше функция  $\Phi$ ) :

**алг** интеграл;

*числ*  $x, h, M$ , *инт*;

**ввод**  $\{a\}$   $x$ ;  $h := (\text{ввод}\{b\} - x) / \text{ввод } M$ ; *инт* := 0;

**повторять**  $M$  **раз** *инт* := *инт* +  $\Phi(x)$ ;  $x := x + h$  **кпвт**;

**вывод** *инт*;

**кон**

Если раньше запись "*инт* := *инт* +  $f(x)$ " была условной и подразумевала, что вместо  $f(x)$  стоит какая-то конкретная функция, то эта запись не условная: сейчас правил вычисления  $\Phi(x)$  в алгоритме нет и быть не должно — они есть только в подпрограмме-функции. В алгоритме задается значение  $x$ , функция возвращает вычисленное значение.

Программирование функций на ПМК ничем не отличается от программирования процедур: аргументы так же засылаются в стек, а единственный результат функции остается в РХ, и его можно использовать в арифметических расчетах. Запрограммируем вычисление интеграла с применением функции.

Адрес	Команды	А-язык	Комментарий
алг интеграл;			
счетчик — Р0			
х — Р1		числ х,	
h — Р2		h,	
М — Р0		М.	М совмещено в памяти со
инт — РХ		инт;	счетчиком
00	х → П 1	ввод $\left\{ \begin{matrix} a \\ x \end{matrix} \right\};$	
01	С/П П → х 1 —	(ввод $\left\{ \begin{matrix} a \\ x \end{matrix} \right\} - x)$	
04	С/П П → х 0 ÷	/ЕВ0; $\lambda^{\frac{1}{2}}$	
05	х → П 2	→ $\lambda;$	
повторять М раз			
06	0	инт := 0;	
07	П → х 1	инт +	Вызвав в РХ значение х, обращаемся к подпрограмме-функции, ее результат тут же используем в выражении.
08	ПП 21	Ф (х)	
10	+	→ инт;	
11	П → х 1 П → х 2 х :=		
13	+ х → П 1 ↔	х + h;	
16	FL0 07	кпвт	
18	П → х 2 С/П	вывод инт;	кон
	х — РХ	функция Ф (числ х);	Начиная с очередного свободного адреса, размещаем функцию.
21	F e <sup>х</sup> 5 —	возврат	Аргумент — в РХ, результат оставляем там же.
24	В/О	e <sup>х</sup> — 5	

На первый взгляд, выделив в этом примере подпрограмму-функцию, мы только проиграли: программа удлинилась на три ячейки. Но это — плата за универсальность. Если предыдущий вариант алгоритма вычисления интеграла был жестко привязан к одной функции, команды вычисления которой были вписаны

в программу, то этот вариант годится для любой функции. Чтобы изменить функцию в старом варианте программы надо было менять команды в середине цикла, и если новая функция требовала бы для своего вычисления больше команд, чем старая, то пришлось бы сдвигать часть команд, при этом поменялись бы адреса и фактически пришлось бы переписывать существенную часть программы. В новом же варианте инструкция по смене функции весьма проста: с адреса 21 ввести команды вычисления функции — и все. А в алгоритмах, где к функции обращаются более чем из одного места, получается выигрыш и в занимаемой памяти.

Поскольку аргументы и результаты передаются через стек, ясно, что их не должно быть более четырех (это — особенности конкретного семейства ПМК). Но для передачи через параметры массивов этот способ в любом случае не годится, так как он очень неэкономичен и по времени, и по объему памяти. Для массивов применяется другой способ передачи: подпрограмме сообщается *адрес, где расположен массив*, тем самым ей доверяется "чужая" память и разрешается с ней работать. В принципе таким способом можно передавать и простые переменные, особенно те, которые являются как аргументами, так и результатами, т.е. переменные, чье значение подлежит изменению с использованием старого значения. Заметим, что одна и та же переменная в списке формальных параметров дважды встречаться не может, но двум разным формальным параметрам может соответствовать один и тот же фактический. Пусть, например, необходимо получить сумму элементов, содержащихся в разных массивах. Это можно реализовать с помощью процедуры, одним из параметров которой является адрес переменной, где накапливается сумма; процедура берет старое значение этой переменной и добавляет туда значения элементов массива. Процедура должна выглядеть так:

проц добавление (арг косв—  $k$ , числ  $M$ , косв  $c$ );

цикл  $M$  раз  $[c] := [c] + [^{-k}]$ ; кпвт;

возврат

Все три параметра процедуры — это аргументы, один из которых обычный числовой, а два других — косвенные, т.е. принимающие в качестве значений адреса. Результатов, которые остаются в стеке, эта подпрограмма не имеет, все нужные изменения она осуществляет в регистрах с указанными ей адресами. Фактическими параметрами, соответствующими формальным с описателями косв, косв+ и косв—, должны быть адреса некоторых переменных.

Если нужно получить сумму пяти элементов таблицы *A* и трех элементов таблицы *B*, то к описанной процедуре нужно обратиться так:

... числ *A* [1:7], *B* [2:4], сумма; ...

... сумма:=0; добавление (адрес *A*[6], 5, адрес сумма) ;

добавление( адрес *B*[5], 3, адрес сумма) ; ...

Программирование на ПМК даст следующую программу:

Адрес	Команды	А-язык	Комментарий
	<i>A</i> [1:7] – P3–P9 <i>B</i> [2:4] – PA–PC сумма – P2	числ <i>A</i> [1:7], <i>B</i> [2:4], сумма;	Из массива в семь элементов будем использовать только пять.
08	0 <i>x</i> → П 2	сумма:=0;	
10	8 В†	добавление (	Адрес <i>A</i> [6] – в стек, число элементов – в стек, адрес <i>сумма</i> – тоже в стек. Обращаемся к подпрограмме.
12	5 В†	адрес <i>A</i> [6], 5,	
14	2	адрес <i>сумма</i> );	
15	ПП 70		
17	1 3 В†	добавление (	Второе обращение аналогично.
...	..	...	
Счетчик – P0 <i>k</i> – P1 <i>M</i> – PX <i>c</i> – PD		проц добавление ( косв – <i>k</i> , числ <i>M</i> , косв <i>c</i> );	Память для головной программы и подпрограмм распределяется одновременно, чтобы отвести нужные регистры счетчикам и косвенным переменным.
70	<i>x</i> → ПD FO	повтор <i>M</i> раз	Достаем аргументы из стека и рассылаем в нужные регистры.
72	<i>x</i> → П0 FO		
74	<i>x</i> → П1		
75	К П → <i>x</i> D	[ <i>c</i> ] + [– <i>k</i> ] → [ <i>c</i> ]	По адресу регистра, где хранится сумма, значение суммы извлекается, изменяется и помещается назад.
76	К П → <i>x</i> 1		
77	+ К <i>x</i> → П D		
79	F L0 75	кпвт;	
81	В/О	возврат	

Понятно, что можно одну часть результатов передавать через стек, а другую часть – через косвенные переменные или глобальные переменные, комбинируя эти способы передачи произвольно.

### Задачи

1. Написать функцию, вычисляющую  $M!$ , и с ее помощью подсчитать число сочетаний по  $k$  предметов из  $n$ :  $C_n^k = \frac{n!}{k!(n-k)!}$ , и число размещений  $k$  предметов из  $n$ :  $A_n^k = \frac{n!}{(n-k)!}$ .

2. Написать функцию, определяющую произведение целых чисел между числами  $K$  и  $M$ , и с ее помощью вычислить  $C_n^k = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k}$ .

3. Написать процедуру, по которой выполняется подсчет числа положительных элементов в таблице, где все параметры задаются адресами: адрес таблицы, число элементов в ней и параметр-результат.

## 9. ВЗАИМОДЕЙСТВИЕ НЕСКОЛЬКИХ ПОДПРОГРАММ

До сих пор мы использовали только по одной подпрограмме в алгоритме, однако один алгоритм может включать произвольное число подпрограмм, причем не только головной алгоритм может обращаться к подпрограмме, но и одна подпрограмма может обратиться к другой. В этом случае возникает проблема с запоминанием возвратов, которая решается следующим образом. Регистр возврата на самом деле является одним из регистров стека возвратов, содержащего пять регистров (см. рис. 2). Команда "ПП" при занесении адреса в регистр возврата сдвигает весь стек (подобно тому, как это делает со стеком операционных регистров команда "П  $\rightarrow$  х"), а команда "В/О" пересылает содержимое регистра возврата в счетчик адреса и двигает стек в обратную сторону. Таким образом, возможна цепочка из пяти обращений подпрограмм друг к другу, а большего практически и не требуется.

Если алгоритм вычисления интеграла оформить как функцию, то мы получим возможность вычислять в одной программе интегралы на разных отрезках и с разной точностью. Добавим еще в алгоритм непосредственное обращение к функции для иллюстрации взаимодействия нескольких подпрограмм.

алг подпрограммы;

числ  $a, b, c, \phi$ , ин1, ин2;

...  $\phi := \Phi(a/2)$ ;

ин1:=интегр( $a, b, 10$ ), ин2:=интегр( $a, c, 20$ ); ...

кон

функция интегр( $a, b, M$ );

числ ин,  $H$ ,  $x$ ;

$H := (b - a) / M$ ;  $x := a$ ; ин := 0;

повторять  $M$  раз ин :=  $\Phi(x)$  + ин;  $x := x + H$ ; кпвт;

возврат ин ·  $H$

функция  $\Phi(x)$ ;

возврат  $e^x - 5$

(в списках параметров функций опущен описатель числ). Пишем программу.

Адрес	Команды	А-язык	Комментарий
а...ин2 – P1..P6 алг подпрограммы; числ $a$ , $b$ , $c$ , $\phi$ , ин1, ин2;			
...			
05	П → x1 2 ÷	$\phi :=$	Значение аргумента – в РХ, команда "ПП" засыпает в регистр возврата адрес 10.
08	ПП 70	$\Phi(a/2)$ ;	
10	x → П 4		
11	П → x1 П → x2	интегр	Теперь команда "ПП" зашелет в стек возвратов адрес 17, а после возврата на адрес 17 стек будет пуст.
13	1 0	( $a$ , $e$ ,	
15	ПП 40	10)	
17	x → П 5	→ ин1;	
18	П → x1 П → x3	интегр	Здесь в стек будет заслан адрес 24, который после возврата будет удален из стека возвратов.
20	2 0	( $a$ , $c$ ,	
22	ПП 40	20)	
24	x → П 6	→ ин2;	
...			
функция интегр (			
$a$	– РХ	$a$ ,	Все параметры используются по одному разу, поэтому хранятся в РХ.
$b$	– РХ	$b$ ,	
$M$	– Р0	$M$ ),	
ин	– РХ	числ ин,	$M$ совместим в памяти со счетчиком.
$H$	– Р7	$H$ ,	
$x$	– Р8	$x$ ;	
счетчик	– Р0		
40	x → П 8 F0		Извлекаем из стека $M$ , $a$ и $b$ подлежат хранению в РХ, поэтому работа с ними нестандартна.
42	← x → П 8	$x := a$ ;	
44	– П → x 0	( $b - a$ )	
46	÷ x → П 7	$ M \rightarrow H$ ;	



Адрес	Команды		А-язык	Комментарий
48	0		инт:=0;	
			повторять $M$ раз	
49	$\Pi \rightarrow x 8$	ПП 70	$\Phi(x)$	Подпрограмма обращается к другой подпрограмме.
52	+		+ин $\rightarrow$ ин;	
53	$\Pi \rightarrow x 8$	$\Pi \rightarrow x 7$	$x + H$	
55	+ $x \rightarrow \Pi$	8	$\rightarrow x$ ;	
57	FL0	49	кпвт;	
59	$\Pi \rightarrow x 7$	$x$ В/О	возврат ин $\cdot H$	
			функция $\Phi(x)$ ;	
	$x - PX$			
70	...			Функция может быть любой.
75	В/О			

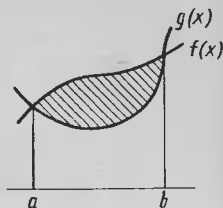
При выполнении команды "ПП" по адресу 50 находившийся в стеке возвратов адрес 17 или 24 (в зависимости от того, из какого места головной программы выполнено обращение к функции *интегр*) уходит в глубину стека, а в окно стека (регистр возврата) заносится адрес 52. Команды "В/О" по адресу 75 выполняет возврат на адрес 52 и удаляет его из стека возвратов. При втором проходе цикла, содержащего команды 49...59, адрес 52 снова попадает в стек, а затем выбрасывается из него командой "В/О" по адресу 75. И, наконец, после  $M$  таких "качаний" стека срабатывает команда "В/О" по адресу 62, которая выполняет возврат на адрес 17 или 24 и опустошает стек.

**Замечание.** Попытка выполнить "В/О" без предварительного выполнения "ПП" (т.е. при пустом стеке возвратов) приводит к переходу на адрес 01. Видимо, это просто ошибка в конструкции ПМК.

Недостатком рассмотренной выше функции *интегр* является то, что она жестко привязана к вычислению интеграла от одной функции. Было бы естественно иметь возможность задавать ей функцию так же, как значения  $a$ ,  $b$  и  $M$ . Тогда в одной программе можно было бы вычислять интегралы от разных функций. Например, показанная на рис. 16 площадь фигуры, ограниченной двумя кривыми выражается как разность интегралов

$$S = \int_a^b f(x) dx - \int_a^b g(x) dx.$$

Рис. 16. К определению площади фигуры



Чтобы одна и та же подпрограмма имела возможность работать с разными функциями, эти функции должны передаваться ей как параметры. А для реализации такой возможности в А-язык необходимо ввести еще один вид описателей: проц и функ. Переменные с такими описателями могут получать значения в виде адресов программной памяти. Их целесообразно использовать только в качестве параметров подпрограмм и придавать им значения, равные адресам начал подпрограмм.

С этими новыми средствами функция *интегр* получает еще один параметр. В результате можем построить такой алгоритм.

алг площадь;

числ  $a, b$ , пл, площ;

... пл:=*интегр* ( $a, b, 20$ , адрес  $g$ );

площ:=*интегр* ( $a, b, 20$ , адрес  $f$ )—пл;

...

кон

функция *интегр* ( $a, b, M$ , функ  $\Phi$ );

числ ин,  $H, x$ ;

$x:=a$ ;  $H:=(b-a)/M$ ; ин:=0;

повторять  $M$  раз ин:= $\Phi(x)$  + ин;  $x:=x+H$ ; кпвт;

возврат ин  $\cdot H$

функция  $f(x)$ ;

...

возврат ...

функция  $g(x)$ ;

...

возврат ...

Тело функции *интегр* не изменилось, но теперь  $\Phi$  — параметр, следовательно, реально будет происходить обращение не к функции с именем  $\Phi$ , а к функции, имя которой задано в фактическом параметре, т.е. к функции  $f$  или  $g$ .

Соответствующими средствами ПМК являются команды *косвенного обращения к подпрограмме* "К ПП м" (м — номер регистра). Адрес перехода эта команда берет из регистра № м, а не из ячейки программной памяти. Если задан регистр 0...3 или 4...6, то адрес предварительно уменьшается или увеличивается на единицу, но практической пользы от этого нет. Правила программирования новых конструкций таковы.

1. Переменной с описателем **проц** или **функ** отводятся регистры 7...D (как и переменным с описателем **косв**).
2. Обращение к процедуре или функции, являющейся параметром выполняется командой "К ПП".

Программа вычисления площади на ПМК получается такой.

Адрес	Команды	А-язык	Комментарий
	а...площ — P1...P4	алг площадь; числ а, b, пл, площ;	
...			
10	П → x 1	П → x 2	интегр (
12	2 0	В ↑	а, b, 20,
15	8 0		адрес g)
17	ПП 38	х → П 3	→ пл;
			Четвертый параметр — адрес начала функции g.
20	П → x 1	П → x 2	интегр (
22	2 0	В ↑	а, b, 20,
25	7 0		адрес f)
27	ПП 38		функции другой.
29	П → x 3 — x → П 4	—пл → площ;	
...			
	Ф — P9	функция интегр (а, b, M, функ Ф);	Распределение регистров прежнее, еще один регистр отведи для Ф.
38	х → П 9	FO	
40	...		Далее весь текст программы без изменений, за исключением одной замены: "ПП мм" заменено на "К ПП 9".
...			
50	К ПП 9	Ф (х)	
...			
70	...	функция f	С адреса 70 начинается вычисление функции f,
80	...	функция g	а с адреса 80 — функция g.
...			

Обратите внимание, что мы не можем записать вычисление площади одним оператором:

площ:=интегр( $a$ ,  $b$ , 20, адрес  $f$ )—интегр( $a$ ,  $b$ , 20, адрес  $d$ )

поскольку его нельзя запрограммировать по правилам ПОЛИЗа (точнее, записать-то можно, но программировать выражения с обращениями к функции следует с осторожностью). Во-первых, при программировании второго обращения к функции засылаемые в стек четыре параметра вытолкнут из стека все, что там было. Чтобы не сталкиваться с подобными ошибками, мы примем правило:

*обращение к функции должно быть первым операндом выражения.*

Как следствие: в одном выражении должно быть не более одного обращения к функции. Это — особенность данного семейства ПМК, вызванная малой глубиной стека.

Вторая связанная с функциями особенность — это так называемая *корректная работа со стеком*. Подпрограмма-функция, вычислявшая  $e^x - 5$ , работала со стеком *корректно*: она брала из него свой аргумент, оставляла там результат, а более ничего в стеке не изменяла. А функция *интегр* работает со стеком *некорректно*: помимо результата она оставляет в стеке значение  $H$ , которое оказывается в РУ. Это может помешать использованию такой функции в выражении, где также используются хранимые в РХ переменные. Выводы из сказанного выше таковы: при использовании подпрограмм-функций следует внимательно следить за стеком.

Тем же способом, что и подпрограммы, можно передавать через параметры и метки. Формальный параметр должен иметь описатель метка и использоваться в операторе перейти к. Соответствующий фактический параметр должен быть адресом метки, т.е. адресом начала оператора, перед которым стоит эта метка. Выход из подпрограммы может выполняться и оператором перейти к, где упомянута метка из головной программы или параметр-метка, в последнем случае происходит переход в головную программу на метку—фактический параметр. Это средство используется крайне редко, но найти применение может. Например, пусть надо построить программу, в которой используются обе нестандартные конструкции, показанные на рис. 14, один раз надо закончить цикл при обнаружении вхождения значения  $x$  в массив, а другой раз — продолжать его. Тогда целесообразно сделать подпрограмму поиска заданного

элемента в массиве; одним из параметров этой подпрограммы будет метка, на которую надо выйти при обнаружении элемента.

проц поиск ( $x$ , метка нашли);

$k := \text{адрес } a [M + 1];$

повторять  $M$  раз если  $[\neg k] = x$  то перейти к нашли все;

кпвт

возврат

(здесь  $k, M, a [1:M]$  — глобальные переменные).

С помощью этой процедуры обе нестандартные конструкции строятся очень легко и коротко:

алг ...

повторять

*вычисление  $x$ ; поиск ( $x$ , адрес Вых); обработка  $x$ ;*

кпвт;

Вых: повторять

*вычисление  $y$ ; поиск ( $y$ , адрес Обр);*

*перейти к  $K$ ;*

Обр: *обработка  $y$ ;*

кпвт;

$K$ : ...

Из подпрограммы *поиск* возврат осуществляется на следующий за обращением к ней оператор, если поиск был безуспешен, и на метку, если поиск был успешен.

**Замечание.** Перед фактическими параметрами, являющимися метками и именами подпрограмм, слово *адрес* можно не ставить: оно подразумевается, так как в отличие от простой переменной нет выбора, что передавать: адрес или значение. ~

Реализация этого нового средства на ПМК выполняется с помощью команд "К БП  $m$ " — косвенный безусловный переход. Они полностью аналогичны командам "К ПП", но не запоминают возврат. Есть также и практически крайне редко употребляемые команды "К  $X=0$ ", "К  $X \neq 0$ ", "К  $X < 0$ " и "К  $X \geq 0$ " — косвенные условные переходы. Покажем использование этих команд на рассмотренном выше примере.

Адрес	Команды	А-язык	Комментарий
$x$	- P2	числ $x$ ,	Переменные $k, M, A$ [1:M] описаны в го- ловной программе и доступны подпрограмме.
$M$	- P3	$M$ ,	
$A[1:M]$	- P?÷PC	$A[1:M]$ ;	
$k$	- P1	косв	

...

10	...	повторять	
...		вычисление $x$	

20	П → $x$ 2	4 5	поиск ( $x$ , Вых);	Второй параметр — 45 — адрес метки Вых.
23	ПП	60		

... обработка  $x$

43	БП	10	кпвт;	
45	...		Вых: ...	Сюда надо выйти по мет- ке. Вторая конструкция ана- логична.
...				

$x$	- P4	проц поиск		Для параметра метки бе- рем регистр из числа P7...PD.
нашли	- PD	( $x$ ,		
счетчик	- P0	метка нашли);		

60	$x \rightarrow$ П 1	FO	
62	$x \rightarrow$ П 4		

63	1 3 $x \rightarrow$ П 1	$k :=$ адрес $A[M+1]$ ;	
----	-------------------------	-------------------------	--

66	П → $x$ 3 $x \rightarrow$ П 0	повторять $M$ раз	
----	-------------------------------	-------------------	--

68	К П → $x$ 1	если [ $-k$ ]	При обнаружении нуж- ного числа в массиве делаем выход на адрес, хранящийся в PD.
69	П → $x$ 2	— $x$	
71	К $X \neq 0$ D	=0 то	
		перейти к нашли; все	

72	F L 0	68	кпвт;
----	-------	----	-------

74	V/O	возврат	
----	-----	---------	--

Наличие меток предполагает использование нестандартных конструкций, что вообще требует аккуратности. Выходы из подпрограмм по оператору **перейти к** можно употреблять только в подпрограммах, к которым обращаются исключительно из головной программы (не из другой подпрограммы): это связа-

но с тем, что при выходе из подпрограммы по метке в стеке возврата остается адрес возврата из этой подпрограммы и его оттуда никак нельзя убрать. Следовательно, этот прием можно применять только тогда, когда информация в стеке возвратов нам не нужна. После выхода в главную программу в стеке остается неиспользованный адрес возврата, но он там не мешает, так как глубже в стеке ничего нет.

#### Задачи

1. Простейший способ решения дифференциального уравнения

$$y' = f(x, y) \quad y(a) = C_0$$

— это метод Эйлера; при заданном  $H$  расчеты выполняются по формулам

$$y(a + H) = y(a) + f(a, y(a)) H$$

$$y(a + 2H) = y(a + H) + f(a + H, y(a + H)) H$$

...

$$y(x + H) = y(x) + f(x, y(x)) H$$

...

и проводятся до тех пор, пока  $x$  не достигнет некоторого значения  $b$ . Написать процедуру, выполняющую решение дифференциального уравнения на отрезке  $[a, b]$  с шагом  $H$  для функции  $f$ , которая также является параметром, и выводящую получаемые значения  $y$ . Выполнить расчеты для отрезка  $[1, 2]$  с шагом 0.1 и 0.05 для функции  $f(x, y) = y/x^2$  при  $y(1) = 1$ .

## Глава 4. УДОБСТВО ИСПОЛЬЗОВАНИЯ ПРОГРАММЫ И ЕЕ КАЧЕСТВО

### 1. ОРГАНИЗАЦИЯ ДИАЛОГА С ПОЛЬЗОВАТЕЛЕМ

Любая программа ведет какой-то обмен информацией с внешней средой. На больших ЭВМ информация может поступать в машину от человека, от другой ЭВМ или от какого-то датчика, установленного на любом функционирующем объекте. Информацию машины выдают на дисплей или бумагу в виде символов для чтения человеком, магнитных сигналов, записанных на магнитную ленту или диск и предназначенных для чтения другой ЭВМ, или же непосредственно в виде управляющих сигналов, поступающих на органы управления объектом. Если программа не получает никакой информации, то она может выполняться только по тем данным, которые заложены в ней

самой. Так выполняются некоторые управляющие физическими объектами программы, но счетные программы такого сорта редки: они всегда будут выдавать одно и то же, а значит, есть смысл такую программу выполнить только однажды и записать результаты. Если же в ходе программы ничего не выдается во внешнюю среду (ничего не выводится), то она просто никому не нужна: результаты ее выполнения остаются в памяти ЭВМ.

Итак, ввод и вывод — это весьма существенные компоненты программы. От их организации зависит удобство эксплуатации программы, а следовательно, и надежность результатов: если работать удобно, то меньше шансов сделать ошибку. Кроме того, следует учитывать, что с программой нередко работает человек, не являющийся ее автором, не желающий знать ее внутреннего устройства, а возможно, и вообще не знакомый с программированием (все большее число пользователей ведут диалог с большими ЭВМ, не зная программирования, они отвечают на вопросы ЭВМ или командуют ею на простейшем языке).

Рассмотрим приемы ввода-вывода (сокращенно в/в), применимые на ПМК. Существует четыре приема, два из которых мы уже применяли.

1. Р-ввод и Р-вывод (регистровый ввод-вывод). При останове программы (в частности, перед началом выполнения или после окончания) данные заносятся в адресуемые регистры командами "х→П" или читаются из регистров командами "П→х", выполняемыми в режиме вычислений.

2. С/П-ввод и С/П-вывод. Данные набирают или читают по одному числу при останове программы по команде "С/П", после чего программа запускается дальше нажатием клавиши "С/П".

3. СТ-ввод и СТ-вывод (стековый ввод-вывод). При останове программы данные засылаются в стек командой "В↑" или читаются из стека командой "FO", после чего программа запускается дальше нажатием клавиши "С/П". Это напоминает получение аргументов и выдачу результатов подпрограммой.

4. ПП-ввод и ПП-вывод. Это разновидность С/П-ввода (вывода), при котором между командами "х→П" ("П→х"), засылающими в регистры (вызывающими из регистров) значения, не ставят команды "С/П", а выполняют эти команды в режиме пошагового прохождения с помощью клавиши "ПП" (откуда и название). Например,

**ППвывод  $a, b, c$ ;**  
программируется так:



Адрес	Команда	А-язык	Комментарий
...	П → x A С/П П → x B	ППывод a ,  b,	Вызываем на индикатор первое значение и останавливаем работу, после нажатия клавиши "ПП" выполняется эта команда и на индикаторе читаем значение b, еще одно нажатие клавиши "ПП" — и читаем с.
	П → x C	c;	Нажатием клавиши "С/П" и запускаем программу дальше.

Аналогично программируется и ввод: первая команда "С/П" останавливает программу, после набора числа нажимаем клавишу "ПП", после набора последнего числа — клавишу "С/П".

Сравним достоинства и недостатки этих способов.

Вид	Достоинства	Недостатки
Р	1. Сокращает текст программы. 2. Можно повторно не вводить неизменные данные, если программа их не портит.	1. Требуется много ручных действий, где легко ошибиться. 2. Требуется знания распределения памяти в программе.
С/П	Наиболее прост в работе; можно работать одной клавишей.	Удлинит текст программы.
ПП	Сокращает текст программы по сравнению с С/П—в/в, но менее, чем Р—в/в.	Требуется внимания при попеременной работе клавишами "С/П" и "ПП".
СТ	Экономит адресуемые регистры.	1. Применим при в/в не более четырех чисел, при вводе числа используются только один раз. 2. Требуется внимания при работе клавишами "С/П", "В↑" и "F O".

Исходя из этой таблицы, можно дать рекомендации по использованию разных видов в/в. В примерах мы использовали Р-ввод только для того, чтобы не отвлекаться на многократное программирование знакомых конструкций. Из-за недостатков этот способ применяется в одном случае: когда при других способах организации в/в программа не уместится в памяти.

Наличие неизменяемых исходных данных, которые нежелательно повторно вводить, не является основанием для применения Р-ввода: удобную работу можно организовать и с С/П-вводом. Например, в расчете, где исходными данными являются  $a$ ,  $b$ ,  $x$  и  $y$ , причем каждой паре  $a$  и  $b$  соответствуют множество значений  $x$  и  $y$ , программу можно организовать так:

ввод  $a, b$ ;

повторять

ввод  $x, y$ ; выполнение расчета;

кпвт

После ввода  $a$  и  $b$  программа выполняется в бесконечном цикле, при этом каждый раз вводятся значения  $x$  и  $y$ . А если нужно заменить значения  $a$  и  $b$ , то программа запускается сначала.

Основным приемом работы следует считать С/П-в/в. Если, чтобы уместить программу в памяти не хватает нескольких ячеек, переходят к ПП-в/в. Чуть большую экономию может дать СТ-в/в, но у этого приема все же очень узкая область применения. И уж в крайнем случае переходят к Р-в/в.

Теперь рассмотрим, что и как следует вводить и выводить. Во-первых, вводить нужно только то, что может в принципе изменяться. Например, если в расчете участвует скорость света  $c = 299998.9 \cdot 10^3$  м/с, то ее не следует задавать в виде исходных данных, а нужно заносить в РХ командами набора числа. Разумеется, эти 10 команд должны работать вне цикла. И только в случае, когда нужно сэкономить несколько команд, а все другие ресурсы экономии исчерпаны, можно потребовать от пользователя вводить эту константу.

При организации ввода следует соблюдать некоторую естественность: например, при задании отрезка  $[a, b]$  естественнее сначала задавать  $a$ , затем  $b$ . Во избежание ошибок именно так и нужно организовывать ввод. А если в программе нужно получить разность  $(b-a)$ , для чего удобнее было бы сначала ввести  $b$ , то лучше добавить команду " $\longleftrightarrow$ " в текст программы. Ввод массива организуется циклом, и если память позволяет, то можно обеспечить пользователю небольшой комфорт, выводя на индикацию номер того элемента массива, который нужно набирать (хотя это и требует времени на программирование и ввод дополнительных команд в память);

$k := \text{адрес } A [M + 1];$

меняя номер повторять  $M$  раз

вывод номер; ввод  $[k];$

кпвт

Это при распределении регистров номер —  $P0, k - P1, A [1:M] - P? \div PD$  даст такой фрагмент программы:

Адрес	Команды	А-язык	Комментарий
10	1 4 $x \rightarrow П 1$	$k := \text{адрес } A[M+1];$	
13	... $x \rightarrow П 0$	меняя номер повтор $M$ раз	Вместо точек ставится значение $M$ или команда чтения $M$ .
15	$П \rightarrow x 0$ С/П	вывод номер;	При останове видим номер в виде "0000000к.",
17	K $x \rightarrow П 1$	ввод [ - $k$ ]	набираем значение элемента
18	F L0 15	кпвт	и нажимаем клавиши "С/П".

Разработка алгоритма обычно начинается с определения того, что дано и что требуется получить. Соответственно в записи сначала появляются строки "ввод..." и "вывод...", а затем между ними появляются операторы, выполняющие переработку. На больших ЭВМ, где экономия нескольких ячеек памяти не нужна, алгоритм обычно остается в таком виде: сначала вводят все необходимые данные, затем выполняют вычисления и выводят все результаты (исключение составляют случаи обработки очень больших массивов данных). При программировании на ПМК, где каждая ячейка памяти и каждый адресуемый регистр на счету, обычно применяют другую организацию алгоритма: данные вводят тогда, когда они могут быть сразу использованы в вычислениях (что, правда, возможно не всегда), результаты выводят сразу же по их получении (опять же по возможности). Это позволяет экономить и адресуемые регистры и команды, нужные для засылки или вызова значений.

Например, вычисление многочлена по схеме Гернера можно запрограммировать так, чтобы коэффициенты запрашивались по очереди, непосредственно перед их использованием:

алг многочлен;

числ  $x$ ,  $M$ ,  $mn$ ;

ввод  $x$ ,  $M$ ,  $\{c_0\}$   $mn$ ;

повторять  $M$  раз  $mn := mn \cdot x + \text{ввод}\{c_i\}$ ; кпвт;

вывод  $mn$ ;

кон

Программа получается очень короткой и выгодна даже для однократного употребления.

Адрес	Команды	А-язык	Комментарий
$M$ , счетчик – $P0$ $x - P1$ $mn - PX$		алг многочлен; числ $M$ , $x$ , $mn$ ;	$M$ совместили со счетчиком.
00	$x \rightarrow P1$	ввод $x, M$ ,	Здесь набираем значение $c_0$ .
03	$C/P$	$\{c_0\}_{mn}$ ; повтор $M$ раз	
04	$P \rightarrow x1$	$x$ $p \cdot x +$	Набранное при этом остано- ве значение $c_i$ добавляется к $mn$ без запоминания в регис- тре.
06	$C/P$	$\{c_i\}$ $\rightarrow mn$	
08	$F L0$	кпвт; вывод $mn$ ; кон	

Получившаяся программа и короче, и выполняется быстрее, чем рассмотренный в п.5 гл.2 вариант, и не имеет ограничений на степень многочлена, но старый вариант она не заменяет, так как данная программа годится только для однократного использования коэффициентов (или же их надо многократно вводить).

Еще один элемент организации ввода-вывода — это *вывод специальных сообщений*, показывающий наличие особого случая или невозможность получить результат. Например, при поиске номера нулевого элемента в массиве в случае его отсутствия выдавался нуль: номер не может быть таким, значит — это признак отсутствия элемента. Но при решении квадратного уравнения любое значение переменных может быть принято за значение корня. В подобных случаях можно принять несколько вариантов действий. Можно ввести сигнальную переменную, принимающую значение 0 или 1: сначала выводится значение сигнальной переменной, если оно нулевое, то корней нет, если оно равно единице, то следует нажать клавишу "C/P" и на следующем останове прочитать значение корня. Но этот вариант требует дополнительного программирования и дополнительной работы на клавиатуре. В этом случае можно просто ничего не делать, так как при отрицательном дискриминанте ПМК даст сигнал ошибки ("ЕГГОГ") при извлечении корня и достаточно нажатием клавиш "F ПРГ" проверить, по какой команде произошел останов (может быть и ошибка переполнения при вычислении  $4 \cdot a \cdot c$  или  $b^2$ ). Но не всегда при невозможности получить решение программа даст сигнал ошибки "ЕГГОГ": например, при отсутствии корней на отрезке метод половинного деления все равно даст какой-то результат и нужно программировать проверку неодинаковости знаков функции и выдачи специального признака отсутствия результата, если такой контроль мы хотим поручить ЭВМ.

Для программирования таких сообщений введем в А-язык новый оператор

### стоп

останавливающий выполнение программы. Он программируется командой "С/П". В инструкции к программе с операторами стоп следует записать "останов по адресу... означает такую-то ситуацию". Можно разнообразить этот оператор выводом на индикатор какой-либо характерной константы, которая вряд ли может быть значением искомого результата (например,  $8.0 \cdot 10^{-99}$ ) или сообщения "ЕГГОГ", умышленно вызванного вычислением корня или логарифма от отрицательного числа либо выполнением несуществующей команды. На А-языке это будем изображать так:

стоп  $8 \cdot 10^{-99}$

стоп ЕГГОГ  $\{\ln(-1)\}$

стоп ЕГГОГ  $\{\text{код}=38\}$

Тогда программу вычисления корня функции можно начать оператором

если  $f(a) \cdot f(b) > 0$  то стоп ЕГГОГ  $\{\text{код}=38\}$  все;

*тело алгоритма метода деления пополам*

В любом случае после останова следует посмотреть на счетчик адреса.

**Замечание.** В литературе по ПМК можно встретить описание приемов вывода на индикатор специальных сообщений вида *Еццц.ццц* (где *ц* — цифра). Пользоваться ими не следует, так как все они основаны на упоминавшейся ошибке в реализации команды "ВП", которая может быть устранена заводом, и не все ПМК будут реализовывать такой вывод. Но было бы полезно иметь на ПМК специальные команды вывода на индикатор нечисловых сообщений (скажем, команды "К А", "К В", "К С" и "К D", выводящие символы "-", "L", "C" и "T" соответственно). Кроме того было бы неплохо, чтобы при выводе сообщения об ошибке "ЕГГОГ" высвечивались и код адреса и какой-либо код, показывающий вид отгибки (скажем, "ЕГГОГ1" — переполнение, "ЕГГОГ2" — корень из отрицательного числа и т.д.).

## 2. ОЦЕНКА КАЧЕСТВА И ОПТИМИЗАЦИЯ ПРОГРАММЫ

Задача сделать программу выполняемой более быстро или на меньшем объеме памяти возникает даже на больших ЭВМ с их быстроедействием в миллионы операций в секунду и емкостью памяти под данные в миллионы ячеек. Тем более актуальна

эта задача на малых ЭВМ, в особенности на микрокалькуляторах с их весьма небольшой памятью и малой скоростью вычислений.

Итак, написав алгоритм на А-языке или программу для ПМК, предназначенную для реального расчета, нужно оценить ее размеры и ожидаемое время выполнения. Если один из этих компонентов нас не удовлетворяет, то программа подлежит оптимизации.

Оценка программы по объему памяти для данных и длине программы весьма проста: число нужных адресуемых регистров можно определить по описанию алгоритма на А-языке, длину самой программы проще выяснить, написав ее. Оценка программы по времени выполнения точно не вычисляется: она зависит от исходных данных и весьма непростоим образом. Если число проходов цикла зачастую прикинуть несложно, то вероятность работы ветви в ветвлении можно определить только приблизительно. Но грубую прикидку сделать можно. Время работы отдельных команд МК-54 приведены в табл. 2 (из ячеек с большими адресами команды выполняются дольше), и в среднем можно считать, что ПМК работает со скоростью 2 команды/с.

Возьмем, например, программу для нахождения корня методом деления пополам (вариант с циклом раз): в ней 16 команд выполняется вне цикла, и если предположить, что ветви в среднем работают одинаковое количество раз (естественное предположение), то в цикле выполняется то 12, то 14 команд — в среднем 13. При  $a = 6$ ,  $b = 7$ ,  $\epsilon = 0.01$  цикл должен сработать  $[\log_2(\frac{a-b}{\epsilon}) + 1] = 7$  раз, т.е. должно сработать  $13 \cdot 7 + 16 = 107$  команд, что займет около 53,5 с (реально программа работала 53 с; как видим, оценка получилась весьма точной). Не всегда ее удастся получить столь точной, но оценить, 10 мин необходимо на выполнение или 1 ч, всегда можно.

*Оптимизировать программу по памяти нужно только тогда, когда она не умещается в память.* В противном случае не стоит тратить время на поиск оптимального решения, проще ввести еще несколько команд в ПМК: все равно незаполненные ячейки и регистры не используются. Дать столь же однозначную рекомендацию о целесообразности оптимизации по времени затруднительно, так как она существенно зависит от многих причин. Если нужно выполнить 20 расчетов по одной программе с разными данными, где на расчет требуется 5 мин, то это — 100 мин. Если, подумав 10 мин, можно ускорить программу вдвое, то 10 мин на раздумье и 50 мин на расчеты — итого 60 мин, т.е. 40 мин экономии времени, значит, есть смысл оптимизировать.

Время выполнения команд микрокалькулятора

Команда	Время	Команда	Время	Команда	Время	Команда	Время
0...9 (1-я цифра числа)	0.37	$B \uparrow$	0.4	БП	0.35	$F \sin$	1.5
		$\longleftrightarrow$	0.25	ПП	0.35	$F \cos$	1.4
0...9 (не 1-я цифра числа)	0.3	$F 0$	0.25	$B/O$	0.35	$F \operatorname{tg}$	1.25
		$\Pi \rightarrow x$	0.45	К НОП	0.15	$F \sin^{-1}$	1.3
	0.25	$x \rightarrow \Pi$	0.25	$F X ? 0$	0.4	$F \cos^{-1}$	1.3
ВП	0.25	$K \Pi \rightarrow x$	0.5	(с переходом)		$F \operatorname{tg}^{-1}$	1.25
0...9 (цифры порядка)	0.25	$K x \rightarrow \Pi$	0.35	$F X ? 0$	0.35	$F e^x$	1.7
		$F \pi$	0.25	(без перехода)		$F \lg$	1.4
/ - /	0.3	$F Bx$	0.4	$F L i$	0.4	$F \ln$	1.3
+	0.2					$F 10^x$	0.22
-	0.2					$F \sqrt{\quad}$	0.28
$\times$	0.2					$F 1/x$	0.2
$\div$	0.2					$F x^2$	0.25
$Cx$	0.3					$x^y$	2.9

**П р и м е ч а н и е.** В таблице приведено время выполнения в секундах (замерено на МК-54) для команд, расположенных в начальных адресах памяти. Команды, размещенные в последних адресах, требуют на 20...25 % больше времени.

Но если программа работает 8 ч, то может быть не стоит ее оптимизировать, а лучше, запустив ее вечером, идти ложиться спать с тем, чтобы прочитать результаты утром (ПМК допускает и круглосуточную работу). Но тогда на 20 расчетов придется потратить неделю, и если результаты нужны раньше, то все же придется заняться оптимизацией.

Строгих правил оптимизации программы не существует, однако если решение о необходимости оптимизировать программу принято, то следует знать общие принципы и направления поиска оптимальных решений.

Можно выделить три уровня оптимизации программы, которые назовем стратегической, тактической и технической оптимизациями.

Стратегическая оптимизация дает наибольший эффект и основывается на выборе алгоритма и определении того, что программа делать должна и что она делать не должна. Алгоритмы, выполняющие одну и ту же работу, различаются по скорости работы, точности результата, требуемой памяти и области применимости. Алгоритмы, имеющие хорошие показатели по одним параметрам, имеют худшие показатели по другим ("золотое правило" программирования: то, что выигрывается в скорости, теряется в памяти или в точности, и наоборот), а алгоритмы, имеющие хорошие показатели и по скорости, и по точности, и по памяти, имеют очень узкую область применения, т.е. годятся только для частных случаев. Итак, во-первых, нужно выбрать подходящий алгоритм. Например, если в программе требуется вычислять интеграл, причем программа велика, т.е. нас лимитирует емкость памяти, а время не лимитирует, то берем простейшую формулу прямоугольников, которая дает наиболее короткую программу, а требуемой точности добиваемся путем задания достаточно большого числа разбиений, т.е. ценой затрат времени. Если же, наоборот, лимитирует время, а емкости памяти хватает, то можно запрограммировать более сложную, но и более точную формулу Симпсона.

Определение того, что следует делать программно, а что вручную, основано на том, что есть вещи, которые человеку гораздо проще сделать, чем запрограммировать. Есть ли смысл программировать проверку неодинаковости знаков функции на концах отрезка, где мы собираемся искать корень, или это проще сделать вручную? Проверка принадлежности точки с вычисленными программой координатами  $(x, y)$  области, заштрихованной на рис. 17, займет до 20 команд, но если места в памяти для них нет, то можно поручить человеку: вычисленные координаты выводятся, человек по чертежу на миллиметровке определяет, попадают ли они в заданную область и вводит значение 1 или  $-1$ . Работа пойдет медленнее, но другого выхода, возможно, и нет.

Тактическая оптимизация основывается на учете взаимосвязей между значениями переменных и выборе подходящей структуры данных. Например, в том же методе половинного деления после того, как было замечено, что число повторений

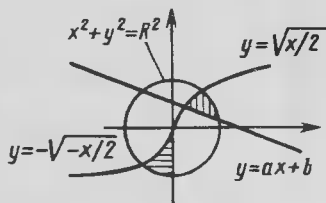


Рис. 17. Сложная область



цикла можно вычислить заранее, а не проверять выполнение условия на каждом шаге, удалось применить цикл раз вместо цикла до или пока, вследствие чего получается почти полукратное ускорение. Второй пример: пусть надо вычислить число положительных и неположительных чисел в массиве. Простейший алгоритм

$k_{пол} := 0; k_{отр} := 0; k := \text{адрес } A[0];$

повторять  $M$  раз

если  $[^+k] > 0$  то  $k_{пол} := k_{пол} + 1$  иначе  $k_{отр} := k_{отр} + 1$  все

кпвт

на первый взгляд кажется единственно возможным, но с учетом того, что в итоге  $k_{пол} + k_{отр} = M$ , можно не считать одно из них:

$k_{пол} := 0; k := \text{адрес } A[0];$

повторять  $M$  раз

если  $[^+k] > 0$  то  $k_{пол} := k_{пол} + 1$  все

кпвт;

$k_{отр} := M - k_{пол};$

и вместо операции, работающей в цикле, появляется равноценная операция, работающая вне цикла.

Выбор подходящей структуры данных хорошо иллюстрируется примером с определением стоимости покупки, где надо было определить  $\sum_i \text{ЦЕНА}_i \cdot \text{КОЛ}_i$ . Выбранная структура данных представляла собой два массива, но можно взять и один массив двойной длины, разместив в нем данные из разных массивов по очереди:  $A[2 \cdot k - 1] \sim \text{ЦЕНА}[k]$ ,  $A[2 \cdot k] \sim \text{КОЛ}[k]$ . Тогда алгоритм позволяет обойтись одной косвенной переменной, расширив возможный размер массива и сэкономив несколько команд в программе:

$\text{сумма} := 0; k := \text{адрес } A[2 \cdot M + 1];$

повторять  $M$  раз  $\text{сумма} := \text{сумма} + [^-k] \cdot [^-k]$  кпвт;

Приемы совмещения нескольких переменных в одном регистре мы уже рассматривали: здесь используются "области нужности" значений.

Как видим, всю тактическую оптимизацию можно провести еще до написания программы для ПМК. Для отражения соответствующих деталей в алгоритме можно расширять средства

А-языка. Например, можно ввести новое слово где, позволяющее указывать совмещение переменных в регистрах:

числ  $a, b, M$  где счетчик, сумма где  $A[1], A[1:5]$ ;

Такая запись означает, что переменную  $M$  следует разместить там же, где счетчик цикла, а сумму можно накапливать в регистре, отведенном первому элементу массива  $A$ .

*Техническая оптимизация программы основывается на использовании таких приемов программирования, которые эффективно реализуются микрокалькулятором.* Все эти приемы в основном рассмотрены ранее. Сюда можно отнести грамотное использование стека, в частности для хранения переменных в РХ, "чистку циклов", заключающуюся в выносе за цикл тех вычислений, которые это допускают (что мы видели на примере выноса за цикл умножения на шаг при вычислении интеграла), предварительное вычисление общих выражений (что мы видели на примере алгоритма решения квадратного уравнения, где предварительно вычислялись  $2 \cdot a$  и корень из дискриминанта), чтобы не повторять вычисления одного и того же выражения многократно. Для этой цели нужно уметь использовать особенности данного семейства ПМК. Наиболее часто применяются переходы с оставлением результата в РХ, позволяющие вынести общую часть действий с ветвей на "ствол", в результате чего экономится программная память (мы это делали, приводя алгоритм вычисления максимального числа  $k$  виду "макс:=если  $a \geq b$  то ..."). Также используют побочные эффекты косвенной адресации, заключающиеся в отсечении дробной части и автоизменении числа в регистре. Все это также можно отразить в записи на А-языке. Например, наиболее экономный способ вычисления количества положительных элементов в массиве имеет вид:

$кпол:=0; k:=адрес A[M+1]$

повторять  $M$  раз если  $[^k] > 0$  то  $[^+ кпол]$  все кпвт

где соответствующая программа имеет вид:

Адрес	Команды	А-язык	Комментарий
10	$0 x \rightarrow П 4$	$кпол:=0;$	косв+ $кпол - P4$
12	$1 \ 4 \ x \rightarrow П 1$	$k:=адрес A[M+1];$	$k - P1; A[1:M] - P?-PD$
15	$П \rightarrow x \ 2 \ x \rightarrow П 0$	повтор $M$ раз	$M - P2; \text{счетчик} - P0$
17	$K П \rightarrow x \ 1 \ /- /$	если $[^k]$	
19	$F X < 0 \ 22$	$< 0$ то	
21	$K П \rightarrow x \ 4$	$[^+кпол]$ все	
22	$F L 0 \ 17$	кпвт	

Здесь команда "К П  $\rightarrow$  х 4" считывает нечто в РХ, чем мы затем не пользуемся, но весь смысл этой команды в побочном эффекте: содержимое Р4 увеличивается на единицу — это наиболее эффективный прием организации счетчиков.

К приемам технической оптимизации по памяти можно отнести и выделение подпрограмм, что позволяет только один раз описать повторяемые в разных местах действия. Но каждое обращение к подпрограмме без параметров требует около 1 с времени.

Как показывает табл. 2, за редким исключением следует стремиться к тому, чтобы программировать фрагмент минимальным числом команд: основное время ПМК тратит на выбор команды из памяти (это — особенность ПМК). При прочих равных условиях следует отдавать предпочтение невычислительным командам: лучше поменять операнды перед делением, чем выполнять "F 1/х" после него, — и быстрее, и точность выше.

## ГЛАВА 5. ОТЛАДКА И ЭКСПЛУАТАЦИЯ ПРОГРАММЫ

### 1. ТЕСТИРОВАНИЕ

Программистский фольклор утверждает, что:  
всякая программа содержит хотя бы одну ошибку;  
если программа не содержит ошибки, то алгоритм содержит ошибку;

когда ни программа, ни алгоритм не содержит ошибок, то программа уже никому не нужна.

И хотя последнее утверждение спорно, но программа всегда является для ее разработчика новым продуктом и, как каждая новая разработка в любой области человеческой деятельности, требует доводки, которая в программировании называется *отладкой*.

Никакая технология разработки программ не гарантирует отсутствие ошибок, поэтому *наличие ошибки во вновь составленной программе является нормальным состоянием программы* и к этому следует относиться спокойно и уметь быстро находить и устранять ошибки.

Ошибки в программах можно разделить на следующие классы.

**Ошибки алгоритмизации.** Что-либо не учитывается или неправильно обрабатывается алгоритмом.

**Ошибки программирования.** Допущенная при переводе алгоритма с А-языка в команды ПМК неточность приводит к тому, что программа не соответствует алгоритму.

**Ошибки ввода.** При вводе программы в память ПМК нажаты не те клавиши, пропущены команды и т.д.

**Ошибки эксплуатации.** Заданы ошибочные исходные данные (или данные на которые программа не рассчитана). Фактически здесь нет ошибки в программе, но эффект получается точно такой же, как и при ошибках в программе.

Читая текст программы из памяти ПМК с помощью клавиш "ШГп" и "ШГл", можно обнаружить ошибки ввода, но нет гарантии, что будут обнаружены все ошибки. Потактовое прохождение с помощью клавиши "ПП" может помочь в отладке, но оно очень трудоемко и должно применяться в редких случаях.

Выполнение программы сразу на тех исходных данных, для которых нужно получить результат, обычно завершается одним из следующих четырех вариантов:

возникает ситуация "ЕГГОГ",

программа закликивается;

получается результат, который "не лезет ни в какие ворота";

получается результат, который похож на истинный, но не правилен.

Заметим, что если инженер считал на ПМК прочность конструкции или медик рассчитывал дозировку лекарств, то последний вариант просто опасен для жизни людей, если результатом начинают пользоваться. В случае первых трех вариантов автор программы начинает искать ошибку, но его поиски сильно затруднены, так как неизвестно, что должно получаться и что из получившегося верно, а что ошибочно. Поэтому перед эксплуатацией программа должна пройти проверку на отсутствие ошибок и устранение ошибок, если таковые обнаружатся. Технология этой работы включает два этапа:

*тестирование*, т.е. выполнение программы на специально подобранных исходных данных с целью обнаружения ошибки;

*отладка*, т.е. поиск причины ошибочной работы и ее устранение.

В этом параграфе рассмотрим тестирование. Во-первых, подчеркнем, что целью тестирования является обнаружение наличия ошибки, а не установление факта, что ошибки нет. При тестировании программист работает "против себя", стараясь найти ошибки в своей работе, в чем и заключается психологическая трудность этой работы, которую, однако, следует научиться преодолевать.

Тест — это специально подобранные исходные данные в совокупности с теми результатами, которые должна выдать по этим данным программа, и служащие для проверки программы.

С одной стороны, тестовые данные должны быть достаточно просты, чтобы результаты всех шагов программы можно было получить и без программы — вручную, с другой стороны, они должны позволять комплексно проверить программу. Фактически во всех предыдущих программах мы изучали их на таких тестовых данных, где заранее знали, что должно получиться.

Для проверки программы, как правило, нужно несколько тестов, чтобы при их работе была выполнена каждая команда программы. Известен принцип, гласящий, что *исчерпывающее тестирование невозможно*, т.е. тестированием нельзя доказать, что ошибок в программе нет. Но на практике можно считать достаточным тестирование каждой ветви, каждой проверки и каждого цикла.

Совокупность тестов должна быть такой, чтобы каждая ветвь программы проработала хотя бы один раз, каждая проверка условия сработала как с результатом "да", так и с результатом "нет" и каждое тело цикла сработало не менее 3 раз подряд.

Понятно, что ошибка в неработавшей ветви может остаться незамеченной, равно как может остаться незамеченным ошибочный адрес в команде условного перехода, если при тестировании переход по нему не выполнялся. Что касается циклов, то нередко ошибки, когда тело цикла в первый раз выполняется правильно, но портит какие-то данные, и второй проход цикла выполняется уже с ошибкой. Как правило, если цикл 3 раза отработал правильно, то и дальше он будет работать верно.

Итак, для фрагмента программы, по которой проверяется принадлежность точки  $x$  отрезку  $[a, b]$ , нужно не менее трех тестов: точка принадлежит отрезку, точка левее отрезка и точка правее отрезка. При этом обязательны проверки предельных случаев, т.е. нужно проверить, считает ли программа принадлежащими отрезку точку  $a$  и точку  $b$  (так можно выявить ошибки, связанные с заменой знаков " $<$ " и " $\leq$ " или " $>$ " и " $\geq$ " друг другом, что особенно актуально для ПМК, где нет полного комплекта проверок).

При тестировании программы следует широко применять выполнение ее небольших частей в пошаговом режиме. Например, если в рассмотренном выше примере  $x$ ,  $a$  и  $b$  получаются в результате сложного расчета и трудно подобрать исходные данные так, чтобы в итоге  $x = a$ , то проще поступить следующим образом: если проверка принадлежности выполняется, с адреса 40, то в режиме вычислений заносим значения в регистры, соответствующие  $a$ ,  $b$  и  $x$ , выполняем команду "БП 40", а затем с помощью клавиши "ПП" выполняем команды проверки, после команд переходов можно проверять и значения счетчика адреса путем перехода в режим программирования.

Если программа содержит подпрограммы, то они подлежат автономному тестированию, что можно сделать двумя способами. Во-первых, можно заменить в подпрограмме "В/О" на "С/П" и выполнить подпрограмму как головную программу, занеся в глобальные переменные и стек нужные значения, а после останова прочитать из глобальных переменных и стека результаты и заменить "С/П" опять на "В/О". Но лучше где-либо в свободном месте памяти разместить маленькую головную программу, содержащую только ввод, обращение к подпрограмме и вывод (с применением Р- или СТ-в/в эта программа может занять всего три команды: "ПП", адрес, "С/П"). Выполняется эта программа путем занесения командой "БП" в режиме вычислений ее начального адреса в счетчик адреса и нажатия клавиши "С/П". Команды этой программы затем можно и не удалять, так как они просто не будут выполняться.

При наличии нескольких достаточно сложных подпрограмм в программе проверку одной из них делают, заменив другую более простой подпрограммой. Например, подпрограмму вычисления интеграла следует протестировать на простейшей функции, затем протестировать автономно программу вычисления нужной функции, а затем уже выполнять все вместе — и с нужной функцией и с нужными исходными данными.

Для удобства тестирования в программу вводят промежуточные остановы, как это делалось ранее. Остановы ставят в таких точках, где контролируемые данные видны на индикаторе, хотя при останове можно просмотреть и другие регистры. После завершения тестирования команды останова удаляют путем замены командой "К НОП".

Порядок пропуска тестов, т.е. выполнение программы с разными тестами, безразличен, хотя предпочтительнее сначала пропускать более простые тесты, т.е. такие, при которых программа выполняется быстрее или работает меньшее число блоков программы. Общий порядок работы с тестами таков.

1. Программа выполняется с очередным набором тестовых данных.

2. Если результаты правильные, то она выполняется со следующими данными (на следующем тесте).

3. Если результаты не соответствуют ожидаемым, то отыскивается и исправляется ошибка (способами, описанными в следующем параграфе) и программа проверяется на всех тестах, в том числе и на ранее давших правильный результат.

4. Тестирование заканчивается, когда программа дает правильные результаты на всем комплекте тестов при условии, что между двумя выполнениями на разных тестах в нее не вносились никакие изменения.

Заметим, что после внесения любого исправления положено проверять программу на всех тестах, так как в процессе исправления одной ошибки можно внести другую.

Методика тестирования одинакова и для больших ЭВМ и для ПМК.

В качестве примера рассмотрим тестирование содержащего ошибки алгоритма поиска номера элемента с заданным значением в массиве.

алг номер данного элемента;

числ  $M, x$ , ном,  $a [1:M]$ ; косв —  $k$ ;

Рввод  $M, a, x$ ;  $k := \text{адрес } a [M]$ ;

пока  $[k] \neq x$  повторять  $M$  раз кпвт

ном :=  $k - \text{адрес } a [0]$ ; Рвывод ном;

кон

В примерах этой главы применен не рекомендованный ранее Рвывод и Рввод, чтобы не удлинять программы несущественными деталями. Алгоритм должен дать номер элемента, равного  $x$ , или ноль, если такого нет, причем более коротким способом, чем его аналог из п.5 гл.3 за счет использования адресации с автоуменьшением, что делает тело цикла пустым, так как заголовок вобрал в себя все нужные действия.

В соответствии со сказанным в п.2, гл.4 для такого алгоритма нужно не менее четырех тестов: нужный элемент находится в середине массива; элемент — первый в массиве; элемент — последний в массиве (контроль предельных случаев) и элемента, равного  $x$ , вообще нет. Возьмем массив, скажем, из пяти эле-

ментов (нужно не менее чем из трех, чтобы цикл отработал не менее 3 раз) со значениями: 1, 2, 3, 4, 5 и значения  $x=2$ ,  $x=1$ ,  $x=5$  и  $x=8$ . При  $x=2$  и  $x=1$  получим правильные результаты (независимо от того, как написана программа и как распределены регистры). Но третий тест:  $x=5$  даст результат ноль; равно как и четвертый:  $x=8$ . Допустим, что сначала был выполнен четвертый тест, для которого ноль — это правильный результат. Тогда, анализируя причину ошибочной работы третьего теста (пока не уточняя способ этого анализа), мы обнаружим, что оператор " $k:=\text{адрес } a[M];$ " нужно заменить на " $k:=\text{адрес } a[M+1]$ ". После внесения исправлений третий тест даст правильный результат, но завершить на этом тестирование — значит сделать ошибку, ибо теперь четвертый тест даст неверный результат  $ном=1$ , так как в этом алгоритме две ошибки, но при ситуации отсутствия элемента они случайно взаимно компенсировались.

Из этого примера понятно, зачем нужно проверять предельные случаи: "ошибки плюс-минус единицы", когда цикл недо-работывает или перерабатывает один раз или начинает (кон-чает) работу не на нужном элементе, а на соседнем, достаточно часты. Понятно также, что первые три теста недостаточны для проверки программы, так как при них ни разу не срабатывает выход из цикла по исчерпанию  $M$  проходов, а всегда срабаты-вает только условие после **пока**.

## 2. ПОИСК ОШИБОК

Выданные программой неправильные результаты показывают, что в программе есть ошибка, но не указывают, где именно она находится и в чем заключается. Поиск ответов на эти вопро-сы и называется отладкой. Хотя программисты и считают отлад-ку искусством, но имеется ряд чисто технических приемов, грамотное выполнение которых доводит отладку до уровня ремесла. В целом эти приемы также едины и для больших ЭВМ и для ПМК, но на ПМК процесс отладки существенно проще вследствие того, что:

- малый объем программы облегчает ее логический анализ;
- малый объем данных позволяет проанализировать все дан-ные;

всегда можно повторить выполнение всей программы или ее части, в частности, по шагам с анализом и даже коррекцией данных в памяти, что позволяет заменить умственную работу чисто технической.



Приступим к отладке программы *номер данного элемента*, алгоритм которой описан в п.1 этой главы. Итак, тестирование показало, что имеется ошибка — не обнаружен последний элемент массива. Первый шаг поиска ошибки — это *проверка, что должен сделать алгоритм именно при тех данных, которые он получил*. Переменная  $k$  получает значение адреса последнего элемента, но при косвенной адресации сначала происходит уменьшение адреса, а затем сравнение элемента массива с  $x$ , т.е. последний элемент не обрабатывается, поэтому программа его и не находит. Ошибка исправляется очень легко. Аналогично работаем и со второй ошибкой: если элемента  $x$  в массиве нет, то условие после пока всегда истинно и цикл отработает  $M$  раз. На каждом шаге значение  $k$  уменьшается на 1, значит по завершении цикла там будет адрес  $A[1]$ , т.е. *ном* получит значение 1, что и имело место. Эта ошибка — типичный пример *ошибки алгоритмизации*: алгоритм не рассчитан на случай отсутствия элемента и обрабатывает его так же, как и в случае присутствия элемента на первом месте. Исправление этой ошибки потребует коренной переделки и алгоритма, и программы.

Человек, допустивший такие ошибки, может и не обнаружить их при логическом анализе. Например, неправильный адрес может быть присвоен вследствие предположения автора, что адрес изменяется не перед, а после чтения числа. В этом случае переходим к следующему этапу: анализ содержимого памяти путем чтения регистров по команде " $\Pi \rightarrow x$ ". Если массив занимает регистры  $P9..PD$ , то в регистре, где хранится  $k$ , будет значение 8, которое и должно быть, если считать, что адрес изменяется после чтения числа. В этом случае такой прием результата не дает, так как в этом алгоритме почти нет изменяемых данных (только переменная  $k$ ), которые можно анализировать. Но применение "тяжелой артиллерии" — пошагового выполнения команды " $\Pi\Pi$ " — результат даст, и очень быстро, так как выход из цикла должен произойти на первом же шаге.

Еще один пример. Рассмотрим отладку алгоритма вычисления многочлена по схеме Горнера

алг Горнер;

числ  $mn, x, A [0:6]$ ;  $qосв+ k$ ;

Рввод  $x, A$  ;  $mn:=A[0]$  ;  $k:=$  адрес  $A [0]$  ;

повторять 6 раз  $mn:=mn \cdot x + [^+ k]$  ;  $kптв$ ;

вывод  $mn$ ;

кон

Распределение регистров:  $mH - PX$ , счетчик —  $P0$ ,  $x - P1$ ,  $A - P2...P8$ ,  $k - P9$ . При таком распределении регистров программа имеет вид:

1  $x \rightarrow P9$  6  $x \rightarrow P0$   $P \rightarrow x2$   $P \rightarrow x1$   $X$   $KP \rightarrow x9$  +  
 F L0 05 C/П

Этот алгоритм имеет только один цикл без ветвлений и с единственным выходом, поэтому здесь достаточно и одного теста. Подберем тестовые данные так, чтобы результат можно было посчитать вручную наиболее просто. Если взять  $x=10$ , а коэффициенты многочлена из одной цифры, то результат будет равен числу, состоящему из цифр-коэффициентов. Возьмем коэффициенты 7, 6, 5, 4, 3, 2, 1. Выполнив программу получаем 8111110 (должны получить 7654321).

С помощью этого результата и небольших логических рассуждений можно сразу определить обе имеющиеся в программе ошибки. Такой результат может получиться, если все коэффициенты равны 10. Делаем вывод, что все коэффициенты берутся из одного и того же регистра, причем из того, где находится  $x$  (больше нигде нет числа 10). Изучение программы подтверждает эту гипотезу: переменная  $k$  получает не тот адрес, который надо, и регистр под  $k$  отведен не соответствующий описанию  $косв+$ . Но если не помогает логика, то применяем технические приемы. Смотрим содержимое регистров:

№ регистра	Содержимое	Комментарий
0	00000001.	Правильное значение счетчика после цикла, значение $x$ правильное, значение $A[0]$ правильное,
1	10.	
2	7.	
...	...	...
8	1.	значение $A[6]$ правильное.
9	00000001.	Значение $k$ неправильное, должно быть 8.
$A$	?	Здесь и далее может быть любое число, оставшееся от предыдущих действий с ПМК.
...	...	

Ошибочное значение  $k$ , во-первых, показывает не на элемент массива  $A$ , а на  $x$ , а, во вторых, похоже, не изменяется. Смотрим в то место программы, где  $k$  получает значение, и видим первую ошибку. Разбираясь с косвенной адресацией, обнаруживаем, что взяли для  $k$  неподходящий регистр. Первую ошибку находим моментально. Если сразу не удалось сообразить, что взят неправильный регистр для  $k$ , то "тяжелая артиллерия" —

пошаговое выполнение — поможет (хотя и ценой дополнительных затрат времени): увидев, что R9 действительно не изменяет значение после команды "K П → x 9", любой поймет, что выбор регистра ошибочен.

Перераспределяем регистры: помещаем  $k$  в R4,  $A$  — в R5..RV и в результате имеем новый вариант программы, получающийся из старого заменой нескольких команд командами, работающими с другими регистрами:

```
5  x→П9 6  x→П0 П→x5 П→x1 X КП→x4 +
F L0 05 C/П
```

На том же тесте получаем результат 7654521. Смотрим регистры.

Номер регистра	Содержимое	Комментарий
0	00000001.	Значение счетчика таким и должно быть.
1	10.	Правильное значение $x$ .
2	7.	Содержимое R2 и R3 осталось от предыдущего расчета, R2 и R3 не используются в программе.
3	6.	
4	00000011.	Значение $k$ правильное.
5	...	R5...R8 содержат правильные значения,
9	5.	а значение R9 испорчено, далее все значения правильные.

Если возникло сомнение в правильности занесения исходных данных, можно исправить содержимое R9 и повторить выполнение: результат будет тот же. Где в программе имеется запись значения 5? Что это за значение? Похоже, что это адрес  $A[0]$ . И действительно, в этом месте программы ошибка: запись идет не в тот регистр.

Из этого примера можно сделать два вывода: во-первых, нужно проверять содержимое всех регистров, а не только тех, которые используются в программе (значение в неиспользуемом регистре может подсказать место ошибки), и, во-вторых, желательно проводить отладку на "чистом" микрокалькуляторе, т.е. чтобы в регистрах и программной памяти не было оставшихся от предыдущих расчетов чисел или команд, которые в результате ошибки могут "вмешаться" в текущий расчет.

В данном примере, несмотря на ошибку, у переменной  $k$

оказалось правильное значение, по чистой случайности оставшееся от предыдущего расчета. Понятно, что так можно и не заметить ошибку.

Делаем еще один следующий отсюда вывод.

Никогда нельзя ограничиваться одним тестом (даже, когда этого достаточно с точки зрения контроля ветвей и проверок), так как один тест может дать правильный результат случайно.

Уточним и понятие "чистого" микрокалькулятора. Нули в неиспользуемых регистрах мало помогают в отладке, так как ноль может получиться и в результате операции и далее может использоваться почти во всех операциях. Лучше заслат в неиспользуемые адресуемые регистры и в стек какое-либо характерное число, вероятность получения которого в результате вычислений мала, например 1.2345678 09. Число с порядком 09 в регистре сразу покажет, что в этот регистр программа ничего не заносила, а использование такого числа в операциях приведет к результату с необычным порядком, что тоже сразу подскажет характер ошибки.

**Замечание.** Еще лучший результат дало бы занесение в регистры числа 9.9999999 99, которое дало бы переполнение почти при всех действиях, но это не позволяет сделать ошибка в ныне выпускаемых моделях ПМК, в результате которой при переполнении от умножения или операции "Гх<sup>2</sup>" ПМК не останавливается, и если последовательным умножением довести результат до числа с порядком, большим 400, то индикатор ПМК гаснет и микрокалькулятор перестает реагировать на любые действия (его можно только выключить, а затем опять включить). Поэтому не следует умышленно выводить ПМК на переполнение от умножения или возведения в квадрат.

Рассмотренные ошибки представляли собой типичные примеры *ошибок программирования*. В частности, типичным является внесение ошибок при исправлениях: во втором примере "забыли" изменить номер регистра в одной из команд. Незамеченные ошибки ввода дают такой же эффект, как ошибки программирования, только исправляются иначе.

Отладим третий вариант той же программы: он отличается от второго тем, что адрес 05 в команде "F LO" заменен на 06. Такие ошибки часто возникают при дописывании или удалении команд в процессе программирования. (Команда по адресу 01 исправлена на "х→П 4".) Результат выполнения на том же тесте равен 288711. При исследовании регистров ошибки не обнаруживаем. Но кроме адресуемых регистров есть еще и стековые. Их просмотр с помощью команды "FO" показывает, что

в РУ, РЗ и РТ находится значение 5. Нетрудно понять, что в РУ должно находиться значение 6 (число повторений цикла). Это подсказывает, что мы неправильно работаем со стеком, а именно: где-то недосылаем в стек операнд и операция "X" или "+" выбирает оттуда "лишние" операнды. Команды засылок операндов в стек в программе есть, остается проверить, выполняются ли они. И на этом пути обнаруживаем, что адрес в команде "F L0" позволяет пропустить выполнение одной из команд "П → x".

Еще раз убеждаемся, что логический анализ — это очень эффективный прием отладки и что ценой временных затрат его можно заменить технической работой. Содержимое регистров нам ничего не подсказало, следовательно, надо проверить ход выполнения программы. Проведем эту проверку так, как будто мы имеем дело с большой программой, для которой пошаговая проверка клавишей "ПП" чрезмерно трудоемка.

Для анализа работы цикла нужны промежуточные остановки, но мы не позаботились о них заранее. Поэтому вставим останов сейчас, заменив им, например, команду "+" по адресу 08. Выполняем текст.

Клавиши	Индикатор	Комментарии
С/П	6.	При первом останове видим на индикаторе значение А [1].
+	76.	Замененную команду выполняем вручную;
С/П	5.	до следующего останова идем в автоматическом режиме, а замененную команду
+	461	выполняем вручную, получаем неправильный результат.

Ясно, что коэффициенты взяты правильно, но умножение идет не на  $x = 10$ , а на откуда-то взявшееся значение 6. Здесь можно понять, что это за значение 6, но допустим, что мы этого не поняли. Тогда придется проделать еще некоторый объем технической работы. Понятно, что точка останова выбрана неудачно, так как в ней мы ничего не смогли увидеть. Восстановим команду "+" и вставим останов вместо команды "X".

Клавиши	Индикатор	Комментарии
С/П	10.	На индикаторе значение $x$ ,
X	70.	выполняем замененную команду,
С/П	76.	пока все шло правильно, но сейчас на индикаторе не $x$ , а текущее значение <i>mn</i> .
F ПРГ	50 61 65 07	Остановились там, где надо, и команда
F АВТ	76.	"П → $x$ 1" перед "С/П" есть,
П → $x$ 1	10.	и переменная $x$ нужного значения.
F О	76.	
F О	6.	но в стек значение $x$ не попало.

Вывод один: команда " $\Pi \rightarrow x 1$ " не выполнялась, следовательно, переход на нее не сработал. Проверка перехода обнаруживает ошибку: неправильный адрес.

Итак, на этом примере видны все приемы отладки, включающие комбинированную работу в программном, автоматическом, пошаговом и вычислительном (автономном) режимах.

1. Организуем промежуточные остановки путем вставки команд "С/П", которые можно помещать как заранее (более предпочтительный вариант), так и вместо каких-либо команд (кроме команд перехода, занимающих две ячейки памяти, и команды "В/О", которую нельзя выполнить в автономном режиме).

2. В автоматическом режиме доходим до очередного останова, просматриваем в режиме вычислений содержимое стека и/или регистров, проверяя их правильность. (Если содержимое стека должно быть сохранено, то перед просмотром регистров содержимое стека нужно записать, а после просмотра опять ввести в стек.)

3. При необходимости отдельные фрагменты можно выполнить клавишей "ПП" в пошаговом режиме.

**Внимание!** На до сих пор выпускаемых экземплярах ПМК команда "ВП", выполняемая после операции, в пошаговом и автономном режимах дописывает порядок к результату, а в автоматическом работает совсем по-другому.

4. Если содержимое регистров в точке останова не позволяет определить ошибку, то точка останова передвигается ближе к началу ошибочно работающего участка и при необходимости ошибочный участок выполняется в пошаговом режиме.

Поиск *ошибок эксплуатации* выполняется точно так же: анализ регистров и хода выполнения программы в какой-то момент обнаруживает неправильно занесенные данные.

По сравнению с технологией отладки на больших ЭВМ можно сказать, что ПМК легко "прощает" ошибки в организации отладки: за них приходится расплачиваться избыточным потраченным на отладку временем, которое в целом оказывается не столь большим.

В качестве завершающего примера рассмотрим отладку программы, содержащей несколько ошибок всех типов. Программа решения некоторого достаточно сложного уравнения методом половинного деления, соответствующая алгоритму с ошибкой имеет вид:

алг метод половинного деления;

числ прав,лев,  $\epsilon$ , флев,  $x$ ;

Рввод прав,лев,  $\epsilon$ , флев:= $\Phi$ (лев);

повторять  $\ln$  (прав—лев)/ $\epsilon$  ) /  $\ln 2$  раз

если  $\Phi(x := (\text{прав} - \text{лев}) / 2) \cdot \text{флев} \geq 0$  то лев:= $x$

иначе прав:= $x$  все;

кпвт;

вывод  $x$ ;

кон

Распределение регистров естественное: счетчик — P0, прав — P1, лев — P2,  $\epsilon$  — P3, флев — P4,  $x$  — P5. При программировании также допущены ошибки:

P $\rightarrow$ x 2	ПП	40		P $\rightarrow$ x 1	P $\rightarrow$ x 2 -	P $\rightarrow$ x 3	$\div$	F ln 2
F ln	$\div$	x $\rightarrow$ P 0		P $\rightarrow$ x 1	P $\rightarrow$ x 2 -	2	$\div$	x $\rightarrow$ P 5 C/P
ПП	40	P $\rightarrow$ x 4	x	F X $\geq 0$	30	P $\rightarrow$ x 5	x $\rightarrow$ P 2	БП 32
P $\rightarrow$ x 5	x $\rightarrow$ P 1	F L 0	13	P $\rightarrow$ x 5	C/P			

В программу вставлен останов по адресу 19 для контроля  $x$ . Точка останова выбрана неплохо, но еще лучше было бы вставить его в начало вычисления функции, когда РХ содержит  $x$ . При вводе допущена незамеченная ошибка: вместо " $F X \geq 0$ " введена команда " $F X = 0$ ". Для отладки возьмем знакомую функцию  $y = \sin x$  и будем искать ее корень на отрезке  $[6, 7]$  с точностью 0.05 (этот процесс уже ранее подробно был рассмотрен).

Приступаем к отладке. Заносим 1.2345678 09 во все регистры (в том числе в P1, P2, P3, если теперь ошибемся в занесении данных, то это будет заметно), заносим значения 6, 7 и 0.05 в P1, P2 и P3 соответственно. Заносим в стек 1.2345678 09, и "не заметив", что переключатель "Р—ГРД—Г" стоит в положении "Г", запускаем программу. Сейчас отладку будем выполнять максимально грамотно.

Клавиши	Индикатор	Комментарии
С/П	ЕГГОГ	Получили ошибку,
F ПРГ	02 18 13 10	останов при вычислении логарифма,
F АВТ	- 20	который пытались вычислить от - 20.
F О	1.2186933-01	В РУ значение флев явно неправиль-
F О	1.2345678 -09	ное, а дальше стек не использовался.
P $\rightarrow$ x 0	1.2345678 -09	В счетчик еще ничего не занесли,
P $\rightarrow$ x 1	6.	здесь должно быть 7 (прав=7),

Клавиши	Индикатор	Комментарии
П → х 2	7.	а здесь должно быть 6,
П → х 3	5.	значение € правильное,
П → х 4	1.2345678	а <i>флев</i> почему-то значения не получило
П → х...	...	В остальных регистрах все в порядке.

После просмотра P1 и P2 стало ясно, что перепутаны значения *прав* и *лев* — типичная ошибка эксплуатации, в данном случае спровоцированная плохой организацией диалога: естественнее сначала задать левую границу отрезка, следовательно, она должна быть в регистре с меньшим номером. Вследствие этого и получили логарифм от отрицательного числа, и значение *флев*, разумеется, не с тем знаком. Но если после просмотра P1 и P2 сразу заняться исправлением, то это и будет грубая *ошибка в организации отладки*, которая израсходует наше время на лишние запуски.

Дальнейший просмотр регистров вскрывает еще одну ошибку: не получает значения *флев*. Посмотрев в программу мгновенно выясняем, что пропущена команда "х → П 4" (приемы вставки пропущенной команды рассмотрены в п. 3). Но и это еще не все. Вопреки сформулированному нами правилу мы не провели отдельное тестирование функции. Это сделано потому, что тестирование проводится неявно: результат мы увидим в переменной *флев* (подобные неформализуемые "маленькие хитрости" часто встречаются в программировании). В *флев* мы ничего не увидели, но то же значение имеется в стеке, правда, оно ошибочно, так как вычислялось не с тем аргументом. Но в подобных ситуациях всегда следует задать себе вопрос: *а может ли найденная ошибка объяснить полученный эффект?* И если ответ отрицательный, надо искать другую ошибку.

Итак, может ли перемена местами значений *прав* и *лев* объяснить ошибку ("ЕГГОГ")? Да. А может ли объяснить, почему в РУ такое значение? Нет! Значение синуса семи радиан должно быть гораздо больше. Итак: еще ошибка — неправильно вычисляется функция. Разбираясь с этой ошибкой рано или поздно обнаружим неправильное положение переключателя "Р—ГРД—Г". Таким образом, первый запуск дал *три ошибки*: две ошибки эксплуатации и одну ошибку программирования (пропуск команды).

Исправляем ошибки и делаем второй запуск. При останове на индикаторе высвечивается 0.5 (в характерном для ПМК виде). В РУ и P0 — число повторений цикла, равное 4.321928, в PZ и P4 — значение *флев*, равное -0.2794154, содержимое



прочих регистров сомнений не вызывает. Проверка первого шага цикла показывает, что  $(\text{прав}-\text{лев})/2$  действительно дает 0.5. Чтобы получить середину отрезка, нужно взять полусумму, а не полуразность. Значение *флев* правильное. При таком значении счетчика цикл будет выполняться 4 раза, отрезок сократится в 16 раз и будет иметь длину около 0.06, а нужно менее 0.05; имеем еще одну ошибку алгоритмизации — нужно округлять до большего, т.е. прибавить единицу.

Вносим исправления и делаем третий запуск. На первом останове получаем 6.5 — все правильно, проверять регистры нет оснований, хотя это и не повредит (правда, на предыдущих запусках уже все проверено). При следующем останове получим результат 6.25. Еще при одном останове — 6.125; здесь должна была работать другая ветвь, но она не сработала. В РУ — 6.25 — следы работы ветви да на предыдущем проходе цикла, в РZ — положительное число — произведение  $\Phi(x) \cdot \text{флев}$ , в остальных регистрах все правильно. Все данные правильны, следовательно, неправильна программа. И проверка организации ветвления дает последнюю ошибку — ошибку ввода команды "F X  $\geq$  0". Следующий запуск дает правильные результаты. Итоговый результат: 6.28125.

После останова полезно также проверить регистры. Например, в Р1 — 6.3125, в Р2 — 6.28125; разность, действительно, меньше  $\epsilon$ . Эта проверка могла бы показать ошибку неправильного вычисления числа циклов, если бы мы ее пропустили незамеченной при изучении числа 4.321928. Нажмем еще раз клавиши "В/П" и "С/П" и, подождав 10 с, остановим ПМК нажатием клавиши "С/П" и посмотрим содержимое переменной *флев*: там 0.001935... — действительно, значение функции от найденного корня достаточно близко к нулю. Впрочем, здесь следует иметь в виду, что у круто идущей вблизи корня функции "достаточно близким к нулю" значением может быть и довольно большое число — тут нужна математическая оценка.

Алгоритм отлажен: все его ветви и проверки проработали и с результатом "да" и с результатом "нет". Можно для надежности проверить его еще на одном тесте. Для поиска ошибок потребовалось всего четыре запуска. Вывод таков: можно делать сколько угодно ошибок, но следует уметь их быстро находить и устранять.

Заметим, что необходимо различать *место ошибки* и *место проявления ошибки*. Ошибочные данные проявили себя только в команде по адресу 09; незапомненное в команде с адресом 04 значение *флев* оказало бы влияние на ход программы только

в команде "F  $X \geq 0$ ". Ошибка в команде перехода проявила себя только на третьем проходе цикла, а неправильно вычисленное число повторений цикла вообще не мешало программе выполняться до самого конца. *Основа отладки — это переход от места проявления ошибки к месту самой ошибки*, которое может быть довольно далеко.

Следует руководствоваться следующим правилом: ищите блок, формирующий ошибочный результат.

За формирование значения *флев* отвечали первые четыре команды (из которых одна была пропущена) и команды вычисления функции. За число повторений цикла отвечали команды 03—12 и т.д. Именно они и используемые ими данные подлежат тщательному анализу при поиске ошибки.

Наша программа готова к эксплуатации. Заменяем останов по адресу 19 на команду "К НОП", вводим программу вычисления нужной функции, тестируем ее выполнением в пошаговом режиме или другим способом (можно использовать и косвенное тестирование — посмотреть значение функции в переменной *флев*). Теперь можно вводить данные и выполнять расчеты.

### 3. ВНЕСЕНИЕ ИСПРАВЛЕНИЙ

Все исправления, которые надо вносить в программу, когда ошибка найдена, бывают трех видов: *замена команд, удаление и вставка*. Замена команд проблем не вызывает. Удаление выполняется заменой команды командой "К НОП". Наибольшие сложности вызывают вставки, так как вставка требует сдвига всех команд, расположенных за местом вставки, с изменением их адресов, а значит, с большой вероятностью ошибок в командах перехода.

Одним из способов облегчения этой работы является *предварительная подготовка* к возможным вставкам: так же, как для облегчения отладки вводили команды "С/П", можно порекомендовать вставлять команды "К НОП" после каждых, скажем, 10 команд. Тогда каждая десятка команд имеет запас в две команды на возможное расширение, которое не затронет всю программу. Не замененные команды "К НОП" так и остаются в окончательном варианте программы — во многих задачах удлинение и замедление программы, вызванное наличием команд "К НОП", несущественно.

Но если вставленных команд "К НОП" нет или же их оказалось недостаточно, то приходится применять другой прием, хорошо известный программистам, некогда программировавшим в кодах ЭВМ (а ныне забытый за ненужностью для больших ЭВМ), под названием "заплата": ближайшие к месту вставки две команды (можно взять или две команды перед местом вставки, или две после него, или одну перед ним, вторую после него) заменяются командой "БП" с адресом любого свободного участка программной памяти. На этом участке размещаются замененные команды и та команда (те команды), которую нужно вставить, а затем ставится команда "БП" с адресом команды, следующей за замененными. Нежелательно заменять команды, на которые имеются переходы, так как тогда придется изменять адреса переходов. И, конечно, нельзя заменять одну из ячеек команды, занимающей две ячейки.

В рассмотренной ранее программе нужно поставить две "заплаты":

Адрес	Команды	Комментарий
01-03	П → x 2    ПП    40	Следующие за точкой вставки команды заменены переходом на произвольный адрес.
04-05	БП        60	
06-...	- ...	
09-10	2        F Lp	А здесь заменена одна команда перед местом вставки и одна после него.
11-12	БП        65	
13-...	П → x 1    ...	
...	...	Далее все без изменений до конца программы.
...	В/О	
...		
60	x → П 4	Первая заплатка: пропущенная команда, замененные команды, и переход к продолжению программы.
61-62	П → x 1        П → x 2	
63-64	БП        06	
65	÷	Вторая заплатка: замененная команда, вставляемые команды, вторая замененная команда, и переход к продолжению программы.
66-67	1    +	
68	x → П 0	
69-70	БП        13	

Если для второй "заплаты" взять команды 12-13, то пришлось бы заменить и адрес перехода в команде "F L0" по адресу 33.

"Заплаты" удлиняют и замедляют программу больше, чем выполнение команды "К НОП", так как каждая "заплата" — это четыре слова программной памяти и около 1 сек времени при каждом ее выполнении.

И, наконец, если программа нужна не для однократного использования или необходима ее предельная эффективность, то единственно возможный способ — это действительно сдвигать команды программы со всеми вытекающими отсюда последствиями.

#### 4. ДОКУМЕНТАЦИЯ ПРОГРАММЫ

Инженер, который использует ПМК постоянно для выполнения характерных для его специальности расчетов, или экономист, регулярно пользующийся ПМК (равно как и человек любой другой профессии), быстро обнаруживает, что ему часто приходится пользоваться программой, которую он написал когда-то давно, или которую написал его коллега, возможно, при этом слегка изменив ее. Вот здесь-то и оказывается, что для того, чтобы просто воспользоваться старой своей или чужой программой, нужно вспомнить или разобраться, как ею пользоваться (что и как вводить, что и в каком порядке выдается). Чтобы модифицировать ее, нужно понять, как она работает, а это по последовательности ее команд сделать довольно непросто. Поэтому человеку, который собирается профессионально использовать любую ЭВМ (в частности, микрокалькулятор), следует запомнить, что программы читаются не столько машинами, сколько людьми, в связи с чем программа должна быть написана так, чтобы ее удобно было читать и легко в ней разбираться, т.е. программа должна сопровождаться пояснительным текстом, описывающим, как она работает и как ею пользоваться. Совокупность материалов, описывающих программу называется *документацией*. Полная документация включает следующие материалы.

1. *Описание задачи*, решаемой данной программой.

2. *Область применимости программы*, т.е. класс исходных данных, на которые она рассчитана, и степень контроля за принадлежностью данных этой области.

3. *Описание метода*, которым решается задача, с указанием его математических характеристик (скорость, точность) и рекомендаций к применению. Должны быть приведены расчетные формулы. Желательны ссылки на литературу, где метод описан подробно.

4. *Инструкция по использованию программы*, которая должна включать описание правил ввода исходных данных (что, как и в каком порядке вводить), правил вывода результатов (что, как и в каком порядке выдается программой) и перечисление всех аварийных ситуаций ("останов по адресу ... означает ...").

5. *Алгоритм работы* программы, записанный на А-языке (наиболее предпочтительный вариант), в виде схемы или в каком-либо другом виде.

6. *Текст* программы с указанием кодов команд и адресов, желательны также комментарии к тексту.

7. *Тесты* с указанием значений исходных данных, получаемых результатов и времени выполнения каждого тестового примера.

8. *Возможные модификации* программы (если они предвидятся). Указываются команды, подлежащие изменению, для такой-модификации программы.

Пункты 1 и 2 необходимы всегда; они нужны для того, чтобы определить, что это за программа и применима ли она для конкретного случая. Пункт 3 позволяет правильно сделать выбор программ (если есть выбор из нескольких), но в сокращенном варианте документации он может и отсутствовать; программой можно пользоваться и не зная расчетных формул.

Пункт 4 необходим всегда. Он может быть описан по-разному, например: "Занести  $a$  в P1,  $b$  — в P3" или " $a, b, c$  задаются С/П-вводом (после ввода  $b$  программа работает около 10 с)". Этот пункт может включать и правила написания используемых подпрограмм, например: "С адреса 45 должна размещаться подпрограмма вычисления  $f(x, y)$ , которая берет аргументы из PA ( $x$ ) и PB ( $y$ ), а результат оставляет в PX".

Пункт 5 необходим, если предполагается в дальнейшем разбираться в программе, а не просто ее использовать. Если вы сами составляли программу, то не торопитесь выбрасывать ее схему или алгоритм на А-языке: он не займет много места, а пригодиться может. Схема может превосходить запись на А-языке в наглядности, но это проявляется только на больших программах, да и то не всегда. Но она требует много излишней чертежной работы (рамочки, стрелки). Как мы видели, программа на А-языке позволяет описать практически все нюансы использования ПМК, а строгие правила перевода с А-языка дают возможность почти однозначно получать программу в командах ПМК, поэтому А-язык предпочтительнее. Отдельную запись алгоритма удобно иметь даже при условии, что А-язык используется для комментирования текста программы.

Пункт 6 необходим всегда, но в справочниках он имеется в усеченном виде: из соображений экономии места не приводятся коды команд (их легко восстановить), нет комментариев

(не предполагается, что в программах будут разбираться), записаны команды в компактной форме, а не так, как удобно для ввода. В личной документации рекомендуется записывать программы по принятой в этой книге схеме: с таблицей распределения регистров, в столбец по одной команде в строке, со столбцом "Коды" и с комментариями на А-языке, а также, возможно, и с дополнительными комментариями.

Пункт 7 необходим всегда, причем должны присутствовать и тесты, контролирующие ошибочные состояния. Например:

"Тест 7:  $a=1$ ,  $b=16$ . Сообщение об ошибке "ЕГГОГ" по адресу 44 через 17 с".

Пункт 8 может отсутствовать. Пример текста в этом пункте: "При изменении коэффициента жесткости заменить команды 13...18 командами набора значения коэффициента".

В качестве примера рассмотрим документацию к программе решения уравнения методом половинного деления.

1. Программа *деление пополам* решает уравнение  $f(x) = 0$ , где  $f(x)$  непрерывная функция.

2. Исходные данные:  $a$ ,  $b$  — границы отрезка, на котором отыскивается корень;  $\epsilon$  — точность определения корня. Ограничения:

а)  $a > b$  (при невыполнении возникает сообщение "ЕГГОГ");

б)  $f(a)$  и  $f(b)$  должны иметь разные знаки. Программа это не проверяет, при невыполнении в качестве корня выдается значение  $b$ ,

в) на отрезке  $[a, b]$  должен быть только один корень, при невыполнении ищется один из корней, заранее не известно, какой.

3. Один из наиболее быстрых методов для функций общего вида. Скорость  $O(\log_2 \frac{(b-a)}{\epsilon})$ . Расчетные формулы здесь приводить не будем. Описание см. в (можно дать ссылку на любую книгу, в которой есть этот метод), стр. ... .

4. *Инструкция*. Подпрограмма вычисления функции  $f$  должна размещаться с адреса 40, аргумент задается в  $PX$ , результат выдается в  $PX$ , функция не должна менять значения в  $P0...P5$ . Начальный адрес программы — 00. Набрать  $a$ , "С/П", при останове на индикаторе высвечивается значение  $f(a)$  (значение можно использовать для тестирования подпрограммы вычисления функции), набрать  $b$ , "С/П", набрать  $\epsilon$ , "С/П", при останове результат высвечивается на индикаторе. Останов по адресу 12

через 4 с после ввода  $\epsilon$  — ошибка:  $a=b$ . Прочие остановы — из-за переполнения, связанного со значениями функции.

5. Алгоритм приводить не будем — примеры были ранее.

6. Текст программы:

Адрес	Код	Команда	А-язык	Комментарий
00	41	$x \rightarrow P1$	Ф (ввод лев)	P0 — счетчик
01	53	ПП		P1 — лев
02	...	...	...	...
и т.д.				

7. Тесты. Функция  $y = \sin x$ : с адреса 40 команды "F sin" и "B/O";  $a = 6$ ,  $b = 7$ ,  $\epsilon = 0.05$  (переключатель "P-ГРД-Г" в положение "P"). Результат — 6.28125. Время счета 56 с.

Та же функция,  $a = 7$ ,  $b = 6$ ,  $\epsilon = 0.05$ . Ошибка "ЕГГОГ". Время счета 4 с.

8. Для данной программы модификации не предусмотрены.

## 5. ЭКСПЛУАТАЦИЯ ПРОГРАММЫ

Программа, прошедшая тестирование, и давшая правильные результаты на всех тестах *считается отлаженной*. Однако в программе все же могут остаться ошибки, которые не были обнаружены при тестировании. Вероятность наличия ошибки в отлаженной программе не поддается никакой теоретической оценке и зависит от квалификации программиста, тестировавшего программу. Даже тестирование всех ветвей и всех проверок оставляет возможность для такой последовательности работы ветвей, которая не была проверена на тестах, и дает ошибочный результат. Поэтому во всех случаях выданные программой результаты подлежат контролю любыми доступными способами: проверкой на здравый смысл, просчетом другим методом и т.д. Этот вопрос совершенно не формализуем. Один из примеров был рассмотрен: в нем правильность вычисления корня проверяли путем вычисления функции от полученного значения.

И разумеется *невозможна абсолютная защита от ошибок эксплуатации*, так как ничто не может помешать просто неправильно ввести данные. Поэтому особо важные расчеты целесообразно выполнять дважды и доверять результату только при совпадении их результатов. При этом необходимо повторно занести исходные данные (по одним и тем же данным ПМК, естественно, даст один и тот же результат).

На МК-52 имеется возможность записать программу в долговременную память, которая хранит программу и при выключении ПМК. При повторном использовании такой программы ее достаточно переписать в программную память, повторного ее тестирования (если исправно работает ПМК) не требуется. При повторном использовании программ, которые вводят с клавиатуры, необходимо полное их тестирование на всех тестах. Тесты обязательно приводятся для всех программ, имеющихся в справочниках. Необходимо их иметь и в своей документации. Единственное отличие в тестировании повторно используемых программ заключается в том, что все ошибки в таких программах — это *ошибки ввода*, и при несовпадении результатов с тестовыми нужно только тщательно проверить текст программы в памяти ПМК.

## Глава 6. ПОГРЕШНОСТИ РАСЧЕТОВ НА МИКРОКАЛЬКУЛЯТОРАХ

### 1. ВЛИЯНИЕ ПОГРЕШНОСТЕЙ НА РЕЗУЛЬТАТЫ

В реальных инженерных расчетах точности в 3–4 знака обычно оказывается достаточно, и на первый взгляд восемь разрядов микрокалькулятора способны обеспечить нужную точность с большим избытком. Однако это впечатление ошибочно. Характерной особенностью расчетов на всех ЭВМ, в том числе и на ПМК, является значительное количество арифметических действий, которые выполняются для получения результата, и маленькая погрешность на каждом действии может в итоге вырасти в большую. Существенную роль здесь играют также *особенности машинной арифметики*, которые приводят к иным правилам накопления погрешностей. Тот факт, что многочисленные расчеты на ПМК (в частности, приведенные в этой книге примеры) не давали оснований для тревоги относительно точности результата, ни о чем не говорит: как будет показано, на взгляд невозможно определить, какой результат точен, а какой — нет, и не исключено, что где-то вы приняли за правильный совершенно неверный результат.

Особенности машинной арифметики, сказывающиеся на точности результатов, это

*работа в ограниченном диапазоне,  
работа с ограниченной точностью.*

Диапазон представимых на ПМК чисел — это  $1 \cdot 10^{-99}$  —



$9.99999999 \cdot 10^{99}$ . Большие по абсолютному значению числа дают переполнение (сообщение "ЕГГОГ"), меньшие — считаются нулем (хотя лучше бы тоже выдавалось сообщение "ЕГГОГ"), как это делается на некоторых ЭВМ]. Точность — восемь десятичных разрядов. На больших ЭВМ эти числовые параметры другие, но суть дела от этого не меняется и проблемы остаются те же. Первым следствием из этих ограничений является невыполнение свойств ассоциативности и дистрибутивности сложения и умножения.

Например, вследствие ассоциативности умножения  $a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c)$ . Но при  $a = 10^{50}$ ,  $b = 10^{60}$ ,  $c = 10^{-40}$  вычисление  $(a \cdot b) \cdot c$  дает ошибку "ЕГГОГ", а вычисление  $a \cdot (b \cdot c)$  дает правильный результат  $10^{70}$ . Если же  $a = 10^{-50}$ ,  $b = 10^{-60}$ ,  $c = 10^{90}$ , то первая формула дает ноль, а вторая — правильный результат  $10^{-20}$ , при этом ПМК никак не предупреждает о том, что очень малое число заменено нулем.

Согласно закону дистрибутивности  $(a + b) \cdot c = a \cdot c + b \cdot c$ . Но при  $a = 1000049$ ,  $b = -1000000$ ,  $c = 2000049$  в результате получим 98002401 и 98000000, т.е. из восьми разрядов половина не точна.

Рассмотрим, как эти нарушения математических законов сказываются на результатах расчетов. Пусть нужно вычислить

$$\sum_{i=1}^{10} x_i^3, \text{ где значения } x_i \text{ таковы: } 1001, 203.298, 17.25, 7, 43.095, \\ i = 1$$

12.03, 3.41, 5.2, 21.5, 96. Результат равен 1012387100. Если же теперь взять эти же числа в обратном порядке, т.е. считать, что  $x_1 = 96$ ,  $x_2 = 21.5$  и т.д., то получим 1012387400 (отличие в три единицы последнего разряда). Меняя порядок суммирования можно получить и результаты в промежутке между приведенными двумя. Взяв еще три числа: опять числа 7, 43.095, 12.03, на сумме из 13 кубов чисел можно получить различие в четыре единицы последнего разряда. Но добавление 0.1 к любому из чисел устраняет эффект: различие получается только в одну единицу последнего разряда. Если же нам нужно вычислить

$$\sin\left(\sum_{i=1}^{10} x_i^3\right), \text{ то будем получать совершенно разные результаты:}$$

0.224... и -0.979...

Теперь проведем вычисление интеграла для простейшей функции  $y = x$ , применив формулу средних прямоугольников, которая для этой функции должна дать точный результат. Алго-

ритм и программа легко получаются из алгоритма и программы, рассмотренных в п. 2 гл. 3: перед циклом  $x$  должно иметь значение  $a + h/2$  (впрочем, можно использовать и программу вычисления интеграла методом левых прямоугольников на отрезке  $[a + h/2, b + h/2]$  с тем же числом разбиений  $M$ ). Получим следующие результаты.

$a$	$b$	Точное значение	$M=9$	$M=18$	$M=27$	$M=180$
100	103	304.5	304.49997	304.5007	304.49998	304.50095
100000	100003	300004.5	300004.47	300004.58	300004.48	300005.43

Таблица показывает, что погрешность с увеличением числа разбиений не убывает согласно теории, а ведет себя достаточно непредсказуемым образом и 10-минутный счет при  $M=180$  дает гораздо худшие результаты, чем полуминутный при  $M=9$ . При попытке анализа регистров с целью выяснения причины таких результатов для последнего варианта исходных данных обнаружим, что  $x = 100003.61$ , т.е. гораздо больше (на несколько шагов), чем нужно.

Разумеется, эти примеры специально подобраны. Но очевидно, что на взгляд нельзя отличить, какой из приведенных результатов более точен, нельзя понять, скольким знакам результата можно верить, а также, чем отличаются те данные, для которых погрешность велика, от тех, которые дадут "нормальные" результаты.

## 2. КЛАССИФИКАЦИЯ И ОЦЕНКА ПОГРЕШНОСТЕЙ

По своему происхождению погрешности подразделяют на три группы: *в исходных данных, метода и вычислительные*. Как правило, исходные данные не точны, следовательно, погрешность появляется в расчете уже с вводом данных — эти погрешности обычно известны, так как определяются источником получения данных. Например, если длину мерили линейкой с миллиметровыми делениями, то погрешность была не более 0.5 мм.

Погрешности метода связаны с тем, что все численные методы основаны на каких-то заменах: в методе прямоугольников при вычислении интеграла функция заменяется отрезками прямых, в методе половинного деления при решении уравнения корень считается серединой отрезка и т.д. Разность между

например, истинным значением интеграла и площадью фигуры, заштрихованной на рис.7, а — это и есть погрешность метода.

Наконец, все вычисления идут с какими-то погрешностями, которые называются вычислительными. Взаимодействие этих погрешностей, которое может оказаться чрезвычайно сложным, и дает итоговую погрешность результата. При этом главный вклад в итоговую погрешность может дать любая из составляющих.

Если обозначить через  $X$  точное значение, а через  $x$  то значение, которое хранится в ЭВМ, то разность

$$\Delta x = X - x$$

называется абсолютной погрешностью, а отношение

$$\delta_x = \Delta x / x$$

— относительной погрешностью. В разных ситуациях нужны оценки разных видов погрешности. Правила разрастания погрешностей для ручного счета на ЭВМ различаются. Но во всех ситуациях оценивается худший случай, которого реально может и не быть. Для оценки погрешностей на ПМК (или другой ЭВМ) введем понятие "порядок числа  $x$ ", который будем обозначать  $p(x)$ :

$$p(x) = [\lg x],$$

где  $[x]$  — это "целая часть".

Можно сказать, что  $p(x)$  — это тот порядок, который мы видим на индикаторе ПМК (точнее, увидели бы, если бы числа всегда представлялись в нормализованной форме). Поскольку 9-й разряд представления числа округляется, то тем вносится погрешность  $\Delta x \leq 0.00000005 \cdot 10^{p(x)}$ , а относительная погрешность в худшем случае имеет оценку

$$\delta x = \frac{5 \cdot 10^{-8} 10^{p(x)}}{m \cdot 10^{p(x)}} \leq 0.5 \cdot 10^{-7},$$

где  $m$  — мантисса числа,  $1 \leq m < 10$ ).

А абсолютная погрешность связана на ПМК с другой относительной формулой:

$$\Delta x \leq \delta x \cdot 10^{p(x)}.$$

При сложении-вычитании чисел в математике погрешности складываются:

$$\Delta(x + y) = ((X + Y) - (x + y)) = ((X - x) + (Y - y)) = \Delta x + \Delta y.$$

Но на ЭВМ это не так. Сначала число с меньшим порядком приобретает погрешность того же порядка, что и число с большим порядком, а затем погрешности складываются:

$$\begin{aligned} 100003 \frac{1}{3} &= 100003.33 + 0.00333 \dots \\ + \quad 1/3 &= 0.33333333 + 0.0000000033 \dots \end{aligned}$$

---


$$100003 \frac{2}{3} = 100003.66 + 0.00666 \dots$$

Выравнивание порядков при сложении выбрасывает за разрядную сетку часть разрядов числа с меньшим порядком (они подчеркнуты) и фактически при сложении-вычитании погрешность результата может достичь удвоенной погрешности большего по порядку из чисел.

При умножении в математике складываются относительные погрешности, что в целом имеет место и на ЭВМ. Но ограниченная длина разрядной сетки не дает выполнить и это правило: любое число с порядком больше восьми (8 — длина разрядной сетки ПМК) приобретает погрешность  $\Delta x \leq 0.5 \cdot 10^{p(x)-7}$ .

Вот эти-то вносимые машинной арифметикой погрешности и вызвали те необычные эффекты, которые наблюдались в простейших примерах. Проследим, как шло накопление погрешностей при вычислении

$$\sum_{i=1}^{10} x_i^3.$$

$x$	$x_i^3$	$\Sigma x_i^3$ на ПМК	Точная $\Sigma x_i^3$	Погрешность
1001	1003003009.0	1003003000	1003003009.0	9
203.298	8402321.7	1011405300	1011405331.7	31.7
17.25	5132.9	1011410400	1011410464.6	64.6
7	343	1011410700	1011410807.6	107.6
43.095	80035.1	1011490700	1011490842.7	142.7
12.03	1741.0	1011492400	1011492583.7	183.7
3.41	39.6	1011492400	1011492623.3	223.3
5.2	140.7	1011492500	1011492764.0	264.0
21.5	9938.4	1011502400	1011502702.4	302.4
96.0	84736.0	1012387100	1012387438.4	338.4

Итак, куб первого же числа имел порядок более 8, и приобрел погрешность, а далее значение одного из слагаемых все время "заставляло" второе слагаемое приобрести погрешность того же порядка, а подобранность чисел заключалась в том, что все приобретаемые погрешности были одного знака. При суммировании в обратном порядке промежуточные результаты имеют

малый порядок и поэтому не приобретают погрешности вообще и только последнее слагаемое дает погрешность вследствие своего большого порядка:

96.0	884736.0	884736	884736.0	0
21.5	9938.4	894674.38	884674.38	0
...	...	...	...	...
203298	8402321.7	9384428.4	9384428.4	0.0 ...
1001	100300309.0	1012387400	1012387437.4	37.4 ...

Как показывает первая таблица (см. с. 188) при суммировании кубов мы имели еще не худший случай: могли бы и на каждом слагаемом терять почти 50 единиц (точнее, на каждом слагаемом можно было потерять 44.444... единицы — так работает алгоритм округления на ПМК). Попробуем оценить погрешности для других примеров. При вычислении выражения  $a \cdot c + b \cdot c$  погрешность произведения  $(a \cdot c)$  может достигнуть  $5 \cdot 10^{p(a \cdot c) - 8} = 5 \cdot 10^4$ , погрешность  $(b \cdot c)$  имеет тот же порядок, значит, в сумме могли получить погрешность до  $10^5$  (т.е. опять же наблюдали не худший случай). Вообще говоря, оценка погрешностей представляет собой чрезвычайно сложную задачу, так как оценка по максимуму на каждом шаге дает завышенный результат, а вопрос, на сколько ее можно снизить, требует специальных исследований.

Оценим погрешность, которую можно ожидать при вычислении интеграла. Поскольку шаг всегда имеет меньший порядок, чем  $a$  и  $b$ , то погрешность будет определяться максимальным из порядков  $a$  и  $b$ . На каждом из  $M$  шагов (при добавлении  $h$  к  $x$ )  $x$  может получать погрешность  $\Delta = 5 \cdot 10^{\max\{p(a), p(b)\} - 8}$  дополнительно к той, которая у него была. В итоге вместо

вычисления суммы  $\sum_{i=1}^M f(x - \frac{h}{2} + ih)$  реально вычисляется

сумма  $\sum_{i=1}^M f(x - \frac{h}{2} + ih + i\Delta)$ . В данном случае из-за простоты

функции само вычисление функции не привносит вычислительной погрешности и

$$h \sum_{i=1}^M f(x - \frac{h}{2} + ih + i\Delta) = h \sum_{i=1}^M f(x - \frac{h}{2} + ih) +$$

$$+ h \sum_{i=1}^M i\Delta = h \sum_{i=1}^M f(x - \frac{h}{2} + ih) + \frac{M(M+1)}{2} \Delta h.$$

Так как для  $a = 100$ ,  $b = 103$  значение  $\max \{p(a), p(b)\} = 2$ , то для  $M = 180$  можно ожидать погрешность до  $10^{-3}$ , а при  $M = 9$  — погрешность до  $7 \cdot 10^{-5}$ . Однако при малых  $M$  начинает играть основную роль погрешность метода (это только для функции  $y = x$  погрешность метода равна нулю), имеющая порядок  $O((\frac{b-a}{M})^2)$ . Как видно из этого примера, при расче-

тах на ЭВМ следует грамотно выбирать шаг разбиения, поскольку принцип "чем мельче — тем лучше" приводит к затратам машинного времени, но не увеличивает точности счета.

### 3. МЕТОДЫ СНИЖЕНИЯ ПОГРЕШНОСТЕЙ

Сформулировать общие правила борьбы с погрешностями практически невозможно, однако можно указать несколько сравнительно простых правил, позволяющих снижать погрешности.

**Правило 1.** *Максимально точно задавайте исходные данные.* Экономия на вводе значащих цифр исходных данных вводит дополнительную погрешность, которая, пройдя через расчет, может достичь значительного значения. Например, объем шара радиусом 20 равен 33510.318, но если вместо  $\pi$  взять 3.14, то получим результат 33493.331, имеющий абсолютную погрешность около 17 единиц и относительную погрешность около 0.05 %. В физических и инженерных расчетах часто встречаются различные константы, известные с большой точностью, такие константы следует вводить в ПМК с максимально допускаемой ПМК точностью, так как в сложных расчетах взаимодействие погрешностей может дать неожиданные результаты.

Наиболее характерен такой пример. Пусть вычисление интеграла методом левых прямоугольников организовано не с помощью цикла раз, а с помощью цикла пока:

инт:=0;  $x := a$ ;

пока  $x < b$  повторять инт:=инт +  $f(x)$ ;  $x := x + h$  кнвт;

инт:=инт ·  $h$ ;

где исходными данными являются  $a$ ,  $b$  и  $h$ . Пусть  $a = 0$ ,  $b = 1$ , а  $h = 1/3$ . Если задать  $1/3$  в виде  $3.3333333 \cdot 10^{-1}$  (максимально возможная точность), то цикл будет выполняться в 3 раза, но если  $1/3$  задать в виде 0.3333333, то после третьего выполнения тела цикл  $x$  получит значение 0.9999999, что меньше 1,

и цикл выполнится четвертый раз. При точном значении интеграла на отрезке  $[1, 2]$  для функции  $y = x$  равном 1.5, при  $h = 3.3333333 \cdot 10^{-1}$  получим результат 1.333333, но погрешность 0.00000003 в задании  $h$  (т.е. при  $h = 0.3333333$ ) уже даст 1.999999.

**Правило 2.** *Промежуточные результаты должны быть в диапазоне максимальной точности.* Вычисления должны быть организованы так, чтобы промежуточные результаты имели минимальный по абсолютному значению порядок, а также минимальное количество цифр. Следует, например, чередовать при умножении большие и малые числа, чтобы промежуточные результаты были возможно ближе к середине диапазона. При сложении нужно сначала складывать числа с меньшими порядками и вообще избегать суммирования чисел с резко отличающимися порядками, в особенности сложения, при котором результат имеет порядок, существенно меньший, чем слагаемые (иными словами, вычитания близких чисел), так как абсолютная погрешность при этом не уменьшается, а относительная погрешность резко возрастает из-за малого порядка результата.

**Правило 3.** *Правильно выбирайте масштабные множители (единицы измерения).* Во многих случаях удержать промежуточные результаты в диапазоне максимальной точности помогает масштабирование, которое в физических задачах фактически является переходом к другим единицам измерения. Например, при вычислении  $\sqrt[4]{a^2 + b^2}$  для  $a$  и  $b$ , имеющих значения порядка  $10^{-60}$ , должен получиться результат порядка  $10^{-30}$ , но вследствие того, что  $a^2 \approx 10^{-120}$  и  $b^2 \approx 10^{-120}$ , получим ноль (это явление называется *антипереполнением*, или *исчезновением*). Но если увеличить порядок на 20 единиц, т.е. при вводе задавать не число  $a$ , а число  $a \cdot 10^{20}$ , то вычисления дадут результат, который в  $10^{10}$  раз больше, чем нужный нам (или же результат получается в других единицах, например в миллиметрах, а не в десятках тысяч километров). Если в формуле содержится несколько слагаемых, например,

$$\sqrt[6]{x^3 + y^3} + C\sqrt{a^2 + b^2},$$

где  $p(x)$ ,  $p(y) \approx 40$ ,  $p(a)$ ,  $p(b) \approx -50$ ,  $p(C) = 90$ , то для  $x$ ,  $y$  нужно взять масштаб  $10^{-10}$ , для  $a$  и  $b$  — масштаб  $10^{10}$ , а коэффициент  $C$  следует заменить на  $C \cdot 10^{-20}$ , после чего получим результаты в  $10^{10}$  раз меньше истинных. В подобных формулах всегда существует "выравнивающий" коэффициент  $C$ , который делает примерно одинаковыми порядки у слагаемых. Но если этот коэффициент таков, что порядки у слагаемых различаются

более чем на восемь единиц, то этот расчет на ПМК не выполним и нужна ЭВМ с более точной арифметикой.

**Правило 4.** *Используйте устойчивые к погрешностям алгоритмы.* Для получения более точных результатов следует использовать алгоритмы, учитывающие особенности машинной арифметики. В общем случае разработка таких алгоритмов весьма сложна, но иногда помогают простейшие методы. Например, при вычислении интеграла от периодической функции сдвиг отрезка интегрирования на период не должен изменять результат. Но выбор отрезка, максимально близкого к нулевой точке, уменьшит вычислительную погрешность при добавлении  $h$  к  $x$ . Например, площадь любой дуги синусоиды одинакова, но вычисление интеграла по формуле средних прямоугольников от функции  $y = \sin x$  с разбиением отрезка на 20 частей даст на отрезке  $[0, \pi]$  результат 2.0020581 при точном значении 2, а на отрезке  $[1000\pi, 1001\pi]$  — результат с погрешностью в 1,5 раза больше: 2.00333.

Как правило, алгоритмы, компенсирующие особенности машинной арифметики, требуют больше времени на свое выполнение (которое при работе на ПМК иногда можно заменить ручной работой). Например, алгоритм суммирования кубов чисел можно дополнить частью, упорядочивающей исходный массив по возрастанию чисел; время, потраченное на упорядочение является платой за точность.

При разработке таких алгоритмов следует стремиться к тому, чтобы каждое значение получалось из исходных данных за минимальное число операций, — тогда погрешности негде накапливаться. Для иллюстрации рассмотрим алгоритм вычисления интеграла, использующий такой прием:

**меняя  $k$  повторять  $M$  раз**

**инт:=инт +  $f(b - k \cdot h)$**

**кпвт**

Понятно, что такой цикл будет проходить медленнее, так как на каждом шаге для вычисления аргумента функции выполняются две операции — умножение и вычитание, вместо одной операции — последовательного добавления  $h$  (это и есть плата за точность). Но теперь аргумент функции получается из  $b$  и  $h$  за две операции, в то время как в ранее рассмотренных алгоритмах последние значения  $x$  получались путем  $M$  сложений. Возьмем отрезок  $[1000000, 1000003]$ ,  $M = 9$  (т.е.  $h = 1/3$ ) и проследим накопление погрешностей при разных способах вычисления аргумента  $x$ .



Точное значение	Вычисление $x := x + h$		Вычисление $x := b - k \cdot h$	
	результат	погрешность	результат	погрешность
1000000	1000000		1000000	0
1000000.333...	1000000.3	- 0.0333...	1000000.3	- 0.0333...
1000000.666...	1000000.6	- 0.0666...	1000000.7	0.0333...
1000001	1000000.9	- 0.0999...	1000001.0	0
1000001.333...	1000001.2	- 0.1333...	1000001.3	- 0.0333...
1000001.666...	1000001.5	- 0.1666...	1000001.6	0.0333...
1000002	1000001.8	- 0.2	1000002	0
...	...		...	

При таком различии в порядках между  $a$ ,  $b$  и  $h$  погрешность накапливается очень быстро. Но при компенсирующем алгоритме погрешность колеблется вокруг нуля и не увеличивается.

**Правило 5.** *Получаемая погрешность и достижимая точность должны оцениваться.* Приемлема или неприемлема получающаяся погрешность, зависит от конкретной ситуации. Нельзя сказать, много или мало те 17 единиц абсолютной и те 0.05 % относительной погрешности, которые получались при вычислении объема шара. Ответ зависит от того, что будет дальше делаться с результатом; возможно, что такая точность вполне приемлема. Но в любом случае, прежде чем применить численный метод, необходимо приложить его теоретические оценки к конкретному случаю, выяснить, какая получается погрешность и приемлема ли она для конкретного расчета.

Пусть, например, надо вычислить интеграл на  $[a, b]$  от функции, для которой  $f(a) = 10$ ,  $f(b) = 20$ ,  $\max_{[a, b]} f'(x) = 2$ , причем  $a$  и  $b$  имеют погрешности 0.0001. Вследствие неточного задания границ интервала сразу же получаем погрешность

$$\Delta_1 = \left| \int_{a + \Delta a}^{b + \Delta b} f(x) dx \right| - \left| \int_a^b f(x) dx \right| \leq \left| \int_a^{a + \Delta a} f(x) dx \right| + \left| \int_b^{b + \Delta b} f(x) dx \right| \approx f(a)\Delta a + f(b)\Delta b = 3 \cdot 10^{-3},$$

которая уменьшена быть не может. Погрешность метода равна  $\frac{\max f'(x)}{2} \cdot \left( \frac{b - a}{M} \right)^2$ . Достаточно выбрать  $M$  такое, чтобы

эта погрешность была на порядок меньше, чем  $\Delta_1$ , и уже не чувствовалась, а из-за наличия  $\Delta_1$  с большей точностью считать нет

смысла. Отсюда  $M$  должно быть таким, чтобы шаг  $(\frac{b-a}{M}) \leq 0.01$ .

Вследствие вычислительной погрешности в определении  $x$  (при условии работы простейшим методом  $x := x + h$ ) при вычислении функции получим погрешность порядка  $M/2 \times (b-a) \max f'(x) \times 5 \times 10^{\max \{p(a), p(b)\} - 8}$  (следовало бы учесть и вычислительную погрешность при работе алгоритма вычисления функции, но этим пренебрежем, поскольку в данном примере конкретная функция не приведена). Если эта погрешность получается не более чем  $10^{-3}$ , то итоговая погрешность определяется неточностью исходных данных.

Пусть  $(b-a) = 10$ ,  $p(a) = p(b) = 2$ . Тогда, чтобы погрешность метода не играла роли, нужно иметь  $M = 1000$ , но в этом случае вычислительная погрешность будет  $5 \cdot 10^{-2}$ . Итак, точность  $10^{-3}$  в данном примере недостижима. Максимальная точность, которую удастся получить, равна  $10^{-2}$ , что достигается при  $M \approx 125$ . Вот это значение  $M$  и следует брать для вычисления интеграла в данных условиях. А если такая точность неприемлема, то нужно искать другой способ получения результата.

Грубую прикидку погрешности можно сделать и так: считаем интеграл 2 раза при разных значениях  $M$  и берем столько цифр в результате, сколько их совпадает в обоих результатах; эти цифры заведомо верны. В нашем примере, рассмотренном в п. 1, гл. 6 получили следующие результаты: 304.49997, 304.5007. 304.49998, 304.50095 — так что за ответ  $304.5 \pm 0.01$  можно ручаться.

В заключение отметим, что в инженерных расчетах точность в три-четыре цифры вполне достаточна, поэтому более четырех (от силы пяти) цифр с индикатора выписывать не надо, а эти цифры чаще всего точные (имеются в виду *значащие цифры*, а не цифры после десятичной точки!).

## ЗАКЛЮЧЕНИЕ

Наш курс программирования закончен. В нем освещены те вопросы и проблемы, которые встают перед программистом, работающим на любом языке и на любой ЭВМ. Через несколько лет в наш быт и производственную практику войдут персональные компьютеры. Программируемые микрокалькуляторы, видимо, тоже не исчезнут совсем; найдется и тот класс задач, которые выгоднее будет решать на ПМК. Можно с уверенностью сказать, что тот, кто освоил этот курс, сделал свой первый шаг

к компьютерной грамотности. При переходе на более современные ЭВМ и более современные языки программирования, чем команды ПМК, он обнаружит, что многое из того, что познано применительно к ПМК, ему необходимо и в работе с ЭВМ, и в некоторых аспектах своей производственной деятельности, непосредственно с машинами не связанной.

## УКАЗАНИЯ К РЕШЕНИЮ ЗАДАЧ

### Гл. 1, п. 1

Единственный правильный вариант

$$4abc / + \times x \sin 42f - / \ln -$$

### Гл. 2, п. 2

1. Простейший вариант:

алг труба;

числ  $d_1$ ,  $d_2$ ,  $p$ , сечение, масса;

ввод  $d_1$ ,  $d_2$ ,  $p$ ;

сечение:  $= \pi \cdot d_1^2 / 4$ ; масса:  $= \pi \cdot (d_2^2 - d_1^2) / (4 \cdot p)$ ;

вывод сечение, масса;

кон

С применением более сложных приемов можно написать оптимальный алгоритм (здесь, как и далее, приведено только его тело):

числ  $\pi 4$ ;

вывод ( { масса = } ( ( - вывод ( { сечение = } ( ввод {  $d_1$  }  $^2 \cdot (\pi / 4 \rightarrow \pi 4)$  )  
ввод {  $d_2$  }  $^2 \cdot \pi 4$  ) · ввод {  $p$  } )

Программа по алгоритмам пишется однозначно.

2. Оптимальный вариант:

вывод ( ( ввод { часы } · 60 + ввод мин ) · 60 + ввод сек )

При программировании не забудьте команду "В  $\uparrow$ ":

В  $\uparrow$  6 0  $\times$  С/П + 6 0  $\times$  С/П + С/П

### Гл. 3, п. 1

#### 1. Близкий к оптимальному вариант

числ  $D, H, h, dH, Ddh, n$ ;

(ввод  $D$ )<sup>2</sup> •  $\pi/4 \rightarrow n$ ;

$(D - \text{ввод } \{d\}) / \text{ввод } h - Ddh$ ;

если  $(\text{ввод } H - \text{ввод } H_0 \rightarrow dH) < 0$

то вывод  $((1 - (1 - (dH/Ddh)^3)) \cdot D^2 / 3 \cdot Ddh + n \cdot H)$

иначе вывод  $(n \cdot H_0)$

все

Обратите внимание, что сложное выражение после то приведено к виду, при котором для его вычисления стека хватает. Отметим также, что возведение в куб удобнее всего сделать командами:  $V \uparrow, F x^2, \times$ .

2. В алгоритме для запоминания результатов взяты те же переменные, что и для данных. Это сделано в чисто иллюстративных целях и никакой экономии не дает.

числ ЧЧ, ММ, Ч, М;

ввод ЧЧ, ММ;

$Ч := Ч + \text{ввод } \{Ч\}$ ;

если  $(М := ММ + \text{ввод } \{М\}) \geq 60$  то  $М := М - 60$ ;  $Ч := Ч + 1$  все;

если  $Ч \geq 24$  то вывод  $\{\text{сутки} = \}$  1,  $Ч - 24$ , М

иначе вывод  $\{\text{сутки} = \}$  0, Ч, М все

Оптимизация за счет изменения порядка ввода данных или их вывода нежелательна, так как программой станет неудобно пользоваться.

### Гл. 3, п. 2

#### 1. Числ $x, H$ ;

ввод  $\{a\} x$ ;

повторять  $(\text{ввод } \{b\} - x) / \text{ввод } H + 1$  раз

вывод  $x, f(x), g(x)$ ;  $x := x + H$ ;

кпвт

Для программирования взять конкретные функции  $f$  и  $g$ .

2. Сложность в составлении этого алгоритма заключается

в том, что слагаемые имеют разные коэффициенты. Поэтому  $f(a)$  и  $f(b)$  следует обработать отдельно, а на одном шаге цикла обрабатывать сразу два слагаемых.

числ  $a, b, H, N, c; x;$

ввод  $a, b, N; H := (b - a)/N; c := f(a) + f(b); x := a + H;$

повторять  $N/2$  раз

$c := c + 4 \cdot f(x) + 2 \cdot f(x + H); x := x + 2 \cdot H$

кпвт;

вывод  $c \cdot H - 3;$

Алгоритм написан без оптимизации. При программировании удобно для  $c$  отвести РХ.

### Гл. 3, п. 3

1. Типичной ошибкой при программировании вычисления рядов является программирование вычисления каждого члена в отдельности: в данном случае для этого потребуется использовать операцию " $F x^y$ " и цикл вычисления факториала, что требует больших затрат времени и множества команд. Здесь следует использовать рекуррентные соотношения

$$c_{i+1} = c_i(2i + 1)/(2i + 3)x^2/(2i + 2)(2i + 3), c_1 = 1,$$

где  $c_i$  —  $i$ -й член ряда. После такого преобразования расчетных формул получаем алгоритм:

числ  $x, \epsilon, shi, i$ , член,  $x2;$

$(shi := \text{член} := \text{ввод } x)^2 \rightarrow x2; \text{ ввод } \epsilon; i := 1;$

пока  $|\text{член}| \geq \epsilon$  повторять

$(\text{член} \cdot i \cdot x2 / (i + 1) / (i + 2)^2 \rightarrow \text{член}) + shi \rightarrow shi;$

$i := i + 2;$

кпвт;

вывод  $shi$

2. В этой задаче удобнее применить цикл до:

числ  $a, H$ ; числ  $b$ ;

ввод  $a, H; b := a;$

повторять  $b := b + H$  до  $f(b) < 0$ :

(или повторять до  $f(b := b + H) < 0$ ;

вывод  $b$ ;

### Гл. 3, п. 4

1. Один из вариантов — с использованием адресации с авто-уменьшением (переменную  $c$  удобно разместить в  $PX$ ):

числ  $a [M:1]$ ,  $c$ ; косв —  $k$  числ  $M$ ;  
 <ввод можно организовать как угодно>  
 $k := \text{адрес } a [0]$ ;  $c := 0$ ;  
 повторять  $M/2$  раз  $c := c + [ \neg k ] \cdot [ \neg k ]$  кпвт;  
 вывод  $c$ ;

2. Если сначала вычислить математическое ожидание, а затем дисперсию, то получим два последовательно работающих цикла, что (и это легко проверить на ПМК) не самый лучший вариант как по скорости, так и размеру программы. Для получения оптимальной программы сначала следует преобразовать формулы (довольно распространенный прием — преобразование формул позволяет получить более качественный алгоритм):

$$\begin{aligned} \sum_{i=1}^N (a_i - M)^2 &= \sum_{i=1}^N a_i^2 - 2 \sum_{i=1}^N (a_i \cdot M) + \sum_{i=1}^N M^2 = \\ &= \sum_{i=1}^N a_i^2 - 2 \cdot NM \cdot M + N \cdot M^2. \end{aligned}$$

Теперь можно параллельно вычислить  $\sum_{i=1}^N a_i$  и  $\sum_{i=1}^N a_i^2$ , после чего получаем результаты.

числ  $a [N:1]$ ,  $N, M, D$ ; косв —  $k \ k2$ ;  
 <ввод любым способом>  
 $k := k2$ ; адрес  $a [0]$ ;  $M := D := 0$ ;  
 повторять  $M$  раз  
 $M := M + [ \neg k ]$ ;  $D := D + [ \neg k2 ]^2$ ;  
 кпвт;  
 вывод  $\left\{ \text{матожидание} = \right\} M/N, \left\{ \text{дисперсия} = \right\} (D - M^2)/(N - 1)$ ;

### Гл. 3, п. 5.

1. Заметим, что если в таблице имеется несколько максимальных элементов, то безразлично, значение какого из них брать:

числ  $a[M:1]$ ,  $M$ , макс; косв —  $k$ ; числ эл;

< ввод организуем любым способом >

макс:= $a[1]$ ;  $k$ :=адрес  $a[1]$ ;

повторять  $M-1$  раз

если (эл:=[ $\neg k$ ]) > макс то макс:=эл все

кпвт;

вывод макс;

2. В этой задаче при наличии нескольких максимальных элементов можно получить номер любого из них. Приведенный алгоритм дает номер первого из них. Подумайте, как его изменить, чтобы он давал номер последнего.

числ  $a[1:M]$ ,  $M$ , макс, ном; косв —  $k$ ; числ эл;

Рввод  $a$ ; ввод  $M$ ;

макс:= $a[M]$ ;  $k$ :=адрес  $a[M+1]$ ;

меняя  $M$  повторять  $M$  раз

если (эл:=[ $\neg k$ ]) макс то макс:=эл; ном:= $M$  все

кпвт;

вывод ном;

3. Для расчетов необходимо взять конкретную функцию, лучше ту же самую, с которой рассматривались прочие примеры с интегрированием.

числ  $a, b, \epsilon, M, x$ , инт, инт1,  $H$ ;

ввод  $a, b, M, \epsilon$ ;

инт:=99999; (можно присвоить любое нереальное для данного интеграла значение)

повторять инт1:=инт;  $H:=(b-a)/M$ ;  $x:=a$ ; инт:=0;

повторять  $M$  раз инт:=инт +  $f(x)$ ;  $x:=x+H$  кпвт;

инт:=инт ·  $H$ ;  $M:=M-2$ ;

до  $|\text{инт}-\text{инт1}| < \epsilon$ ;

вывод инт;

### Гл. 3, п.6

1. В данном случае нужно одновременное выполнение двух условий. Выпишем только "действующий" оператор:

если  $\sqrt{(x - x_0)^2 + (y - y_0)^2 + h^2} \leq D$  и  $h \geq H$  то вывод 1  
иначе вывод 0

все

2. Тело алгоритма без заголовка, описаний и ввода:

$x := a; \phi := f(a);$

пока  $f(x) \cdot \phi > 0$  повторять

$x := x + H$

до  $x \geq 0$ ;

Здесь можно применить и цикл пока или цикл до со сложным условием.

Гл. 3, п. 7

Оптимальный вариант дает нестандартная управляющая конструкция:

если  $T_1 = 0$  то перейти к C2;

если  $T_2 = 0$  то перейти к C1;

если  $T_1 < T_2$  то C1:  $T := T_1$  иначе C2:  $T := T_2$  все

В последней строке можно "вынести за скобки" присваивание:

(если  $T_1 < T_2$  то C1:  $T_1$  иначе C2:  $T_2$ ) все)  $\rightarrow T$ ;

Гл. 3, п. 8

1. Функция:

функ факт ( $M$ );

числ  $\phi$ ;

$\phi := 1$ ;

меняя  $M$  повторять  $M$  раз  $\phi := \phi \cdot M$  кпвт;

возврат  $\phi$ ;

При программировании удобно для  $\phi$  отвести РХ. Вследствие недостатков в конструкции ПМК эта функция не будет давать правильный результат при  $M = 0$ . Чтобы избавиться от этого недостатка предпоследнюю строку нужно записать в виде:

если  $M > 0$  то меняя  $M$  повторять  $M$  раз  $\phi := \phi \cdot M$  кпвт все



Головная программа:

алг сочетания и размещения;

числ  $k, h$ , разм;

разм:=факт(ввод  $n$ );

разм:=вывод (факт  $(n - \text{ввод } k)^{-1} \cdot \text{разм}$ );

вывод факт  $(k)^{-1} \cdot \text{разм}$

кон

Возведение в степень  $-1$  программируется командой "F  $1/x$ ". Заметим, что функция *факт* при программировании по общим правилам *некорректно работает со стеком* (объяснение этого термина см. в п. 9). Если же ее запрограммировать так:

$x \rightarrow \text{П} 0 \quad \text{FO} 1 \text{ П} \xrightarrow{\quad} x 0 \quad \times \quad \text{FLO} \quad \square \quad \text{V/O}$

то обращение к ней может быть и не первым в выражении, что позволяет гораздо короче запрограммировать головную программу:

вывод {сочетаний=}

(вывод {размещений=} (факт(ввод  $n$ )/факт( $n - \text{ввод } k$ ))  
/факт( $k$ ));

2. Функция имеет вид

функ произведение  $(K, M)$ ;

числ пр; косв+  $k$ ; (вариант: косв-  $m$ )

пр:=1;  $k:=K$ ; (вариант  $m:=M$ )

повторять  $M - K + 1$  раз

пр:=пр  $\cdot k$ ; [ $+k$ ]; (вариант: пр:=пр  $\cdot m$ ; [ $-m$ ])

кпвт;

возврат пр;

Такая функция правильно обрабатывает все случаи  $K \leq M$ . Головная программа сложностей не представляет.

3.

проц числ положит (арг косв-  $a$ , косв  $M$ , чис);

косв+ счет;

```

    счет:=0;
    повторять [M] раз
        если [a] > 0 то [+ счет] все
    кпвт;
    [чис]:=счет;
    возврат;

```

Гл. 3, п. 9

1.

алг дифуравнение;

дифур (1, 2, 0.1, 1, адрес  $\Phi$ ); дифур (1, 2, 0.05, 1, адрес  $\Phi$ )

кон

проц дифур (арг  $a, b, H, y_0$ , функ  $f$ );

числ  $x, y$ ;

$x:=a$ ;  $y:=y_0$ ;

повторять  $(b - a)/H$  раз

вывод ( $y:=y + f(x, y) \cdot H$ );  $x:=x + H$ ;

кпвт;

возврат;

При программировании процедуры удобно для  $a, b$  и  $y_0$  взять РХ.

функция  $\Phi(x, y)$ ;

возврат  $y/x^2$ ;

Это уравнение допускает простое аналитическое решение  $y = c e^{-1/x^2}$ , которым можно воспользоваться для контроля.

## СПИСОК ЛИТЕРАТУРЫ

### Учебно-справочная по программируемым микрокалькуляторам

1. Блох А.Ш., Павлонский А.И., Пенкрат В.В. Программирование на микрокалькуляторах. Минск: Высшая школа, 1981. 192 с.
2. Данилов И.Д., Пухначев Ю.В. Микрокалькуляторы для всех. М.: Знание, 1986. 192 с.
3. Дьяконов В.П. Справочник по расчетам на микрокалькуляторах. М.: Наука, 1986. 224 с.
4. Трохименко Я.К., Любич Ф.Д. Инженерные расчеты на микрокалькуляторах. Киев: Техніка, 1980. 384 с.
5. Цветков А.Н., Епанечников В.А. Прикладные программы для микро-ЭВМ "Электроника БЗ-34", "МК-56", "МК-54". М.: Финансы и статистика, 1985. 175 с.

### Учебно-популярная по программируемым микрокалькуляторам

1. Данилов И.Д. Секреты программируемого микрокалькулятора. М.: Наука, 1986. 159 с. (Б-ка "Квант". Вып. 55).
2. Кибернетика. Микрокалькуляторы в играх и задачах. М.: Наука, 1986. 160 с.
3. Романовский Т.Б. Микрокалькулятор в рассказах и играх. Рига: Зинатне, 1984. 120 с.
4. Трохименко Я.К., Любич Ф.Д. Микрокалькулятор, ваш ход! М.: Радио и связь, 1985. 224 с.

### Общие вопросы программирования

1. Звенигородский Г.А. Первые уроки программирования. М.: Наука, 1985. 207 с. (Б-ка "Квант". Вып. 41).
2. Штернберг Л.Ф. Разработка и отладка программ. М.: Радио и связь, 1984. 96 с.

## СОДЕРЖАНИЕ

Предисловие . . . . .	3
Введение . . . . .	4
<b>Глава 1. Знакомство с микрокалькулятором. . . . .</b>	<b>7</b>
1. Структура микрокалькулятора и выполнение вычислений. . . . .	7
2. Использование стека. . . . .	14
3. Запоминание и исполнение программы . . . . .	20
<b>Глава 2. Основные понятия программирования . . . . .</b>	<b>27</b>
1. Алгоритм, исполнитель, программа . . . . .	27
2. Ввод, вывод, переработка данных . . . . .	30
3. Формы записи программы . . . . .	40
4. Синтаксическая и смысловая правильность программы. . . . .	42
<b>Глава 3. Программирование . . . . .</b>	<b>53</b>
1. Условие и ветвления. . . . .	53
2. Циклы . . . . .	62
3. Итерационные циклы . . . . .	78
4. Обработка таблиц . . . . .	89
5. Сочетания ветвлений и циклов . . . . .	102
6. Сложные условия . . . . .	113
7. Нестандартные управляющие конструкции. . . . .	120
8. Подпрограммы . . . . .	127
9. Взаимодействие нескольких подпрограмм. . . . .	142
<b>Глава 4. Удобство использования программы и ее качество . . . . .</b>	<b>150</b>
1. Организация диалога с пользователем. . . . .	150
2. Оценка качества и оптимизация программы . . . . .	156
<b>Глава 5. Отладка и эксплуатация программы. . . . .</b>	<b>162</b>
1. Тестирование . . . . .	162
2. Поиск ошибок . . . . .	167
3. Внесение исправлений. . . . .	177
4. Документация программы. . . . .	179
5. Эксплуатация программы . . . . .	182
<b>Глава 6. Погрешности расчетов на микрокалькуляторах. . . . .</b>	<b>183</b>
1. Влияние погрешностей на результат . . . . .	183
2. Классификация и оценка погрешностей . . . . .	185
3. Методы снижения погрешностей . . . . .	189
Заключение. . . . .	193
Указания к решению задач . . . . .	194
Список литературы. . . . .	202

НАУЧНО-ПОПУЛЯРНОЕ ИЗДАНИЕ

Штернберг Леонид Фрицевич

КУРС ПРОГРАММИРОВАНИЯ МИКРОКАЛЬКУЛЯТОРОВ  
БЗ-34, МК-52, МК-54, МК-56, МК-61

Редактор *А.В. Лысенко*  
Художественный редактор *А.В. Ро*  
Обложка художника *М.В. Носовой*  
Технический редактор *Н.В. Павлова*  
Корректор *Е.И. Березина*

ИБ № 5798

---

Сдано в набор 31.12.87.	Подписано в печать 14.10.88.	Т-18265.
Формат 84×108 1/32.	Бумага офсетная № 2.	Гарнитура Пресс-Роман.
Печать офсетная.	Усл.печ.л. 10.92.	Усл.кр.-отт.11,24.
Уч.-изд.л.10,94.		
Тираж 50 000 экз.	Заказ 1528	Цена 55 коп.

---

Ордена Трудового Красного Знамени издательство  
"Машиностроение", 107076, Москва, Стромьинский пер., 4

---

Отпечатано в московской типографии № 6 Союзполиграфпрома  
при Государственном комитете СССР по делам издательств, полиграфии  
и книжной торговли,  
109088, Москва, Ж-88, Южнопортовая ул., 24,  
с оригинала-макета, изготовленного в издательстве "Машиностроение"  
на наборно-пишущих машинах

## УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Издательство "Машиностроение"  
выпустит в 1989 году  
следующие книги:

### *Производственная литература*

К о с а р е в, Е. А. Естественная форма диалога с ЭВМ. — Л.: Машиностроение, 1989. — 7,5 л.: ил. — (ЭВМ в производстве). — (В обл.): 80 к.

В книге рассказано о практических возможностях общения человека с ЭВМ или автоматизированной системой с помощью голоса на основе теории распознавания и синтеза речи. Показаны преимущества речевого диалога с ЭВМ по сравнению с традиционным клавиатурно-экраным обменом. Рассмотрены составные части проблемы распознавания речи и вопросы применения устройств распознавания и синтеза речи в автоматизированных системах различного назначения. Приведены структуры устройств речевого диалога.

Для инженерно-технических работников, занимающихся автоматизацией производственных процессов, средств связи, научных исследований.

**Л о с е в Г. М. Системы передачи данных сетей ЭВМ. — Л.: Машиностроение, 1989. — 8,5 л.: ил. — (ЭВМ в производстве). — (В обл.): 80 к.**

В книге изложены минимально необходимые сведения по теоретическим основам, техническим средствам и принципам построения систем передачи данных для вычислительных сетей научных и производственных предприятий. Приводятся примеры реализации систем на базе существующей аппаратуры и реальных каналов связи.

Для инженерно-технических работников машиностроительных предприятий, не имеющих специальной подготовки в области вычислительной техники.

**М а й о р о в В. Г., Г а в р и л о в А. И. Практический курс программирования микропроцессорных систем. — М.: Машиностроение, 1989. — 16 л.: ил. — (В пер.): 1 р. 30 к.**

Рассмотрены основные принципы и приемы программирования микропроцессорных систем на базе микропроцессоров серии КР 580. Дан лабораторный практикум для начинающих программистов.

Для инженерно-технических работников, занятых вопросами разработки и применения микропроцессорных систем в различных отраслях народного хозяйства для начинающих программистов микропроцессорных систем.

## *Учебная литература*

### Для повышения квалификации

Захаров И.В. Техническое обслуживание и эксплуатация микроЭВМ "Электроника-60": Учеб. пособие для подготовки и повышения квалификации кадров в учебной сети ЦСУ СССР. — М.: Машиностроение, 1989. — 18 л.: ил. — (В пер.): 75 к.

Рассмотрены структуры, принципы построения и эксплуатации вычислительных комплексов, построенных на базе микроЭВМ "Электроника-60". Описаны устройство и функционирование узлов и блоков микроЭВМ "Электроника-60", а также даны рекомендации по ее техническому обслуживанию и ремонту.

Может быть полезно всем специалистам, использующим в своей работе данную ЭВМ, а также занимающимся вопросами ее эксплуатации, технического обслуживания и ремонта.



**М ы м р и н М. П.** Конструкция, применение, программирование и ремонт ПЭВМ "Агат": Учеб. пособие для подготовки и повышения квалификации кадров в учебной сети ЦСУ СССР. — М.: Машиностроение, 1989. — 21 л.: ил. — (В пер.): 90 к.

Рассмотрены устройство и функционирование узлов и блоков персональной ЭВМ (ПЭВМ) "Агат", являющейся одной из базовых ЭВМ для средних общеобразовательных школ при изучении предмета "Информатика и вычислительная техника". Изложены вопросы применения машины и программирования на ней. Наряду с рекомендациями по обслуживанию и ремонту машины приведено описание блока ее контроля и набор тестовых программ. Даны практические советы по их использованию при выявлении неисправностей ПЭВМ "Агат". Книга снабжена большим количеством примеров и упражнений, позволяющих получить необходимые навыки по самостоятельному программированию и обслуживанию машины.

Может быть полезно широкому кругу специалистов, занимающихся использованием, обслуживанием и ремонтом персональных ЭВМ.

