

С. С. ЛАВРОВ

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ

*Допущено Министерством
высшего и среднего специального образования СССР
в качестве учебного пособия
для студентов вузов, обучающихся
по специальности «Прикладная математика»*



ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
Москва 1973

Введение в программирование, С. С. Лавров.
Главная редакция физико-математической литературы
изд-ва «Наука», М., 1973.

Книга является учебным пособием по курсу «Вычислительные машины и программирование». Основное внимание уделено программированию на машинном языке, владение которым необходимо многим программистам, в первую очередь — разработчикам математического обеспечения ЭВМ. В книге также освещены общие вопросы составления алгоритмов и кратко описан алгоритмический язык АЛГОЛ 60. В основу книги легли лекции, прочитанные автором на факультете вычислительной математики и кибернетики Московского государственного университета. Изложение сопровождается большим количеством примеров и упражнений, снабженных решениями.

Книга рассчитана на студентов, специализирующихся в области прикладной математики и системного программирования, и может оказаться полезной преподавателям соответствующих дисциплин, а также лицам, для которых работа на ЭВМ является или должна стать основным видом деятельности.

Илл. 14, библи. 32 назв.

© Издательство «Наука», 1973.

Л $\frac{0224-1789}{042(02)-73}$ 12-73

ОГЛАВЛЕНИЕ

Предисловие	5
Введение	9
Глава 1. Общие сведения об ЭВМ	11
§ 1. Назначение, устройство и схема работы ЭВМ	11
§ 2. Запоминающее устройство	13
§ 3. Арифметическое устройство	14
§ 4. Устройство управления	15
§ 5. Вводные устройства	17
§ 6. Выводные устройства	18
Глава 2. Представление чисел	20
§ 1. Позиционные системы счисления	20
§ 2. Перевод чисел из одной системы счисления в другую	24
§ 3. Представление чисел в машине	31
§ 4. Выполнение арифметических операций в машине	33
Глава 3. Команды и их представление в машине	37
Глава 4. Алгоритмы	43
§ 1. Общее понятие об алгоритмах	43
§ 2. Еще один пример алгоритма	47
§ 3. Средства, используемые при записи алгоритмов	48
§ 4. Другие средства записи алгоритмов	52
§ 5. Детализация алгоритмов	57
§ 6. Алгоритм приведения подобных членов в многочлене	59
Глава 5. Система команд	64
§ 1. Машины типа М-20	64
§ 2. Арифметические операции	67
§ 3. Операции преобразования кодов	69
§ 4. Операции управления	70
§ 5. Запись программ	72
Глава 6. Программирование арифметических выражений и присваиваний	74
Глава 7. Циклы	83
§ 1. Виды циклов	83
§ 2. Программирование простых итерационных циклов	85
§ 3. Программирование разветвлений в программах	88
§ 4. Итерационные циклы (продолжение)	95
§ 5. Приемы вычисления конечных сумм. Чистка циклов	99
§ 6. Программные массивы	107
§ 7. Программирование простых структурных циклов	109
Глава 8. Элементы языка АЛГОЛ 60	124
§ 1. Значения и типы	124
§ 2. Обозначения	125

ОГЛАВЛЕНИЕ

§ 3. Переменные	129
§ 4. Выражения	131
§ 5. Операторы присваивания	135
§ 6. Блоки	136
§ 7. Операторы перехода	139
§ 8. Операторы ввода и вывода	140
§ 9. Условные и составные операторы	142
§ 10. Операторы цикла и пустые операторы	145
§ 11. Пример составления программы	151
§ 12. Процедуры	157
§ 13. Примеры процедур	163
§ 14. Вызов параметров	170
§ 15. Функции	172
§ 16. Переключатели	174
Глава 9. Кратные циклы	176
§ 1. Размещение в памяти многомерных массивов	176
§ 2. Программирование кратных циклов	179
Глава 10. Индекс-регистр	194
§ 1. Система команд М-20 (продолжение)	194
§ 2. Программирование с использованием индекс-регистра	199
Глава 11. Программирование циклов с нелинейностями	210
Глава 12. Программирование логических выражений	216
§ 1. Представление логических значений в машине	216
§ 2. Логические операции и операции сдвига в системе команд машины М-20	219
§ 3. Приемы программирования логических выражений	224
§ 4. Применение логических операций и сдвигов для преобразования произвольных слов	238
Глава 13. Арифметические операции машины М-20 (продолжение)	242
§ 1. Выполнение арифметических операций в машине	242
§ 2. Модификации арифметических операций	249
§ 3. Примеры применения модификаций арифметических операций	252
Глава 14. Ввод, вывод и обмен с внешней памятью	255
§ 1. Устройства ввода-вывода машин типа М-20	255
§ 2. Внешние запоминающие устройства	262
§ 3. Контроль операций ввода, вывода и обмена	263
§ 4. Операции ввода, вывода и обмена	267
§ 5. Ввод и вывод чисел	280
Глава 15. Подпрограммы	292
§ 1. Подпрограммы и обращения к ним	292
§ 2. Программирование по образцу	294
§ 3. Неизменяемые подпрограммы	299
§ 4. Элементы модульного программирования	303
§ 5. Настройка модулей по параметрам	309
§ 6. Настройка модулей по месту	316
Заключение	320
Решения упражнений	324
Литература	347
Предметный указатель	349
Указатель операций машины М-20	351

ПРЕДИСЛОВИЕ

В наши дни большинство программ для электронных вычислительных машин (ЭВМ) составляется с помощью машинно независимых языков программирования, таких, как АЛГОЛ 60 [2], ФОРТРАН [26] и др. Важность этих средств программирования несомненна. С их помощью удастся во много раз сократить затраты человеческого труда на программирование и решение задачи. Но можно ли считать, что подготовка программиста должна сводиться к изучению одного из этих языков? По-видимому, нет. При этом остается неудовлетворенной естественная любознательность человека, знающего, что написанная им программа переводится затем на какой-то совсем другой язык, который только и способна понять машина. Сужается кругозор программиста, для которого остаются пустым звуком практические рекомендации о целесообразности применения тех или иных средств входного языка или, наоборот, о нежелательности пользоваться отдельными приемами программирования, доступными в этом языке. Но главное не в этом. Программист, не владеющий машинным языком, оказывается в положении человека, попавшего в чужую страну, не знающего ее обычаев и не способного объяснить без переводчика. Он скован в своих действиях, он может двигаться только по придуманным для него маршрутам и не в состоянии оценить последствия многих своих поступков.

Программисту-профессионалу, для которого общение с вычислительной машиной — не эпизод, а существо его деятельности, знание машинного языка жизненно необходимо. Программы, которые он составляет, предназначены, как правило, для массового и длительного употребления. Они должны быть эффективны и экономичны во всех отношениях. Для этого программист должен, прежде всего, быть специалистом в той области знания, к которой относятся решаемые им задачи. Но он обязан также в совершенстве владеть главным своим оружием — вычислительной машиной. Для этого нужно многое — уметь распоряжаться памятью и другими ресурсами машины, знать, как можно представить в машине различные связи и отношения, существующие между элементами обрабатываемой информации, владеть приемами быстрого поиска нужных данных в обширных справочных (информационных) массивах, уметь

анализировать тексты на различных формальных языках, владеть методами моделирования на машинах различных физических, технологических, экономических, биологических и социальных процессов. Но в основе всего этого лежит знание основных принципов устройства ЭВМ, взаимодействия ее частей, типов машинных операций и приемов их использования для описания простейших вычислительных процессов, разветвления этих процессов и, самое главное, — циклического повторения их частей, методов составления подпрограмм и организации их взаимодействия.

Именно этот круг вопросов вместе с изучением основ одного из универсальных языков программирования целесообразно охарактеризовать как «введение в программирование» для будущих специалистов по приложениям математики к решению естественно-научных, технических, экономических и управленческих задач. Именно в этих вопросах, несмотря на их сравнительную простоту, чувствуют себя неуверенными почти все начинающие программисты. Поэтому именно здесь они допускают больше всего просчетов, принимают больше всего неоправданных, излишне сложных или просто неверных решений. Есть известная закономерность в том, что элементарные сведения и навыки, не усвоенные в свое время, воспринимаются впоследствии с большим трудом, — овладению ими мешает ложное ощущение, что и без них можно прожить, можно чем-то их заменить. В действительности же такая замена никогда не бывает полноценной.

Эти соображения (а также программы, действующие во многих университетах) определили содержание книги. Разумеется, она составляет не более чем введение в программирование, и будущим программистам-профессионалам предстоит изучить еще немало программистских дисциплин. Однако все они в большей или меньшей степени основываются на материале этой книги.

Одна из самых трудных проблем, встающих перед автором учебника по программированию, — это выбор учебной машины. Любое решение этой проблемы может стать предметом довольно обоснованной критики. Многие авторы конструируют свою собственную, реально не существующую учебную вычислительную машину. Однако достичь общности и абстрактности основных понятий программирования лишь этим путем не удастся. Просто одни технические детали заменяются при этом другими, обычно столь же искусственными. В этой книге в качестве учебной рассматривается машина из семейства М-20, не существенно, какая именно — отличиями одной марки от другой как раз можно пренебречь. Для целей, поставленных в этом курсе, устарелость машин типа М-20 (отсутствие аппарата базирования, слабость механизма индексирования, неспособность к многопрограммному режиму работы, примитивность схемы сочленения внешних устройств с центральным процессором) не является помехой. Более того, нововведения второго и

третьего поколений ЭВМ, имея огромную ценность с точки зрения повышения эффективности машин, облегчения ручного, и особенно автоматического, программирования, мешают, если слишком рано начать с ними знакомиться, усвоению основ программирования и могут быть по-настоящему поняты и оценены только после овладения этими основами. Это — задача последующих курсов.

Машина М-20 хороша также тем, что она — трехадресная и в программах для нее нет нужды прибегать к явным операциям считывания содержимого ячеек на сумматор и записи содержимого сумматора в оперативную память. Ее система команд отличается простотой и стройностью, редкими для машин первого поколения. Может быть, решение избрать машину М-20 в качестве учебной было продиктовано и тем, что мне довелось больше всего иметь дело с машинами этого семейства. Не исключено, что некоторые чересчур специфические детали системы команд М-20 все же излишне отягощают книгу. Но, с другой стороны, они, так же как отдельные более сложные примеры и упражнения, могут дать пищу уму и фантазии наиболее заинтересованных читателей, обогатить их арсенал программистских приемов. Предлагаемая книга — это не учебник, усвоение всего материала которого строго обязательно, а пособие, из которого каждый может черпать в меру своих интересов.

Возможно, некоторым читателям глава 8, посвященная языку АЛГОЛ 60, покажется недостаточно органически связанной с остальным материалом книги. Первоначально книга не содержала этой главы, хотя и была рассчитана на параллельное изучение АЛГОЛа. Дело в том, что пособие по АЛГОЛу [18], достаточно полное и обстоятельное, уже было написано тем же автором. Но в конце концов я признал правоту рецензентов, настоявших на включении этой главы в книгу, а заодно постарался учесть и критику, высказанную по адресу работы [18].

Хотя машина М-20 и язык АЛГОЛ 60 широко используются в этой книге для разъяснения понятий и демонстрации приемов программирования, не они, а эти понятия и приемы являются предметом изучения. Человеку, прочитавшему эту книгу и собирающемуся составлять свои программы на языке одной из машин семейства М-20 или на АЛГОЛе, придется доучиваться еще многим конкретным вещам. Возможно, для этого потребуется столько же усилий, сколько для практического овладения языком машины Минск-22 или ФОРТРАНа. Однако можно надеяться, что эти трудности будут носить уже не принципиальный, а технический характер.

Содержащиеся в книге примеры, программы, упражнения, рисунки, таблицы и формулы имеют свою нумерацию в каждом параграфе и упоминаются в пределах этого параграфа под своим простым номером. В других параграфах ссылки на них составляются из номера главы, номера параграфа и номера примера, программы

и т. д. внутри параграфа. Например, программа 2 параграфа 3 гл. 15 упоминается в других параграфах как программа 15.3.2.

Обычные знаки препинания в текстах на языке АЛГОЛ 60 имеют особый смысл. Поэтому в предложениях, включающих в себя такие тексты, правила пунктуации русского языка иногда не соблюдаются.

Автор глубоко признателен О. Н. Кацковой за деятельное участие в подготовке одного из первых вариантов рукописи и за многочисленные беседы, в ходе которых вырабатывался стиль изложения, А. П. Ершову, Л. Н. Королеву и В. М. Меншикову за обстоятельные рецензии и многочисленные замечания, которые по мере своих возможностей я постарался учесть, А. Н. Балуюеву, И. Л. Братчикову, С. М. Ермакову и Б. А. Кацеву за ряд полезных замечаний, также принятых во внимание. Особенно мне хотелось бы поблагодарить А. И. Срагович. Это на ее программах я постигал искусство программирования и из них черпал идеи, которые много лет спустя воплотились в этой книге. Должны быть с благодарностью упомянуты студенты факультета вычислительной математики и кибернетики Московского университета (прием 1970 года), в общении с которыми автору удалось обнаружить множество недостатков этого курса в его первоначальном виде. Я должен также с благодарностью назвать имена Н. Вирта, Э. Дейкстры, Д. Кнута и Дж. Маккарти. Их работы сильнее всего способствовали формированию моих теперешних взглядов на программирование.

С. Лавров

ВВЕДЕНИЕ

Электронные вычислительные машины (ЭВМ) — одно из крупнейших достижений современной науки и техники. Каждый год приводит к новым усовершенствованиям схем и конструкций этих машин, расширяет области их применения, обогащает методы их использования для решения самых различных задач. Каждое десятилетие характеризуется настолько крупными изменениями в технологии производства, конструкции, оснащении и способах использования ЭВМ, что есть все основания говорить о появлении нового поколения этих машин.

Созданные для решения задач вычислительной математики, ЭВМ находят применение для управления производственными процессами, для экономических расчетов, для хранения и обработки статистической и справочной информации, для моделирования различных форм умственной деятельности человека.

Дальнейшее развитие электронных вычислительных машин со временем приведет к тому, что в большинстве отраслей умственного труда человека машины будут оказывать ему такую же помощь и настолько же расширять его возможности, насколько сейчас облегчают машины физический труд людей.

Уже сейчас требуются многие тома, чтобы описать хотя бы основные способы применения ЭВМ даже к решению только важнейших типов задач. Дело не только в том, что сами задачи и подходы к их решению весьма разнообразны, но и в том, что существуют разные способы описания этих задач, их представления в виде, пригодном и удобном для обработки на ЭВМ. Наконец, многообразны сами машины и их оборудование, что также накладывает свой отпечаток на характер и методы использования машин для решения задач.

Однако все электронные вычислительные машины имеют и много общего между собой, во всех машинах можно найти устройства и

узлы, выполняющие сходные функции. Более того, самые разные задачи, взятые из различных областей науки, техники или экономики, при ближайшем рассмотрении также имеют общие черты в своей структуре, в природе действий, из которых складывается процесс их решения, в структуре их исходных данных и характере связей между элементами этих данных. Соответственно этому много общего есть и в программировании — так называется совокупность средств, с помощью которых описываются исходные данные, способы, процессы и результаты решения задач на ЭВМ. Изложить это общее, рассказать начинающему программисту об основных идеях, методах и практических приемах программирования, пригодных для подавляющего большинства ЭВМ, — такова задача книги.

Глава 1. ОБЩИЕ СВЕДЕНИЯ ОБ ЭВМ

§ 1. Назначение, устройство и схема работы ЭВМ

Электронные вычислительные машины предназначены для автоматического решения разнообразных трудоемких задач обработки данных. *Данными* называется все, что может быть представлено различными устойчивыми состояниями некоторого *носителя* данных. Данными могут служить, например, знаки, написанные или напечатанные на бумаге, отверстия, пробитые в перфоленте или перфокартах, наличие или отсутствие напряжения на контактах, направление намагниченности участка магнитного слоя, появление или отсутствие некоторого сигнала в определенный промежуток времени. *Обработка данных* — это процесс, который применяется к некоторой совокупности данных (*исходных данных*) и приводит к созданию новой совокупности данных — *результату* обработки. Как правило, предполагается, что этот процесс детерминирован — если его применить любое число раз к одной и той же совокупности исходных данных, то результат получится каждый раз один и тот же. Задача обработки данных считается трудоемкой, если выполнение ее трудно и утомительно для человека, а главное — если человек, в силу своих психологических особенностей, не в состоянии выполнить ее без ошибок, обнаружение и исправление которых требует еще более длительных усилий. Именно в надежности работы, в достоверности результатов состоят предпосылки применения ЭВМ в очень многих случаях.

Автоматическое решение означает, что процесс обработки данных вплоть до получения результата происходит без участия и без вмешательства человека, хотя исходные данные могут быть заданы, составлены или подготовлены человеком. Чтобы машина могла решать разные задачи, описание конкретного процесса обработки данных должно сообщаться машине вместе с исходными данными. Такое описание называется *программой*. Таким образом, исходные данные фактически распадаются на две части: программу обработки и собственно данные, подлежащие обработке.

Из сказанного выше видно, что в машине должно храниться и перерабатываться большое количество данных (программа, исходные данные, промежуточные и окончательные результаты

обработки). Устройство, предназначенное для хранения данных любого назначения и происхождения, называется *запоминающим устройством (ЗУ)* или *памятью* машины.

Обработка данных очень часто заключается в вычислениях — выполнении арифметических операций над числами, хранящимися в машине. Но как будет видно, машина должна иметь возможность выполнять, кроме арифметических, целый ряд других операций, в которых, наряду с арифметическими элементами, выполняются действия совсем иного характера. Большинство этих операций выполняется узлом машины, носящим название *арифметического устройства (АУ)*.

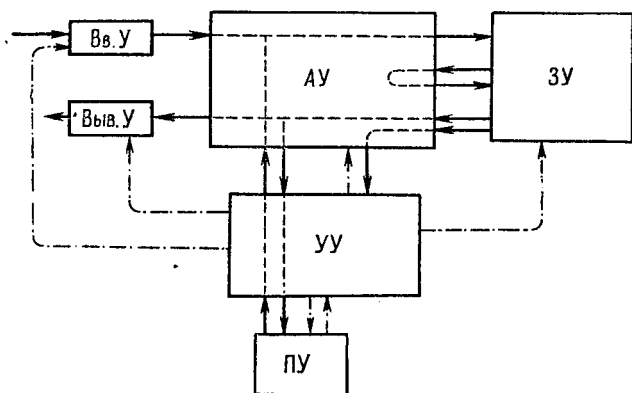


Рис. 1.1.1. Схема устройства ЭВМ.

Специальное *вводное устройство* (Вв. У) необходимо для ввода данных в машину, точно так же, как *выводное устройство* (Выв. У) — для вывода результатов работы машины.

Работа всех устройств машины должна быть согласована между собой. Для этой цели служит *устройство управления* (УУ). Частью УУ является *пульт управления* (ПУ), с помощью которого человек (оператор машины) производит начальные операции по пуску машины в действие, следит за ее работой и в случае нужды останавливает машину или иным способом вмешивается в ее работу. Схема машины изображена на рис. 1.1.1. Сплошными линиями показаны пути информации, штрихпунктирными — управляющие цепи.

Программа, по которой машина решает задачу, а точнее — выполняет некоторый процесс обработки данных, распадается на множество мелких составных частей, называемых *командами*. Каждая команда определяет элементарную частичку процесса обработки данных — *машинную операцию*. Как и у всего процесса, у каждой операции есть свои исходные данные, называемые ее *операндами*, и свой результат. Результат операции вырабатывается в зависи-

мости от оперантов по точно определенным для данной операции правилам, заложенным в конструкцию машины. Для разных операций эти правила различны, в остальном же выполнение любой машинной операции (так называемый *элементарный цикл*, или *такт* работы машины) происходит по одной и той же схеме и складывается из следующих действий.

1. Из запоминающего устройства в устройство управления поступает (считывается) очередная команда.

2. Устройство управления анализирует команду и определяет тип операции.

3. В арифметическое устройство выбираются операнды операции. Обычно операнды считываются из запоминающего устройства, но для некоторых операций операнды содержатся в самой команде.

4. В арифметическом устройстве по сигналам, поступающим из устройства управления, вырабатывается результат операции.

5. Результат, как правило, записывается в запоминающее устройство. Однако для некоторых типов операций результат может быть передан в устройство управления или остаться в арифметическом устройстве, с тем чтобы им могли воспользоваться последующие операции.

6. Определяется, какая команда должна выполняться следующей, после чего можно вернуться к началу цикла.

Эта грубая схема в дальнейшем будет уточнена. Кроме того, для конкретных машин возможны различные отступления от этой общей схемы. Об этом тоже будет сказано ниже, а для той машины (М-20), на примере которой будут излагаться методы программирования, схема эта будет изложена со всеми необходимыми деталями.

§ 2. Запоминающее устройство

Запоминающее устройство предназначено для хранения перерабатываемых данных и данных, содержащих управляющую информацию, во время работы машины. В большинстве случаев современные ЭВМ имеют не одно, а несколько ЗУ, различающихся между собой как по своим функциям, так и по характеристикам.

ЗУ характеризуется своим объемом — количеством данных, которые могут быть в нем размещены, временем обращения к нему — временем, необходимым для занесения определенного объема данных в ЗУ или извлечения его оттуда, а также способом обращения к нему. Обращение к ЗУ необходимо при выполнении каждой машинной операции. Прежде всего нужно извлечь из ЗУ команду, по которой будет выполняться операция. Затем из ЗУ должны быть вызваны операнды данной операции. Наконец, результат операции должен быть отослан на хранение в ЗУ. В некоторых случаях часть этих обращений к ЗУ может и не понадобиться, но это не меняет существа дела, заключающегося в необходимости постоянной

тесной связи арифметического и запоминающего устройств и обмена данными между ними.

Если машина обладает несколькими ЗУ, то такую постоянную связь с АУ имеет, как правило, лишь одно из них, называемое *оперативным* запоминающим устройством (ОЗУ) или *внутренним* запоминающим устройством. Остальные ЗУ называются *внешними*.

Помимо наличия оперативной связи с АУ оперативное ЗУ отличается от внешних и по другим характеристикам. Оно обладает, как правило, наименьшим объемом среди всех ЗУ, но зато и время обращения к нему значительно меньше, чем к внешним ЗУ.

Данные из ОЗУ можно извлекать только минимальными порциями, например, по одному числу или по одной команде, тогда как обмен данными с внешними ЗУ идет обычно более или менее значительными группами. Этот обмен происходит между ОЗУ и внешними ЗУ. Когда по ходу решения задачи появляется необходимость в дополнительных данных, которых нет в ОЗУ, они вызываются из внешних ЗУ в оперативное. Наоборот, когда оперативное ЗУ заполнилось данными, которые не будут нужны в ходе ближайших вычислений, их можно послать на хранение в одно из внешних ЗУ, освободив место в ОЗУ для текущих данных: результатов промежуточных вычислений или очередных частей программы.

В связи с необходимостью обращаться к ОЗУ за минимальными порциями информации все оно разбивается на части — *ячейки*, каждая из которых предназначена для хранения одной такой порции: кода числа или команды. Ячейки имеют номера, которые необходимо указывать при обращении к ним; эти номера называются *адресами*. Каждая ячейка имеет только один номер, разные ячейки должны иметь разные номера. Нередко возникает несоответствие между объемами единицы хранения данных (ячейки) и единицы использования данных (например, числа или команды). Происходит это, например, потому, что команды разных операций могут иметь разную длину, или потому, что требуется (и конструкция машины позволяет) производить вычисления над числами разной разрядности. В таких случаях система адресации обрабатываемых данных и команд отличается от адресации ячеек.

Внешние ЗУ также делятся на ячейки, но эти ячейки могут и не иметь определенных номеров. Нумеруются в таких случаях лишь такие группы данных, которые предназначены для одновременной передачи из внешнего ЗУ в оперативное или наоборот.

§ 3. Арифметическое устройство

В арифметическом устройстве выполняются все операции над данными. Название устройства не является вполне точным, так как ряд операций не имеет чисто арифметического характера. Арифметическое устройство содержит несколько *регистров* — элементов,

предназначенных для хранения и преобразования данных. Емкость регистра — количество данных, которые могут быть на нем размещены, обычно соответствует емкости ячейки памяти. Выполнение операции начинается с пересылки ее операндов из оперативной памяти на *регистры операндов*. По завершении операции ее результат образуется на *регистре результата*, называемом иногда *сумматором*. Это последнее название объясняется тем, что выполнение многих операций сводится к последовательности сложений чисел, представленных порциями данных, хотя наряду со сложениями на сумматоре выполняются и другие преобразования элементов данных.

Кроме этой главной функции арифметическое устройство выполняет и другие задачи. Через регистры АУ обычно проходит весь поток данных в машине (см. рис. 1.1.1). При вводе программы в машину данные поступают порциями из вводного устройства сначала на регистры АУ, а затем в ОЗУ. То же самое, но в другом направлении, происходит при выводе данных из ОЗУ на выводное устройство.

Наконец, при обмене данными между оперативным и внешними запоминающими устройствами порции передаваемых данных последовательно вызываются из одного ЗУ на регистр АУ и уже оттуда передаются в другое ЗУ. Такой обмен, так же как ввод и вывод данных, может сопровождаться некоторыми контрольными операциями, производимыми также в арифметическом устройстве. Это возможно благодаря тому, что скорость выполнения операций в АУ обычно бывает намного выше, чем скорость поступления данных из Вв. У. или вывода через Выв. У, а также выше, чем скорость обмена данными с внешними ЗУ. Таким образом, осуществление контроля не вызывает замедления ввода, вывода или обмена данными.

Результат операции, полученный на регистре результата, обычно записывается в одну из ячеек оперативной памяти. Однако, во многих машинах существует возможность сохранить результат операции на регистре результата, не записывая его в ОЗУ, если он нужен только как операнд для следующей операции. При этом отпадает надобность в двух обращениях к ОЗУ — одном для записи результата предыдущей операции, другом — для считывания операнда следующей операции.

§ 4. Устройство управления

Под контролем устройства управления протекает вся работа машины, заключающаяся в многократном повторении элементарного цикла, описанного в § 1. Уже сейчас мы подготовлены к тому, чтобы внести в это описание некоторые уточнения. В состав устройства управления входят, в частности, два регистра — *регистр*

адреса команды и регистр выполняемой команды. К началу выполнения элементарного цикла на регистре адреса команды должен содержаться адрес ячейки ОЗУ, в которой хранится очередная команда программы (мы рассматриваем простейший случай, когда каждая ячейка ОЗУ вмещает в точности одну команду). Содержимое ячейки с этим адресом считывается на регистр выполняемой команды. Код, поступивший на этот регистр, расчленяется на отдельные части, определяющие тип операции, адреса (или непосредственно значения) операндов, адрес результата и другие данные, необходимые для выполнения операции. Часто адреса, по которым при выполнении операции происходит обращение к ЗУ, не содержатся в команде в готовом виде, а должны быть вычислены. Для этой цели устройство управления содержит так называемый *сумматор адресов*, фактически являющийся небольшим арифметическим устройством для выполнения простых действий.

Одно из типичных применений сумматора адресов — это вычисление адреса команды, которая должна выполняться вслед за данной. Существуют операции (называемые *операциями управления*), которые определяют этот адрес сами, в качестве результата своей работы. Для остальных операций адрес следующей команды получается в результате добавления единицы к адресу, хранящемуся на регистре адреса команды. Это сложение выполняется на сумматоре адресов, и его результат снова посылается на регистр адреса команды.

При выполнении некоторых операций над заданными операндами (например, при делении, если делитель окажется равным нулю) результат операции не имеет смысла или выражается числом, которое не может быть представлено в машине. В таких случаях УУ производит *автоматический останов* машины, или сокращенно «Авост». Это сокращение часто расшифровывается как «аварийный останов», что не совсем точно, так как аварию здесь терпит не столько машина, сколько программа, в которой программист или просто ошибся, или не предусмотрел возможность неблагоприятного исхода вычислений. Некоторые команды служат для задания операций ввода, вывода или пересылки данных между различными ЗУ. Управление в таких случаях может передаваться *местным управляющим устройствам*. Под местным управлением целесообразно также выполнять некоторые действия, входящие в состав отдельных операций, выполняемых в арифметическом устройстве. Такое разделение обязанностей между центральным устройством управления и местными устройствами позволяет совместить во времени различные действия, из которых складывается выполнение сложной операции, и существенно ускорить работу машины.

Для контроля за работой машины на пульте управления имеются сигнальные регистры, показывающие состояние регистров АУ, регистра выполняемой команды, адрес выполняемой команды и адреса,

по которым происходит обращение к ОЗУ. Пульт управления имеет также кнопочные или клавишные регистры, с помощью которых можно, приостановив автоматическую работу машины, выполнять вручную отдельные команды, проверять и изменять содержимое ячеек ЗУ, выполнять другие контрольные и управляющие функции. На ПУ имеются кнопки или ключи, позволяющие очищать устройства и регистры машины от содержащейся в них информации, пускать машину в ход или останавливать ее. Может быть предусмотрен останов при выполнении команды с заданным адресом или при обращении к ячейке ЗУ с заданным номером. Отдельные части пульта управления служат для управления работой вводных, выводных и внешних запоминающих устройств.

Специальные ключи могут блокировать выполнение некоторых этапов в нормальной работе машины, например, запись данных в ОЗУ, выборку команд из ОЗУ, переход к выполнению следующей команды и т. п.

§ 5. Вводные устройства

Программист записывает программу и другие исходные данные для своей задачи буквами, цифрами и другими знаками на обычной бумаге или на бланках определенной формы. Затем на специальных устройствах — *перфораторах* эти данные переносятся на перфоноситель — перфокарты или перфоленты. Перфораторы имеют клавиатуру, очень похожую на клавиатуру пишущей машинки. При нажатии на клавишу в очередной позиции носителя пробивается несколько отверстий, расположение которых соответствует нажатой клавише. Таким образом, данные не только переносятся на другой носитель, но преобразуются в другую форму. Однако это преобразование обратимо — по пробивкам можно восстановить исходный текст.

Колода перфокарт или перфолента с исходными данными задачи устанавливается на вводное (или, как еще говорят, *читающее*) устройство. Это устройство способно обследовать все позиции перфоносителя, где могут быть пробиты отверстия, и выяснить, в каких из этих позиций они фактически пробиты. Эти сведения преобразуются в данные, накапливаемые в памяти машины. Опять-таки вся существенная информация, представленная данными на перфоносителе, отображается в данных, записанных в память. Правда, объем данных, которые могут быть размещены в памяти, обычно превышает объем вводимых данных. По окончании ввода уже невозможно установить, какие данные в памяти были только что введены, а какие находились в ней ранее. Чтобы эти старые данные не влияли на работу программы, память перед вводом данных новой задачи обычно *очищают* — переводят во вполне определенное нейтральное состояние. В этом состоянии в памяти машины нет

ни одной содержательной команды, и следовательно, машина не может выполнить никаких полезных действий. Чтобы, несмотря на это, был возможен ввод данных, на пульте управления имеется кнопка «Начальный ввод». Нажатие на эту кнопку запускает вводное устройство. Устройство останавливается (прекращает ввод), либо когда будет введено определенное количество данных, либо когда среди вводимых данных будет обнаружен специальный код, служащий признаком конца вводимых данных. После этого управление передается введенной программе, обычно — команде, попавшей в ячейку с заранее определенным адресом, после чего программа работает автоматически.

Данные, вводимые с читающего устройства, обычно попадают в ОЗУ. Количество введенных данных при этом не может превышать объема ОЗУ. Часто от кнопки «Начальный ввод» вводится лишь *пусковая программа* — короткая программа, состоящая из фиксированного числа команд и располагающаяся в определенных ячейках ОЗУ. Пусковая программа содержит указания о вводе основной программы, о месте размещения этой программы в ОЗУ, о записи программы во внешнее ЗУ, если это требуется, об адресе команды, с которой начинается выполнение основной программы. Если для работы машины по данной программе потребуются дополнительные данные, они могут быть введены командами ввода, содержащимися в основной программе.

Запись программы во внешнее ЗУ сразу же после ее ввода делается обычно для того, чтобы в случае *сбоя* — нарушения нормальной работы машины во время решения задачи — можно было бы повторить решение, не повторяя ввода — операции обычно более длительной, чем передача данных из внешнего в оперативное ЗУ. Во внешнем ЗУ можно также хранить программу задачи во время решения других задач. Таким образом, внешние ЗУ можно рассматривать как разновидность вводных устройств, правда, лишь для программ, уже вводившихся однажды в машину с обычного Вв. У.

§ 6. Выводные устройства

По мере того как в ходе решения задачи образуются или накапливаются данные, являющиеся конечным результатом обработки, они могут выводиться из машины. Наиболее распространенным способом вывода результатов вычислений является печать их на бумаге. Для этого необходимо *печатающее устройство*, которое по сигналам, поступающим из машины, приводит в действие механизмы, печатающие цифры, знаки или буквы в той последовательности, в какой их выдает машина.

Применяются и другие виды выводных устройств, например, для вывода на перфорацию. Отперфорированный материал может быть затем отпечатан на устройствах, не связанных непосредственно с ма-

шиной, или вновь введен в ту же или другую машину через вводное устройство.

Роль выводного устройства могут выполнять и внешние ЗУ. Вывод при этом делается либо для хранения данных во время решения других задач, либо для накопления данных, с тем чтобы со временем они могли быть обработаны другой программой, либо для передачи данных на другую машину.

Одна ЭВМ может иметь несколько разных типов выводных и вводных устройств и по несколько экземпляров устройств каждого типа. Этим обеспечивается возможность ввода данных различного происхождения через разные устройства, а также вывод данных разного назначения с помощью разных устройств вывода и на разные носители. Например, программа обработки данных какого-либо эксперимента может быть введена с перфокарт, а сами экспериментальные данные — с перфолент или даже непосредственно от прибора, участвующего в эксперименте. Результаты обработки эксперимента с необходимыми пояснениями могут быть отпечатаны на специальном печатающем устройстве широкого формата в виде таблиц и текста. Текущий протокол испытаний может быть отпечатан на электрической пишущей машинке, управляемой ЭВМ. Часть обрабатываемых данных можно записать на магнитную ленту, с тем чтобы впоследствии подвергнуть их обработке по другой программе. Машина может по ходу обработки подавать управляющие сигналы на прибор и воздействовать тем самым на ход эксперимента. Некоторые данные могут быть выведены на экран электронно-лучевой трубки, чтобы оператор машины мог следить за ходом эксперимента и т. п.

Глава 2. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ

§ 1. Позиционные системы счисления

Обычная система записи чисел, которой мы привыкли пользоваться в повседневной жизни, представляет собой так называемую *позиционную* систему. В позиционных системах один и тот же знак (цифра) может выражать различные количества в зависимости от его места в записи числа по отношению к запятой (или к правому концу записи числа, если оно — целое). Так, например, в числе 454 первая четверка выражает собой четыре сотни, а вторая — четыре единицы. Пятерка в этом числе означает пять десятков, а в числе 0,185 — пять тысячных долей единицы. Позиционная система записи чисел очень удобна и экономична не только для записи чисел, но и для выполнения над ними арифметических действий. В этом легко убедиться, сравнив ее, например, с записью тех же чисел словами или римскими цифрами. Удобна она не только для записи чисел знаками на бумаге, но и для механического представления чисел. Вспомним, например, счеты. Каждому разряду числа (единицам, десяткам, сотням, тысячам и т. д., а также десятым и сотым долям единицы) на счетах соответствует своя проволока. Костяшки на этой проволоке могут занимать десять различных положений (одиннадцатое положение — когда все десять косточек находятся с левой стороны — допускается лишь в середине вычислений, а в конце их является запретным: все десять косточек должны быть переброшены направо, а на следующей проволоке одна косточка переброшена справа налево). Возможны и применяются на практике и другие способы физического представления десятичных чисел: с помощью нескольких колес, каждое из которых может фиксироваться в одном из десяти возможных положений; перфокарт, в каждой из вертикальных колонок которых может пробиваться отверстие на одном из десяти уровней по высоте и т. п. Общим для всех этих представлений является то, что некоторый физический носитель состоит из некоторого числа n однородных элементов (проволок с костями, колес, вертикальных колонок перфокарты или мест по отношению к запятой), каждый из которых может находиться в одном из десяти состояний. Любая такая система пригодна для записи

или изображения 10^n различных чисел. Это могут быть числа из разных диапазонов. Например, при $n = 5$ это могут быть либо целые числа от нуля (00000) до 99999, либо правильные десятичные дроби от нуля (0,00000) до 0,99999, либо числа арифметической прогрессии 0; 0,01; 0,02; ...; 999,99. Все зависит от того, где мы условимся в данном числе помещать запятую. Если же снабдить нашу систему еще одним элементом, показывающим положение запятой в числе, то диапазон чисел, изображаемых такой системой, может быть очень сильно расширен. Так, если к системе добавлен шестой элемент, также изображающий одну из цифр от 0 до 9, обозначающую, после какого знака в числе, изображаемом остальными пятью элементами, стоит запятая, то такая система способна изобразить (кроме нуля) все десятичные числа, имеющие не более пяти значащих цифр, от 0,00001 до 999990000. (Первое из написанных чисел

изобразится состоянием системы

0	0	0	0	0	1	0

, второе — состоянием

9	9	9	9	9	9

, число 207,36 — состоянием

2	0	7	3	6	3

, нуль — состоянием

0	0	0	0	0	*

, где

значение последнего элемента произвольно, и т. п.) Такой способ изображения чисел эквивалентен записи числа в виде: $q \cdot 10^p$ — где q — правильная десятичная дробь, имеющая 5 знаков (в общем случае — n знаков) после запятой, p — целое число, лежащее в пределах от 0 до 9 (в общем случае эти пределы могут быть другими).

Изложенные принципы изображения чисел применяются и в ЭВМ. Каждый регистр арифметического устройства, каждая ячейка памяти представляет собой физическую систему, состоящую из некоторого числа однородных элементов, каждый из которых способен находиться в нескольких состояниях. Каждый такой элемент служит для изображения одного из разрядов числа и поэтому обычно называется *разрядом* данного регистра или ячейки.

При использовании обычной, десятичной, системы счисления каждый разряд машины должен иметь возможность принимать 10 различных устойчивых состояний. Такие машины существуют. Однако при этом конструкция элементов машины оказывается сложной. Поэтому значительно более распространены машины, работающие в самой простой системе счисления — двоичной.

Наша десятичная система счисления характеризуется тем, что в ней десять единиц какого-либо разряда образуют единицу следующего старшего разряда. Другими словами, единицы различных разрядов представляют собой различные степени числа десять. Число десять называется *основанием* этой системы счисления.

В системе счисления с основанием m (m -ичной системе счисления) единицами разрядов служат последовательные степени числа m , иначе говоря, m единиц какого-либо разряда образуют единицу следующего разряда. Для записи чисел в m -ичной системе счисле-

ния требуется m различных цифр, изображающих числа $0, 1, \dots, m-1$. Запись числа m в m -ичной системе счисления имеет вид 10.

Для примера выпишем начало натурального ряда чисел в десятичной (верхняя строка) и четверичной (нижняя строка) системах счисления:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
1	2	3	10	11	12	13	20	21	22	23	30	31	32	33	100	101	102	103	110	...

Десятичное число $64 = 4^3$ запишется в четверичной системе в виде 1000; десятичное число $100 = 4^3 + 2 \cdot 4^2 + 4^1$ изобразится в виде: 1210 и т. п.

Аналогично десятичным дробям в m -ичной системе счисления могут применяться m -ичные дроби, в которых единица первого разряда после запятой изображает число $1/m$, единица второго разряда после запятой — число $1/m^2$ и вообще единица i -го разряда после запятой — число $1/m^i$. Так, например, в четверичной системе запись 0,302 изображает число $\frac{3}{4} + \frac{0}{4^2} + \frac{2}{4^3} = \frac{25}{32}$, а запись 23,11 — число $2 \cdot 4^1 + 3 \cdot 4^0 + \frac{1}{4} + \frac{1}{4^2} = 11 \frac{5}{16}$.

Из всех систем счисления особенно проста и поэтому интересна двоичная система счисления. В ней для записи чисел используются всего две цифры: 0 и 1. Запись 10 означает число 2. Для иллюстрации выпишем в двоичной системе начало натурального ряда чисел (верхняя строка — в десятичной системе):

1	2	3	4	5	6	7	8	9	10	11	12	13
1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101
14	15	16	17	18	19	20	21	22	...			
1110	1111	10 000	10 001	10 010	10 011	10 100	10 101	10 110	...			

Некоторым неудобством двоичной системы является бросающийся в глаза быстрый рост числа разрядов, необходимых для записи чисел. В остальном же двоичная система очень удобна, прежде всего — минимальным числом различных цифр. Для сложения и

вычитания чисел достаточно помнить, что $0 + 0 = 0$; $0 + 1 = 1$; $1 + 1 = 10$.

Таблицы умножения в двоичной системе вообще не существует, если не считать тривиальных равенств: $0 \times 0 = 0$; $0 \times 1 = 0$; $1 \times 1 = 1$. Поэтому умножение сводится лишь к сдвигам множимого и сложениям.

Например:

$$\begin{array}{r} \times 111 \\ \hline 101 \\ 111 \\ \hline 111 \\ \hline 100011 \end{array} \quad \begin{array}{l} (2^2 + 2^1 + 2^0 = 7) \\ (2^2 + 2^0 = 5) \\ (2^5 + 2^1 + 2^0 = 35) \end{array}$$

При делении цифры частного определяются без всякого труда, простым сравнением делителя с остатком.

Например:

$$\begin{array}{r} 101101 \\ \hline 1001 \\ \hline 1001 \\ \hline 1001 \\ \hline 0 \end{array} \quad \begin{array}{l} 1001 \\ \hline 101 \end{array} \quad \begin{array}{l} (2^5 + 2^3 + 2^2 + 2^0 = 45; \quad 2^3 + 2^0 = 9) \\ (2^2 + 2^0 = 5) \end{array}$$

или:

$$\begin{array}{r} 1001101 \\ \hline 111 \\ \hline 1010 \\ \hline 111 \\ \hline 111 \\ \hline 111 \\ \hline 0 \end{array} \quad \begin{array}{l} 111 \\ \hline 1011 \end{array} \quad \begin{array}{l} (2^6 + 2^3 + 2^2 + 2^0 = 77; \quad 2^2 + 2^1 + 2^0 = 7) \\ (2^3 + 2^1 + 2^0 = 11) \end{array}$$

В любой системе счисления существует *нормализованная запись чисел*. Нормализованной называется запись числа, не равного нулю, в виде:

$$\pm m^p \cdot q$$

при условии, что p — целое число (положительное или отрицательное, или нуль), q — правильная m -ичная дробь, у которой первая цифра после запятой не равна нулю, т. е.

$$1/m \leq q < 1.$$

Так, например, в десятичной системе нормализованная запись некоторых чисел имеет вид:

$$\begin{aligned} 1 &= 10^1 \cdot 0,1 \\ 280 &= 10^3 \cdot 0,28 \\ -0,0039 &= -10^{-2} \cdot 0,39 \end{aligned}$$

Условимся в дальнейшем в записи числа x в виде $x = m^p \cdot q$ называть p *порядком* числа x , а q — *мантиссой* числа, независимо от того, является ли эта запись нормализованной или нет.

В нормализованной двоичной записи первая цифра мантиссы должна быть 1. Например:

$$\begin{aligned} 100 &= 2^{11} \cdot 0,1 \\ -11010 &= 2^{101} \cdot (-0,1101) \\ 101,011 &= 2^{11} \cdot 0,101011 \\ 0,0000101 &= 2^{-100} \cdot 0,101 \end{aligned}$$

Здесь порядки и мантиссы записаны в двоичной системе, обозначение системы (2) обозначено своей десятичной цифрой. Число нуль не может быть записано в нормализованной форме так, как она была определена. Поэтому относительно нормализованной записи нуля приходится прибегать к особым соглашениям. Например, можно условиться, что нормализованной называется запись нуля, если и мантисса и порядок равны нулю.

В дальнейшем мы будем встречаться с записями чисел в различных системах счисления. Там, где из текста неясно, в какой системе записано число, мы будем снабжать его справа внизу индексом, указывающим на основание системы счисления. Например:

$$31_{10} = 37_8 = 1111_2.$$

§ 2. Перевод чисел из одной системы счисления в другую

Рассмотрим задачу о переводе числа x , записанного в одной системе счисления, в другую. На первый взгляд, после разъяснений, сделанных в предыдущем разделе, такой перевод уже не составляет проблемы. Тем не менее, некоторые осложнения при решении этой задачи все же возникают.

Уточним постановку задачи. Обычно из всех систем счисления одна может считаться «своей», а все остальные — «чужими». Например, человеку даже трудно свыкнуться с мыслью, что запись числа в десятичной системе счисления — это еще не само число, а всего лишь некоторый текст — последовательность знаков, которую мы привыкли отождествлять с числом, забывая о том, что числа, даже целые, — это некие абстрактные понятия, что запись «3» отображает одно из таких понятий ничуть не лучше, чем «три», «III», «three» или «11», если в последнем случае использовать двоичную систему счисления. Своя система счисления — это система, в которой человек (или машина) умеет проводить вычисления. Если человеку (умеющему пользоваться одной системой) сравнительно легко освоить приемы вычислений в любой позиционной системе счисления, то для машины своя система счисления определяется ее конструкцией. Для большинства машин своей является двоичная система, хотя встречаются и исключения, например, даже машины, «умеющие» вычислять в нескольких системах счисления (при этом обычно для каждой системы используется свое арифметическое устройство).

Итак, пусть система счисления с основанием p — это своя система, в которой выполняются все вычисления, а система с основанием q — чужая. Тогда возможны две задачи: 1) для данного числа x найти его запись в системе с основанием q (перевод из своей системы в чужую) и 2) по данной записи в системе с основанием q найти изображаемое ею число (перевод из чужой системы в свою).

Когда мы говорим, что число x дано или найдено, это означает, что известна его запись в системе с основанием p . Поэтому внешне обе задачи похожи одна на другую. Но над записями в своей системе можно производить операции, отображающие абстрактные арифметические действия над числами, а тексты в чужой системе можно только расчленять на знаки, из которых они состоят, и составлять их из таких знаков. Кроме того — и это очень важно — для каждого знака α , являющегося q -ичной цифрой, необходимо уметь находить его числовой эквивалент $n(\alpha)$, и, наоборот, для каждого целого числа k из диапазона $0 \leq k < q$ требуется уметь записывать соответствующую цифру $d(k)$ в системе счисления с основанием q .

В ряде случаев задача перевода не может быть решена точно, например, десятичная дробь 0,9 не может быть точно представлена никакой конечной двоичной дробью. В таких случаях задача решается приближенно. Степень приближения либо задается при постановке задачи, либо выбирается так, чтобы возможная погрешность результирующей записи была близка к погрешности исходной записи.

После этих замечаний перейдем к изложению методов решения задач перевода. Начнем с более простой задачи 2. Пусть запись числа в системе с основанием q имеет вид

$$\alpha_r \dots \alpha_1 \alpha_0, \alpha_{-1} \dots \alpha_{-s}, \quad (1)$$

где α_j ($-s \leq j \leq r$) — q -ичные цифры. Мы знаем, что эта запись изображает число

$$x = n(\alpha_r) \cdot q^r + \dots + n(\alpha_1) \cdot q + n(\alpha_0) + n(\alpha_{-1}) \cdot q^{-1} + \dots + n(\alpha_{-s}) \cdot q^{-s}. \quad (2)$$

Выполнив все вычисления, предписанные этим выражением, мы найдем значение x . Надо лишь постараться провести эти вычисления самым экономным способом. Рассмотрим два важных частных случая.

Пусть $s = 0$, иначе говоря, пусть дана запись

$$\alpha_r \dots \alpha_1 \alpha_0,$$

изображающая целое число. Тогда значение x , являющееся членом от q степени r с коэффициентами $n(\alpha_r), \dots, n(\alpha_0)$, проще всего вычислить по следующей схеме, известной под названием

схемы Горнера. Положим

$$x_0 = n(\alpha_r) \quad (3)$$

и вычислим

$$x_j = x_{j-1}q + n(\alpha_{r-j}) \quad (4)$$

для $j = 1, \dots, r$. Ясно, что

$$x_r = n(\alpha_r) \cdot q^r + \dots + n(\alpha_1) \cdot q + n(\alpha_0)$$

— искомое значение. В этом случае задача решается точно.

Пусть $r = -1$, т. е. дана запись правильной q -ичной дроби

$$0, \alpha_{-1} \dots \alpha_{-s}. \quad (5)$$

Здесь также целесообразно воспользоваться схемой Горнера, но «с другого конца», так как требуемое значение является многочленом от q^{-1} . Положим

$$x_0 = 0$$

и вычислим

$$x_j = \frac{x_{j-1} + n(\alpha_{j-s-1})}{q} \quad (6)$$

для $j = 1, \dots, s$. Тогда

$$x_s = \frac{n(\alpha_{-1})}{q} + \dots + \frac{n(\alpha_{-s})}{q^s}$$

даст нам искомое значение. Если дробь $1/q$ может быть точно записана в виде конечной p -ичной дроби, то и значение x_j при любом j может быть точно представлено конечной дробью в p -ичной системе счисления. Однако, если исходная запись (1) — не точная, а приближенная, то ее точный перевод в p -ичную систему может оказаться бессмысленным.

Пусть, например, $q = 2$ и запись $0,1011$ представляет собой двоичную дробь, все цифры которой — верные. Погрешность этой записи не превышает половины единицы младшего разряда, т. е. числа $\frac{1}{2} \cdot 2^{-4} = \frac{1}{32} \approx 0,03_{10}$. Для записи $0,1011_2$ существует точный десятичный эквивалент — запись $0,6875_{10}$. Однако уже вторая цифра после запятой может оказаться неверной. Поэтому достаточно (даже с некоторым запасом точности) ограничиться десятичным числом $0,69$ в качестве результата перевода.

В общем случае, когда задана s -разрядная q -ичная дробь, можно предполагать, что все ее цифры точные и погрешность этой дроби не превышает $\frac{1}{2}q^{-s}$. Если результат перевода приближенно изобразить t -разрядной p -ичной дробью, то погрешность этого приближения может быть сделана не превосходящей $\frac{1}{2}p^{-t}$. Исходя из

требования равноточности обоих представлений, получаем приближенное равенство

$$p^{-t} \approx q^{-s}$$

или

$$t \approx \frac{s \cdot \log q}{\log p}$$

(логарифмы можно брать при любом основании). Чтобы обеспечить минимальный запас точности у результата перевода, в качестве t следует принять наименьшее целое число, большее или равное $s \cdot \log_p q$. Если вести вычисления по описанной выше схеме, то во всех промежуточных результатах надо сохранять не менее t значащих цифр. Но при этом могут накопиться ошибки округления. Несколько проще и, вообще говоря, точнее оказывается следующая схема. Переводим в p -ичную систему целое число, q -ичная запись которого имеет вид

$$\alpha_{-1} \dots \alpha_{-s}$$

Разделив полученный результат на целое число q^s , получим требуемое значение. Только при последнем делении потребуется произвести округление до t верных знаков, так как предшествующие вычисления выполняются над целыми числами, которые записываются в p -ичной системе менее чем t цифрами. Такая схема особенно удобна при ручном счете.

Пример 1. Перевести число $0,1011$ из двоичной системы в десятичную. Сначала переводим в десятичную систему целое число $1011_2 = ((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 = 11_{10}$. Затем делим 11 на $2^4 = 16$, сохраняя две цифры после запятой: $11/16 \approx 0,69$.

Вернемся к общему случаю, когда в p -ичную систему переводится число, представленное q -ичной записью (1). Можно отдельно перевести целую и дробную части этого числа и сложить полученные результаты. Но можно поступить и так. Переводим (точно) в p -ичную систему целое число с q -ичной записью

$$\alpha_r \dots \alpha_1 \alpha_0 \alpha_{-1} \dots \alpha_{-s}$$

Полученный результат делим на q^s , сохраняя t знаков после запятой, где t имеет прежний смысл.

Пример 2. Перевести число $3,14$ из десятичной системы в двоичную, считая двоичную систему своей. Здесь $p = 2$, $q = 10 = 1010_2$, $r = 0$, $s = 2$ и должно быть выполнено неравенство

$$t \geq 2 \frac{\log 10}{\log 2} \approx 6,6;$$

так что принимаем $t = 7$. Переводим целое число 314 в двоичную систему, учитывая, что $n(\alpha_0) = 11$, $n(\alpha_{-1}) = 1$, $n(\alpha_{-2}) = 100$:

$$(11 \cdot 1010 + 1) \cdot 1010 + 100 = 11111 \cdot 1010 + 100 = 100111010.$$

Делим полученный результат на $100_{10} = 1100100_2$ с точностью до 7-го знака после запятой:

$$\begin{array}{r}
 1001110,1 \quad | \quad 11001 \\
 \underline{11001} \quad | \quad 11,0010001 \\
 11100 \\
 \underline{11001} \\
 11100 \\
 \underline{11001} \\
 110000 \\
 \underline{11001} \\
 10111
 \end{array}$$

Так как остаток больше половины делителя, округляем результат, добавляя 1 к младшему разряду. Получаем

$$3,14_{10} \approx 11,0010010_2.$$

Пример 3. Перевести число $11,0010010_2$ из двоичной системы в десятичную. Здесь $p = 10$, $q = 2$, $r = 1$, $s = 7$ и

$$t \geq 7 \frac{\log 2}{\log 10} \approx 2,1.$$

Переводим число 110010010_2 в десятичную систему:

$$(((((((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0 = 402.$$

Делим на $2^7 = 128$ с тремя знаками после запятой:

$$\frac{402}{128} = 3,141.$$

Обратимся теперь к задаче 1 — переводу из своей системы в чужую. Здесь те же вычисления, которые выполнялись при переводе из чужой системы в свою, надо выполнять в обратном порядке.

Снова начнем с перевода целых чисел. Пусть дано целое число x . Примем его за x_j в формуле (4). Эта формула служила для определения x_j по x_{j-1} и α_{r-j} . Сейчас требуется, наоборот, определить x_{j-1} и α_{r-j} по x_j . Поскольку $0 \leq n(\alpha_{r-j}) < q$, ясно, что x_{j-1} является частным, а $k = n(\alpha_{r-j})$ — остатком от деления x_j на q . Здесь число цифр в записи x в системе с основанием q нам заранее неизвестно, так что нумерацию чисел $x_r, \dots, x_0$ естественно заменить на обратную. Приходим к следующей схеме вычислений.

Полагаем $x_0 = x$. Если очередное x_j меньше q , то полагаем на основании формулы (3) $r = j$ и $\alpha_r = d(x_j)$ и заканчиваем на этом перевод. В противном случае находим частное x_{j+1} и остаток k_j от деления x_j на q . Полагаем $\alpha_j = d(k_j)$ и переходим к рассмотрению x_{j+1} . Искомая запись числа x в системе с основанием q имеет вид

$$\alpha_r \dots \alpha_0.$$

Пример 4. Перевести число $x = 1973$ из своей десятичной системы счисления в восьмеричную. Последовательно находим

$$\begin{aligned}x_0 &= 1973 = 246 \cdot 8 + 5, & k_0 &= 5, & \alpha_0 &= \langle 5 \rangle \text{ (восьмеричная цифра);} \\x_1 &= 246 = 30 \cdot 8 + 6, & k_1 &= 6, & \alpha_1 &= \langle 6 \rangle; \\x_2 &= 30 = 3 \cdot 8 + 6, & k_2 &= 6, & \alpha_2 &= \langle 6 \rangle; \\x_3 &= 3 < 8, & & & \alpha_3 &= \langle 3 \rangle.\end{aligned}$$

Следовательно, $1973_{10} = 3665_8$.

Рассмотрим перевод правильной дроби из своей системы в чужую. Здесь требуется заранее определить достаточное для сохранения минимального запаса точности число цифр результата. Очевидно, это — наименьшее целое, удовлетворяющее неравенству

$$s \geq t \frac{\log p}{\log q}, \quad (7)$$

где t — число верных p -ичных цифр данной дроби. Сами вычисления получаются обращением вычислений по формуле (6), где при любом j числа x_j и x_{j-1} — правильные дроби, а $n(\alpha_{j-s-1})$ — целое. Ясно, что $n(\alpha_{j-s-1})$ представляет собой целую часть, а x_{j-1} — дробную часть произведения $x_j q$. Здесь также меняем нумерацию чисел $x_s, \dots, x_1$ на обратную и приходим к следующей совокупности правил перевода правильной дроби x из своей системы в чужую.

Полагаем $\alpha_0 = \langle 0 \rangle$ (q -ичная цифра) и $x_0 = x$. Далее для $j = -1, -2, \dots$ проделываем следующие вычисления. Находим целую часть k_j и дробную часть x_j произведения $x_{j+1}q$ и полагаем $\alpha_j = d(k_j)$ (напомним, что так мы обозначаем q -ичную цифру, изображающую целое число $k_j < q$). Если номер удовлетворяет неравенству $|j| \geq t/\log_p q$, то перевод закончен и $s = j$, в противном случае переходим к следующему значению j . Результатом перевода является q -ичная запись

$$\alpha_0, \alpha_{-1} \dots \alpha_{-s}.$$

Недостаток этого метода в том, что погрешность результата может оказаться больше $\frac{1}{2} q^{-s}$ (если $x_s > 1/2$). Для исправления этого недостатка следует при $x_s > 1/2$ добавить к полученному результату число q^{-s} . Однако это сложение придется выполнять в чужой системе счисления. Чтобы не иметь дела с вычислениями в чужой системе, можно определить s в начале перевода, принять в качестве k_0 целую часть, а в качестве x_0 дробную часть числа $x + \frac{1}{2} q^{-s}$, положить $\alpha_0 = d(k_0)$ (α_0 может оказаться равным $\langle 1 \rangle$, а не только $\langle 0 \rangle$), а затем определить цифры $\alpha_{-1}, \dots, \alpha_{-s}$ описанным выше способом.

В общем случае, когда переводится произвольное число x , можно поступить, например, так. Определить из соотношения (7)

желаемое число s цифр дробной части после перевода, добавить к x число $\frac{1}{2}q^{-s}$, перевести описанным выше способом отдельно целую и дробную часть полученной суммы и выписать в качестве результата последовательность знаков

$$\alpha_r \dots \alpha_0, \alpha_{-1} \dots \alpha_{-s}.$$

Пример 5. Перевести число 365,2422 из десятичной системы в шестнадцатеричную. Для обозначения чисел 10, 11, 12, 13, 14 и 15 в шестнадцатеричной записи использовать знаки $\bar{0}$, $\bar{1}$, $\bar{2}$, $\bar{3}$, $\bar{4}$ и $\bar{5}$. Здесь $p = 10$, $q = 16$, $t = 4$ и для сохранения точности s должно удовлетворять неравенству

$$s \geq 4/\log_{10} 16 \approx 3,3.$$

Полагаем $s = 3$, т. е. идем на некоторую потерю точности. Переводим целую часть числа $365,2422 + \frac{1}{2} \cdot 16^{-3} \approx 365,24232$ в шестнадцатеричную систему:

$$\begin{aligned} x_0 &= 365 = 22 \cdot 16 + 13, & k_0 &= 13, & \alpha_0 &= \langle \bar{3} \rangle; \\ x_1 &= 22 = 1 \cdot 16 + 6, & k_1 &= 6, & \alpha_1 &= \langle \bar{6} \rangle; \\ x_2 &= 1 < 16, & & & \alpha_2 &= \langle \bar{1} \rangle. \end{aligned}$$

Переводим правильную дробь 0,24232 в шестнадцатеричную систему:

$$\begin{aligned} x_0 &= 0,24232, \\ 16x_0 &= 3,8771, & x_{-1} &= 0,8771, & \alpha_{-1} &= \langle \bar{3} \rangle; \\ 16x_{-1} &= 14,034, & x_{-2} &= 0,034, & \alpha_{-2} &= \langle \bar{4} \rangle; \\ 16x_{-2} &= 0,54, & x_{-3} &= 0,54, & \alpha_{-3} &= \langle \bar{0} \rangle. \end{aligned}$$

Результат перевода:

$$365,2422_{10} \approx 16\bar{3},3\bar{4}\bar{0}_{16}$$

Возможны и иные способы перевода чисел в чужую систему счисления. Можно, например, определив s из соотношения (7), умножить данное число x на q^s , округлить произведение до ближайшего целого числа, перевести его в систему с основанием q и в полученной последовательности q -ичных цифр отделить запятой s цифр справа.

Пример 6. Перевести число 3,14 из (своей) десятичной системы в двоичную. Здесь $p = 10$, $q = 2$, $t = 2$, так что должно быть

$$s \geq 2/\log_{10} 2 \approx 6,6.$$

Принимаем $s = 7$ и умножаем $3,14$ на $2^7 = 128$. Округленный результат 402_{10} переводим в двоичную систему:

$$\begin{array}{ll} x_0 = 402 = 201 \cdot 2, & \alpha_0 = \langle 0 \rangle; \\ x_1 = 201 = 100 \cdot 2 + 1, & \alpha_1 = \langle 1 \rangle; \\ x_2 = 100 = 50 \cdot 2, & \alpha_2 = \langle 0 \rangle; \\ x_3 = 50 = 25 \cdot 2, & \alpha_3 = \langle 0 \rangle; \\ x_4 = 25 = 12 \cdot 2 + 1, & \alpha_4 = \langle 1 \rangle; \\ x_5 = 12 = 6 \cdot 2, & \alpha_5 = \langle 0 \rangle; \\ x_6 = 6 = 3 \cdot 2, & \alpha_6 = \langle 0 \rangle; \\ x_7 = 3 = 1 \cdot 2 + 1, & \alpha_7 = \langle 1 \rangle; \\ x_8 = 1 < 2, & \alpha_8 = \langle 1 \rangle. \end{array}$$

Итак, $402_{10} = 110010010_2$ и $3,14_{10} = 11,0010010_2$. Полезно сопоставить этот пример с примером 2, где решается та же задача, но роли систем иные, и примером 3, где все действия заменены обратными и выполняются в обратной последовательности.

§ 3. Представление чисел в машине

Итак, наиболее обычной для представления чисел в машине является двоичная система счисления. Применение этой системы позволяет использовать в качестве разрядов элементы, имеющие всего два устойчивых состояния. Одно из них служит для изображения единицы в соответствующем разряде числа, другое — для изображения нуля. Такие элементы просты по устройству и надежны в работе. Точно такие же элементы можно использовать для представления знаков чисел. Обычно состояние элемента, служащее для представления нуля, используется в регистре знака для изображения знака $+$, а состояние, изображающее единицу, — для представления знака $-$. Из-за простоты выполнения арифметических действий в двоичной системе применение этой системы позволяет добиться простейшей схемы арифметического устройства.

Независимо от системы счисления существуют два основных типа представления чисел, называемые представлениями с *фиксированной* и с *плавающей запятой*. В машинах, использующих представление чисел с фиксированной запятой, все разряды регистра или ячейки, кроме разряда знака числа, служат для изображения разрядов чисел, причем каждому разряду регистра соответствует определенный, всегда один и тот же разряд числа. Такая система упрощает выполнение арифметических действий, но сильно ограничивает диапазон чисел, которые могут быть представлены в машине. Чаще всего это бывает диапазон $-1 < x < 1$. Этот случай соответствует соглашению, что при чтении чисел, записанных в машине, запятая ставится непосредственно перед старшим цифровым разрядом.

Другими словами, если число цифровых разрядов машины равно n , основание системы счисления, используемой в машине, равно m , то состоянию регистра (или ячейки) машины

$\pm$	α_1	α_2	$\dots$	α_n

соответствует число

$$\pm \left(\frac{\alpha_1}{m} + \frac{\alpha_2}{m^2} + \dots + \frac{\alpha_n}{m^n} \right).$$

Наибольшее по абсолютной величине число, которое может быть представлено в машине, равно $1 - m^{-n}$, наименьшее, отличное от нуля, — m^{-n} . Для представления чисел, не укладывающихся в этот диапазон, программист должен вводить *масштабные множители*, т. е. заменять истинные величины, участвующие в решении задачи, их произведениями на коэффициенты, подобранные так, чтобы не выйти за пределы диапазона чисел, представимых в машине. Для различных величин эти коэффициенты могут оказаться различными. Более того, эти коэффициенты, возможно, потребуется менять по ходу вычислений. Все это заметно усложняет составление программы и саму программу.

Другой недостаток представления чисел с фиксированной запятой состоит в том, что числа, малые по абсолютной величине, могут быть представлены со значительно меньшим количеством значащих цифр, т. е. с меньшей относительной точностью, чем числа, близкие к единице. Масштабные множители должны поэтому по возможности приближать числа, реально участвующие в вычислениях, к верхнему допустимому пределу.

Чтобы избавиться от этих недостатков, в современных электронных вычислительных машинах высокого класса возможно представление чисел с плавающей запятой. При этом представлении часть разрядов отводится для записи порядка числа, остальные разряды — для записи мантииссы. По одному разряду в каждой группе служит для изображения знака порядка и мантииссы.

Например, можно представить себе такое распределение разрядов регистра (или ячейки памяти):

s_p	s_q	β_k	β_{k-1}	$\dots$	β_2	β_1	α_1	α_2	$\dots$	α_{n-1}	α_n

Первые два разряда — двоичные (т. е. могут принимать одно из двух состояний каждый) и служат для изображения знаков порядка и мантииссы. Следующие k разрядов используются для изображения абсолютной величины порядка числа, остальные n разрядов — для изображения абсолютной величины мантииссы.

Тогда изображенному на схеме состоянию регистра соответствует число

$$(-1)^s m^p \left(\frac{\alpha_1}{m} + \frac{\alpha_2}{m^2} + \dots + \frac{\alpha_n}{m^n} \right),$$

где

$$p = (-1)^s p (\beta_k m^{k-1} + \beta_{k-1} m^{k-2} + \dots + \beta_2 m + \beta_1).$$

При такой системе записи наибольшее по абсолютной величине число, которое может быть представлено в машине, равно

$$m^{m^k-1} (1 - m^{-n}),$$

а наименьшее по абсолютной величине число, отличное от нуля, равно

$$m^{-(m^k-1)} \cdot m^{-n} = m^{-m^k-n+1},$$

если допускать использование ненормализованных чисел, и равно

$$m^{-(m^k-1)} \cdot m^{-1} = m^{-m^k},$$

если пользоваться лишь нормализованными числами.

Обычно в машинах с плавающей запятой применение ненормализованной записи чисел допускается, но не рекомендуется, так как некоторые операции над ненормализованными числами машина либо выполняет неверно, либо не может выполнить вообще (происходит автоматический останов).

§ 4. Выполнение арифметических операций в машине

Использование в машине представления чисел с плавающей запятой усложняет схему арифметического устройства. При сложении и вычитании чисел нужно производить подготовительную операцию, называемую *выравниванием порядков*. Она состоит в том, что мантисса числа с меньшим порядком сдвигается в своем регистре в сторону младших разрядов на количество разрядов, равное разности порядков данных чисел. После этой операции одноименные разряды чисел оказываются расположенными в соответствующих разрядах обоих регистров и можно производить сложение или вычитание мантисс.

Поясним эту операцию примером (предполагая, что в ячейках памяти можно разместить один разряд порядка и пять разрядов мантиссы):

$$+ \frac{10^2 \cdot 0,23619}{10^{-1} \cdot 0,71824}$$

Перед сложением второе слагаемое нужно преобразовать к виду: $10^2 \cdot 0,00071824$, после чего можно приступить к сложению:

$$+ \frac{10^2 \cdot 0,23619}{10^2 \cdot 0,00071824} \\ \hline 10^2 \cdot 0,23690824$$

Результат получили с большим количеством разрядов, чем вмещает ячейка, поэтому он округляется и записывается в память в виде:

$$10^2 \cdot 0,23691.$$

При умножении двух чисел с плавающей запятой их порядки необходимо сложить, а мантииссы — перемножить. При делении из порядка делимого надо вычесть порядок делителя, а мантииссу делимого разделить на мантииссу делителя.

После операций над порядками и мантииссами мы получаем, вообще говоря, порядок и мантииссу результата, но последняя может не удовлетворять ограничениям, накладываемым на мантииссы нормализованных чисел. Так как от результата арифметических операций в машине обычно требуется, чтобы он был нормализованным числом, необходимо дополнительное преобразование результата, называемое его *нормализацией*. В зависимости от величины полученной мантииссы результата, она сдвигается вправо или влево так, чтобы ее первая значащая цифра попала в первый разряд после запятой. Одновременно порядок результата увеличивается или уменьшается на величину, равную числу разрядов сдвига.

Более подробно на методике выполнения арифметических операций мы пока останавливаться не будем.

Заметим, что над мантииссами в арифметическом устройстве должны выполняться все четыре арифметических действия, а также операция сдвига, тогда как над порядками производятся только действия сложения и вычитания.

Остановимся теперь на вопросе о представлении в машине отрицательных чисел.

Наиболее естественным кажется помещать в разряды цифровой части числа его абсолютную величину, а в разряде знака ставить соответствующий код знака. Такой способ представления отрицательных чисел называется *прямым кодом*.

Но этот способ не единствен даже в обычной математической практике. Вспомним, например, как принято записывать в таблицах логарифмов отрицательные логарифмы, в частности, логарифмы тригонометрических функций. Обычно такие значения логарифма увеличиваются на 10 единиц.

Вместо $-0,27589$ в таблицу помещается значение $9,72411$, вместо $-1,41630$ — значение $8,58370$ и т. п. Такой способ записи отрицательных чисел называется *дополнительным кодом* (в данном случае — по модулю 10). Разумеется, он применим лишь тогда, когда точно известно, в каком диапазоне лежит так представленное число (или отрицательные числа снабжаются особой пометкой).

Применяется этот способ довольно часто и в электронных вычислительных машинах. Особенно часто — для записи порядков.

Рассмотрим, например, случай, когда для записи порядков в машине, работающей в двоичной системе, отводится шесть разрядов: один для знака и пять — для цифр порядка. Положительные порядки будут изображаться кодами с цифрой 0 (код знака +) в старшем разряде, например:

$$\begin{array}{r} +1 \quad 0\,00001 \\ +5 \quad 0\,00101 \\ +26 \quad 0\,11010 \end{array}$$

Отрицательные порядки в дополнительном коде будут выражаться так:

$$\begin{array}{r} -1 \quad 1\,11111 \\ -5 \quad 1\,11011 \\ -26 \quad 1\,00110 \end{array}$$

Если все эти шесть цифр прочесть как целое число в двоичной записи, то получим положительное число, отличающееся от закодированного им порядка на 2^6 . Иначе говоря, в данном случае дополнительный код берется *по модулю* 2^6 .

Удобство дополнительного кода (то же самое, что и при записи логарифмов) состоит в том, что при сложении порядков разных знаков не нужно менять характер действия (т. е. переходить от сложения к вычитанию абсолютных величин слагаемых). Достаточно сложить коды порядков, например:

$$\begin{array}{r} + +10 \quad 0\,01010 \\ + -3 \quad 1\,11101 \\ \hline + 7 \quad (1)0\,00111 \end{array}$$

или

$$\begin{array}{r} + +11 \quad 0\,01011 \\ + -27 \quad 1\,00101 \\ \hline -16 \quad 1\,10000 \end{array}$$

Единица в седьмом (старшем) разряде, получившаяся в первом примере, разумеется, должна быть отброшена. В машине это происходит автоматически, так как эта единица не сохраняется в разрядах, отведенных для записи порядков с их знаками.

При сложении двух положительных порядков может образоваться число, кодирующее отрицательный порядок, например:

$$\begin{array}{r} +18 \quad 0\,10010 \\ +21 \quad 0\,10101 \\ \hline +39 \quad 1\,00111 \end{array}$$

Машина должна «уметь» распознавать такие случаи. Это достигается простым приемом, но мы не будем касаться этого вопроса.

Еще один способ изображения отрицательных чисел в машине — это запись их в *обратном* коде, получающемся заменой цифры

каждого разряда прямого кода на ее дополнение до $m - 1$, где m — основание системы счисления.

Например, для рассмотренных уже отрицательных порядков в двоичной системе обратные коды будут:

— 1	1 11110
— 5	1 11010
—26	1 00101

Целое число, изображаемое этими шестью разрядами, на единицу меньше числа, получающегося при дополнительном коде. Иначе говоря, обратный код представляет собой дополнительный код по модулю ($2^6 - 1$).

Удобство обратного кода заключается в простоте его получения из прямого, особенно в двоичной системе: для изменения знака числа нужно произвести замену всех нулей единицами, а единиц — нулями. Этот прием, называемый *инверсией* кода, используется при вычитании чисел, которое заменяется сложением уменьшаемого с инверсным кодом вычитаемого. Дополнительная единица в младшем разряде, необходимая для перехода от обратного кода к дополнительному, учитывается отдельно.

С особенностями представления чисел в машине, изложенными выше, программисту приходится сталкиваться сравнительно редко, главным образом, при чтении чисел на контрольных регистрах пульта управления. Однако программисту высокой квалификации необходимо точно знать не только способ представления чисел, но и методы выполнения арифметических и иных операций в машине, все встречающиеся при этом частные и особые случаи, применяемые в машине блокировки от получения неправильных результатов, и т. п. Все эти сведения можно найти в описаниях машин.

Глава 3. КОМАНДЫ И ИХ ПРЕДСТАВЛЕНИЕ В МАШИНЕ

Команда — это инструкция, приказ машине о выполнении одной операции. Как уже говорилось при описании устройства управления, в команде должны быть в самом общем случае указаны: вид операции, место хранения в машине исходного материала, над которым производится операция, место, куда должен быть направлен результат операции, а также следующая команда, к которой машина должна перейти после выполнения данной команды.

В случае, если речь идет об арифметической операции, исходным материалом для нее служат два числа, точнее, коды этих чисел, результатом операции является также код некоторого числа. Если операнды хранятся в оперативном запоминающем устройстве, результат должен быть направлен в это же устройство и следующая команда должна быть также вызвана из некоторой ячейки ОЗУ, то запись команды должна содержать *код операции* и четыре *адреса*: два адреса операндов, адрес результата и адрес следующей команды. Однако большей частью необходимое число адресов меньше четырех. Как правило, машина, выполнив команду, хранившуюся в некоторой ячейке памяти, следующей выполняет команду из соседней ячейки — с номером, на единицу большим. В таком случае необходимыми оказываются лишь три адреса. Поэтому трехадресные машины (точнее: машины с *трехадресными командами*) довольно широко распространены.

Упомянутое выше значение адресов: два адреса для вызова (считывания) операндов операции и третий адрес для отсылки (записи) результата — остается справедливым не только для арифметических, но и для ряда других операций. Наряду с этим существуют операции с иным значением адресов. Речь о них пойдет ниже.

Есть машины с двухадресными и одноадресными командами.

При *двухадресной* системе адреса могут иметь одно из следующих значений:

а) По обоим адресам вызываются операнды выполняемой операции, результат операции засылается по одному из этих же адресов, например, по второму. Такой режим использования адресов возможен потому, что во многих случаях один из операндов, исполь-

зованных в данной операции, уже не будет нужен в последующих операциях.

б) По обоим адресам вызываются операнды, результат сохраняется на регистре результата.

в) Один из операндов для выполнения операции сохранился на регистре результата от предыдущей операции, второй операнд вызывается из ОЗУ по первому адресу, результат засылается в ОЗУ по второму адресу.

Возможен, наконец, и такой вариант, когда один из операндов сохранился на регистре результата, а второй вызывается по одному из адресов, результат же операции никуда не отсылается и сохраняется лишь на регистре результата. В этом варианте команда содержит лишь один рабочий адрес, второй остается свободным. Могут оставаться свободные адреса и в трехадресных командах.

Последний вариант использования адресов (с одним рабочим адресом) для машин с двухадресными командами является побочным — одной из возможных модификаций команд. Для машин же с *одноадресными* командами такой режим является единственно возможным для операций с двумя операндами. В этих машинах должны существовать операции, осуществляющие лишь пересылку кодов: чтение кода по заданному адресу на регистр результата (сумматор) и запись кода с регистра результата в ЗУ по адресу, указанному в команде.

В дальнейшем мы будем разбирать методы программирования только для машин с трехадресными командами.

Каждому виду операции, который может выполнять машина, присваивается некоторый номер: целое число. Этот номер называется *кодом операции* (КОП).

Таким образом, код команды состоит из выписанных подряд чисел: кода операции и адресов и имеет вид:

КОП	A_1	...	A_s

где s — число адресов в команде.

Группа чисел: $A_1, \dots, A_s$ называется *адресной частью* команды.

Код операции и адреса записываются в той же системе счисления, в какой работает машина, т. е. в той системе, в которой изображаются числа в машине и выполняются действия над ними.

В трехадресных машинах длина (количество разрядов) кода команды и кода числа обычно одинаковы и совпадают с длиной ячейки. При этом один и тот же набор цифр может быть прочитан и как число, и как команда. Благодаря этому оба вида кодов могут размещаться в любых ячейках или регистрах машины. Любая операция в машине принципиально может быть выполнена как над кодами чисел, так и над кодами команд. Эта особенность представ-

ления чисел и команд в электронных счетных машинах оказывается очень важной и во многих случаях сильно расширяет возможности машины.

Необходимость операций над кодами команд будет ясна из дальнейшего. Практически ни одна сколь-нибудь сложная программа не может быть составлена без использования операций над командами.

Для того чтобы операции над числами, т. е. арифметические операции, применять к кодам команд, и наоборот — операции, предназначенные для преобразования кодов команд, применять к кодам чисел, программист должен отчетливо представлять себе, как код команды в данной машине прочесть в виде числа, а код числа — в виде команды. К сожалению, возможности конкретного представления чисел и команд в машинах столь разнообразны, что общих правил здесь не существует.

В тех машинах, где числа изображаются с плавающей запятой, очень часто мантисса числа размещается в тех же разрядах, что и адресная часть команды, а порядок числа — в разрядах кода операции. Разряды знака числа и знака порядка могут при записи команд входить либо в адресную часть, либо в код операции команд. Возможно и совсем другое распределение разрядов.

Приведем для иллюстрации сказанного два примера. Пусть ячейка машины содержит 18 двоичных разрядов. Машина может оперировать с числами с фиксированной запятой, лежащими в пределах от $-(1-2^{-17})$ до $+(1-2^{-17})$. Для записи цифровой части числа x используются 17 разрядов, кроме самого левого (старшего), служащего для изображения знака числа («+» изображается цифрой 0 в этом разряде, «-» — цифрой 1). Система команд в машине — одноадресная. Оперативное ЗУ машины имеет 4096 ячеек, занумерованных двоичными числами от 000 000 000 000 до 111 111 111 111. Количество различных операций в машине — 64. Коды этих операций — целые двоичные числа от 000 000 до 111 111. При записи команд код операции и адрес выписываются подряд, причем код операции K занимает старшие 6 разрядов, а адрес A — младшие 12 разрядов ячейки. Итак, вид кода команды: $\boxed{K | A}$. Как прочесть этот код, если считать его кодом числа?

Разряд единиц кода операции соответствует разряду 2^{-5} кода числа, а разряд единиц адреса — разряду 2^{-17} кода числа. В цифровую часть числа входят лишь младшие 5 разрядов кода операции, старший разряд кода операции занимает разряд знака числа, причем число оказывается отрицательным, если код операции больше или равен $32_{10} = 100\,000_2$, и положительным при коде операции, меньшем 32_{10} .

Все сказанное можно записать формулой:

$$x = \begin{cases} K \cdot 2^{-5} + A \cdot 2^{-17} & \text{при } K < 32, \\ -((K - 32) \cdot 2^{-5} + A \cdot 2^{-17}) & \text{при } K \geq 32, \end{cases}$$

или

$$x = -\operatorname{sgn}(K - 32) \{ (K \cdot 2^{-5}) + A \cdot 2^{-17} \},$$

если воспользоваться обозначениями: $\operatorname{sgn} m$ для функции

$$\operatorname{sgn} m = \begin{cases} +1 & \text{при } m \geq 0, \\ -1 & \text{при } m < 0, \end{cases}$$

$\{m\}$ — для дробной части числа m , т. е. такого числа, которое лежит в пределах $0 \leq \{m\} < 1$ и отличается от m на целое число.

Предположим, что мы имеем код некоторой команды

K	A

и хотим получить код другой команды с тем же кодом операции, но с адресом, увеличенным на некоторое целое число n :

K	A + n

Числа, соответствующие этим командам, будут

$$x = -\operatorname{sgn}(K - 32) (\{K \cdot 2^{-5}\} + A \cdot 2^{-17})$$

и

$$x_1 = -\operatorname{sgn}(K - 32) (\{K \cdot 2^{-5}\} + (A + n) \cdot 2^{-17}).$$

Они различаются между собой на величину:

$$x_1 - x = -\operatorname{sgn}(K - 32) \cdot n \cdot 2^{-17}.$$

Это число можно прочесть как команду с кодом операции 0 (000 000 в двоичной системе) и адресом n , если исходная команда имела код операции $K < 32$, и как команду с кодом операции 32 и адресом n , если $K \geq 32$. Таким образом, для преобразования команды $\begin{bmatrix} K & A \end{bmatrix}$ к виду $\begin{bmatrix} K & A + n \end{bmatrix}$ нужно «сложить» ее с кодом

$\begin{bmatrix} 0 & n \end{bmatrix}$ или с кодом $\begin{bmatrix} 32 & n \end{bmatrix}$ в зависимости от величины K . «Сложение» команд понимается здесь в том смысле, что складываются числа, изображаемые теми же кодами. Важно то, что для преобразования кодов команд можно воспользоваться операцией сложения чисел. Однако уже в этом простом примере мы столкнулись с той трудностью, что для изменения адреса команды на одну и ту же величину n нужно «складывать» ее с различными кодами в зависимости от кода операции преобразуемой команды. Приходится различать команды на «положительные» (с кодом операции $K < 32$) и «отрицательные» (с кодом операции $K \geq 32$) и использовать для их преобразования коды соответствующего знака (код $\begin{bmatrix} 0 & n \end{bmatrix}$ «положительный», $\begin{bmatrix} 32 & n \end{bmatrix}$ — «отрицательный») или разные арифметические действия:

«сложение» с кодом $\begin{bmatrix} 0 & n \end{bmatrix}$ для «положительных» команд и «вычитание» того же кода — для «отрицательных».

Есть другой путь, позволяющий стандартизовать преобразование команд, сделать его независимым от «знака» команды. Этот путь — введение специальной операции «сложение адресов команд». В рассматриваемом примере эту операцию можно определить так: при «сложении адресов» преобразуемой команды и кода, используемого для преобразования, получаем команду, код операции которой всегда совпадает с кодом операции преобразуемой команды, а адрес равен сумме адресов обоих кодов. Но может встретиться случай, когда сумма адресов окажется больше $2^{12} - 1$ (111 111 111 111 в двоичной системе), т. е. будет записываться больше, чем 12 двоичными цифрами. Как поступить в таком случае? Естественнее всего дополнить определение операции «сложение адресов команд» условием, что в случае, когда сумма адресов оказывается больше или равной 2^{12} , от нее в качестве адреса команды — результата операции сохраняются лишь младшие 12 двоичных разрядов, а 13-й просто отбрасывается. Таким образом, расширенная операция «сложение адресов команд» оказывается пригодной не только для увеличения адресов команд, но и для их уменьшения.

Для уменьшения адреса команды на целое число n достаточно использовать для ее преобразования вспомогательный код, имеющий вид команды, адрес которой представляет собой поправку — n , записанную дополнительным кодом по модулю 2^{12} , т. е. равен $2^{12} - n$. Действительно, сумма адресов будет равна

$$A + (2^{12} - n) = 2^{12} + (A - n)$$

и после отбрасывания 1 в тринадцатом разряде (при $A \geq n$) получим адрес команды-результата $A - n$.

Перейдем к другому примеру. Рассмотрим трехадресную машину, ячейка которой состоит из 39 двоичных разрядов. Разряды (занумеруем их справа налево номерами от 1 до 39) используются для записи команд следующим образом: старшие 6 разрядов (с 34-го по 39-й) — для записи кода операции (6-разрядное целое двоичное число), разряды с 23-го по 33-й — для записи первого адреса, с 12-го по 22-й — для записи 2-го адреса и младшие 11 разрядов — для записи третьего адреса. Все адреса — 11-разрядные, следовательно, непосредственно адресуемая память может иметь до 2048_{10} ячеек. При записи чисел в разрядах с 34-го по 39-й помещается порядок числа в дополнительном (по модулю 2^6) коде, в 33-м разряде — код знака числа и в разрядах с 1-го по 32-й — мантисса числа. Правило кодирования знаков обычное («+» — 0; «-» — 1). При таком распределении разрядов код операции K соответствует порядку числа p с его знаком, причем соответствие описывается следующей формулой:

$$K = \begin{cases} p & \text{при } p \geq 0, \\ 2^6 + p & \text{при } p < 0. \end{cases}$$

Примеры:

$$\begin{aligned} K = 000\ 000_2 &= 0_{10}, & p &= 0_{10} \\ K = 000\ 001 &= 1, & p &= 1 \\ K = 011\ 111 &= 31, & p &= 31 \\ K = 100\ 000 &= 32, & p &= -32 \\ K = 100\ 001 &= 33, & p &= -31 \\ K = 111\ 111 &= 63, & p &= -1 \end{aligned}$$

Зависимость p от K можно выразить формулой

$$p = \begin{cases} K & \text{при } K < 32, \\ K - 2^6 & \text{при } K \geq 32 \end{cases}$$

или

$$p = 2^6 \left\{ \frac{K + 2^5}{2^6} \right\} - 2^5,$$

где $\{ \}$ обозначают дробную часть числа.

Адресная часть команды в рассматриваемом примере занимает те же разряды, что мантисса и знак числа. При этом разряды единиц первого A_1 , второго A_2 и третьего A_3 адреса соответствуют разрядам 2^{-10} , 2^{-21} и 2^{-32} мантиссы q числа x . Знак числа кодируется цифрой старшего разряда первого адреса. Отсюда следует, что

$$q = \begin{cases} A_1 \cdot 2^{-10} + A_2 \cdot 2^{-21} + A_3 \cdot 2^{-32} & \text{при } A_1 < 2^{10}, \\ -((A_1 - 2^{10}) \cdot 2^{-10} + A_2 \cdot 2^{-21} + A_3 \cdot 2^{-32}) & \text{при } A_1 \geq 2^{10}, \end{cases}$$

или

$$q = -\operatorname{sgn}(A_1 - 2^{10}) \cdot (A_1 \cdot 2^{-10} + A_2 \cdot 2^{-21} + A_3 \cdot 2^{-32}).$$

Окончательно, связь между числом и командой, выраженными одним и тем же кодом, дается формулой:

$$x = -2^5 \left\{ \frac{K + 2^5}{2^6} \right\} - 2^5 \operatorname{sgn}(A_1 - 2^{10}) \cdot (A_1 \cdot 2^{-10} + A_2 \cdot 2^{-21} + A_3 \cdot 2^{-32}).$$

Таким образом, и в этом примере команды имеют «знак», зависящий уже не от кода операции, а от величины первого адреса («+» при $A_1 < 2^{10}$, «-» — при $A_1 \geq 2^{10}$). Для преобразования команды $\boxed{K \mid A_1 \mid A_2 \mid A_3}$ в команду

$\boxed{K \mid A_1 + n_1 \mid A_2 + n_2 \mid A_3 + n_3}$ принципиально также можно воспользоваться операцией сложения чисел. Но вид «поправочной» команды, используемой для преобразования, в этом случае оказывается гораздо сложнее. Он зависит не только от «знаков» преобразуемой и преобразованной команд, но и от знака (без кавычек) поправки n_1 , а если $n_1 = 0$, то от знака n_2 , если же и $n_2 = 0$, то от знака n_3 .

Кроме того, здесь возникает другая трудность, которой не было вообще в первом примере. Дело в том, что при арифметических операциях над числами с плавающей запятой обычно предполагается, что числа, над которыми они производятся, являются нормализованными и результат их представляется также нормализованным числом. Числа же, соответствующие командам, будут, вообще говоря, не нормализованными. Точнее говоря, число, соответствующее команде

$\boxed{K \mid A_1 \mid A_2 \mid A_3}$, будет нормализованным, если $\{A_1 \cdot 2^{-10}\} \geq \frac{1}{2}$, и ненор-

мализованным, если $\{A_1 \cdot 2^{-10}\} < \frac{1}{2}$. Поэтому для преобразования команд нужны специальные модификации арифметических операций, например, операции, в конце выполнения которых не производится нормализация результата.

Для облегчения работы программиста и в этом примере целесообразно иметь в числе операций машины специальную операцию «сложение адресных частей команд». Оперантами этой операции являются код преобразуемой команды и вспомогательный код. В результате выполнения операции получаем команду, имеющую тот же код операции, что и преобразуемая команда, и адресную часть, равную или «сумме адресных частей» двух исходных кодов, если эта сумма меньше 2^{33} , или сумме адресных частей, уменьшенной на 2^{33} , если сама сумма больше или равна 2^{33} . Адресные части рассматриваются при этом как неотрицательные целые двоичные числа, состоящие из 33 разрядов:

$$A = A_1 \cdot 2^{22} + A_2 \cdot 2^{11} + A_3.$$

Точно так же, как в первом примере, операция «сложение адресных частей» может быть использована не только для увеличения, но и для уменьшения адресов команд. Пусть, например, команду $\boxed{K \mid A_1 \mid A_2 \mid A_3}$ нужно преобразовать в команду $\boxed{K \mid A_1 - 2 \mid A_2 + 3 \mid A_3 + 1}$. Для этого преобразования нужно использовать в операции «сложение адресных частей» код, имеющий вид команды с адресной частью:

$$A = 2^{33} - 2 \cdot 2^{22} + 3 \cdot 2^{11} + 1$$

и произвольным (обычно с нулевым) кодом операции. В двоичном представлении этот вспомогательный код имеет вид

000 000 11 111 111 110 00 000 000 011 00 000 000 001

Для увеличения первого адреса на две единицы и уменьшения третьего адреса на три единицы нужно использовать вспомогательный код с адресной частью

$$A = 2 \cdot 2^{22} - 3$$

или в двоичном представлении:

... 00 000 000 001 11 111 111 111 11 111 111 101

(старшие шесть разрядов произвольны). Вспомогательные коды такого типа называются «константами переадресации».

Кроме сложения адресных частей команд используются другие операции и приемы преобразования кодов команд, но о них мы будем говорить позднее.

Глава 4. АЛГОРИТМЫ

§ 1. Общее понятие об алгоритмах

Решение всякой задачи проходит через следующие стадии. Сама задача возникает, когда в реальном мире (то ли в материальном мире, то ли в мире идей) складывается новая ситуация и человек осознает это изменение. Возможно, при этом приходится выработать целую систему новых понятий, но гораздо чаще ситуация описывается в терминах уже сложившихся понятий. В результате этого процесса возникает постановка задачи. Если задача должна быть решена математически, а только такие задачи мы и рассматриваем, то постановка задачи должна быть формализована, т. е. выражена в терминах понятий, обладающих точно определенными свойствами и находящимися между собой в так же строго определенных отношениях. Не исключено, конечно, что задача с самого начала возникла как математическая задача, т. е. уже в формальной постановке.

Когда мы говорим, что задача поставлена, то предполагается, что известны не только характеристики существующей ситуации, но и требования к решению задачи, т. е. к изменению этой ситуации в желательном для нас направлении. Формализация постановки задачи позволяет выводить точные логические следствия из известных нам данных как об исходном, так и о конечном (желаемом) состоянии. Сравнительно редко путь, который ведет от исходного состояния к конечному, от постановки задачи к ее решению, бывает сразу ясен. Как правило, этот путь намечается именно в результате логического анализа формальной постановки задачи, в результате выявления новых свойств и отношений между понятиями и объектами, о которых в ней идет речь.

Этот путь решения задачи, последовательность действий, которые надо осуществить, чтобы от исходной ситуации прийти к желаемому новому состоянию, называется *алгоритмом* решения задачи. Может показаться странным, что мы даем такое расплывчатое, скорее философское, чем математическое, определение алгоритма. На это есть причины, которые вкратце сводятся к тому, что в математике существует несколько определений понятия алгоритма или,

точнее, несколько определений различных понятий, которые, в конечном счете, все оказываются эквивалентными между собой. Тем не менее все они имеют право на жизнь, так как каждое определение освещает это понятие с какой-то своей существенной стороны. Эти определения можно рассматривать, как различные уточнения (формализации) того неформального понятия алгоритма, которое было только что дано.

Итак, все сказанное выше можно рассматривать как первый пример неформальной постановки задачи, формализацию и решение которой мы пока давать не будем. Приведем другой, значительно более конкретный и ясный пример, известный под названием алгоритма Евклида.

Начнем с того, что сформулируем (а вернее, напомним) одно определение. Будем рассматривать целые неотрицательные числа. Скажем, что число d является *делителем* числа m , если существует такое число k (также целое и неотрицательное), что

$$m = kd.$$

Из этого определения и из тождества

$$m = m \cdot 1$$

следует, что число 1 является делителем любого числа m . Таким образом, у двух чисел m и n всегда есть по крайней мере один общий делитель — число 1. Но могут существовать и другие общие делители. Спрашивается, среди общих делителей двух данных чисел m и n существует ли наибольший, и если да, то как его найти?

Так в результате данного определения и анализа (весьма элементарного) его последствий возникла постановка задачи, причем уже формальная. Так как решение задачи не очевидно (предположим, что это так), подвергнем постановку задачи дополнительному анализу.

Из тождества

$$0 = 0 \cdot d$$

следует, что любое число d является делителем числа 0. Таким образом, если оба данных числа m и n равны нулю, то любое число является их общим делителем и наибольшего общего делителя не существует. Чтобы исключить этот случай, будем рассматривать лишь неравные исходные числа. Здесь мы допустили логический дефект — достаточно было бы потребовать, чтобы хотя бы одно из чисел m и n было отлично от нуля, и допустить их равенство, если оба они положительны. Но при решении более сложных задач сплошь и рядом случается так, что алгоритм строится не для самого общего случая, для которого поставлена задача. Иногда это делается потому, что для общего случая алгоритм найти не удастся, то ли из-за наших ограниченных способностей, то ли из-за того, что

алгоритм просто не существует. Так, например, нельзя разделить произвольный угол на три равные части, пользуясь только циркулем и линейкой — алгоритма такого построения не существует. Но может быть и так, что общий алгоритм очень сложен, а для частного случая найден простой алгоритм и многие практически интересные варианты задачи относятся к этому случаю. Возможно, что мы нашли алгоритм, который, может быть, и решает задачу в общем случае, но доказать, что он приводит к решению, нам удалось лишь при дополнительных ограничениях на исходные данные.

В нашей задаче исходные числа равноправны, поэтому обозначим через m большее из них, так что при сделанных предположениях

$$m > n \geq 0. \quad (1)$$

При $m > 0$ любой делитель числа m , в частности, наибольший общий делитель чисел m и n , если он существует, не превосходит m . Но отсюда вытекает, что он действительно существует, так как его следует искать среди конечной совокупности целых чисел, заключенных между 1 и m . В этой совокупности общих делителей m и n тоже конечное число, а из конечного набора чисел всегда можно выбрать наибольшее. (Кстати, мы пришли и к одному из возможных методов решения задачи: перебрать все целые числа от 1 до m , отбросить те из них, которые не являются общими делителями m и n , а среди оставшихся взять наибольшее. Но мы отвергнем этот метод из-за его громоздкости при практическом применении, хотя он и привел нас к важному теоретическому результату — доказательству существования решения задачи при сделанных предположениях.)

Обозначим наибольший общий делитель чисел m и n через $\text{НОД}(m, n)$ и исследуем его свойства. Заметим, что из тождества

$$m = 1 \cdot m$$

следует, что любое число m является своим делителем. Рассмотрим случай, когда $n = 0$. Любой делитель числа m является при этом делителем n , иначе говоря, общие делители m и n совпадают с делителями m , наибольший из которых равен m , так что

$$\text{НОД}(m, 0) = m \quad \text{при} \quad m > 0.$$

Пусть теперь $n > 0$. При этом возможно разделить m на n с остатком, т. е. представить число m в виде

$$m = qn + r,$$

где q и r — целые числа и

$$0 \leq r < n.$$

Пусть d — общий делитель m и n , т. е. существуют такие целые a и b , что

$$m = ad, \quad n = bd.$$

Тогда

$$r = m - qn = ad - qbd = cd,$$

где $c = a - qb$ — целое число. Это означает, что d — делитель числа r и, следовательно, общий делитель чисел n и r .

Пусть d_1 — общий делитель чисел n и r . Совершенно аналогично предыдущему доказывается, что d_1 является общим делителем чисел m и n . Итак, совокупность общих делителей пары чисел (m, n) совпадает с совокупностью общих делителей пары (n, r) . Следовательно, и наибольшие элементы этих совокупностей совпадают:

$$\text{НОД}(m, n) = \text{НОД}(n, r).$$

Итак, если $n > 0$, то задачу отыскания $\text{НОД}(m, n)$ можно свести к задаче отыскания $\text{НОД}(n, r)$, причем $n > r \geq 0$. Из сопоставления этого неравенства с неравенством (1) видно, что вторая задача относится к тому же классу задач, что и первая, и отличается от нее обозначениями исходных данных (n вместо m и r вместо n) и их значениями. Обозначения существенной роли, очевидно, не играют. Заменяем прежнее значение, обозначенное буквой m , значением n , а значение n — значением r . После этого мы вернемся к исходной ситуации, включая сюда и обозначения рассматриваемых в задаче величин, но с другими, меньшими, числовыми значениями этих величин. Продолжая действовать таким же образом, мы в конце концов придем к ситуации, когда значение, обозначенное через n , окажется равным нулю, а мы уже знаем, что при этом наибольший общий делитель пары чисел (m, n) равен m , точнее — тому числу, которое к этому моменту обозначено буквой m . Но, по доказанному, этому же числу равен наибольший делитель пары чисел, обозначенных буквами m и n в самом начале.

Отделяя во всем сказанном выше описание действий, приводящих к отысканию наибольшего общего делителя двух данных чисел, от обоснования этих действий, придем к следующей записи.

Даны два целых числа m и n , таких, что $m > n \geq 0$. Требуется найти их наибольший общий делитель d .

Шаг 1. Если $n = 0$, принять $d = m$ и закончить на этом процесс, в противном случае перейти на шаг 2.

Шаг 2. Представить число m в виде $m = qn + r$, где q, r — целые, $0 \leq r < n$. Заменить значение m значением n , заменить значение n значением r . Перейти на шаг 1.

Запись такого рода и называется алгоритмом. Это определение также не вполне четкое, к тому же оно противоречит предыдущему,

в котором под алгоритмом понимался сам процесс, приводящий к постановленной цели, а не его описание. Но, увы, среди математиков нет единого мнения по поводу того, что называть алгоритмом — процесс или его описание. Более логично сохранить термин алгоритм за описанием процесса. Дело в том, что фактически единого процесса, соответствующего данному описанию, не существует. Каждой совокупности исходных данных — значений m и n — соответствует свое протекание процесса, которое можно назвать *реализацией* алгоритма при этих исходных данных. Если же говорить о процессе вообще, отвлекаясь от значений исходных данных, то это понятие становится несколько неопределенным. Так, например, неизвестно, сколько раз в этом процессе выполнится каждый из шагов и, в частности, выполнится ли хотя бы раз шаг 2.

Тем не менее, иногда мы будем называть алгоритмом не только запись, определяющую ход некоторого процесса, но и сам процесс, рассматриваемый в его общем виде, т. е. отвлеченно от конкретных значений исходных данных.

§ 2. Еще один пример алгоритма

Алгоритмы могут быть использованы и фактически используются для определения целого ряда математических понятий, главным образом для того, чтобы определить новую операцию через старые, ранее известные операции. В качестве примера рассмотрим определение операции, обозначаемой знаком Σ . Для определенности рассмотрим смысл этого знака в следующем контексте:

$$\sum_{i=1}^n f(i), \quad (1)$$

где n — некоторое целое число или обозначение целого числа, $f(i)$ — выражение, зависящее от i . Часто смысл выражения (1) объясняют, говоря, что это сокращенное обозначение для

$$f(1) + f(2) + \dots + f(n). \quad (2)$$

Но смысл выражения (2) тоже нужно объяснить. Во-первых, кроме знака $+$, смысл которого будем считать понятным (т. е. будем считать, что операция сложения двух чисел была ранее определена), это выражение содержит многоточие, не обозначающее никакой известную нам операцию. Во-вторых, возможны случаи $n = 1$, $n = 0$ и даже $n < 0$, когда смысл записи (2) становится еще менее ясным. Поэтому определим смысл обозначения (1) при помощи следующего алгоритма.

Дано выражение $f(i)$, содержащее переменную (обозначение) i и целое число n . Требуется определить значение выражения (1).

Шаг 1. Принять $i = 1$ и $s = 0$, перейти на шаг 2.

Шаг 2. Если $i > n$, то взять s в качестве требуемого значения, и закончить процесс, в противном случае перейти на шаг 3.

Шаг 3. Заменить значение s значением выражения $s + f(i)$, заменить значение i значением $i + 1$. Перейти на шаг 2.

Этот алгоритм определяет значение выражения (1) при любом целом n . При этом предполагается, что в качестве $f(i)$ используется выражение, которое мы умеем вычислять, что операция сложения, встречающаяся при вычислении $s + f(i)$, уже определена, а также, разумеется, что нам понятен смысл всех других действий, используемых при описании алгоритма. Этих действий немного.

§ 3. Средства, используемые при записи алгоритмов

Перейти на шаг с номером k . Предполагается, что все шаги алгоритма пронумерованы различными, явно выписанными целыми числами, так что шаг с указанным номером определен однозначно. Для сокращения записи условимся считать, что если при выполнении действий, указанных в шаге с номером i , не определился номер следующего выполняемого шага, то подразумевается переход на шаг с номером $i + 1$.

Выбрать действие по условию. Этот выбор (*разветвление*) определяется записью, имеющей вид:

Если ---_1 , то ---_2 , в противном случае ---_3

где вместо ---_1 стоит описание условия, вместо ---_2 и ---_3 — описания каких-то действий. Когда условие удовлетворено, то выбирается и затем выполняется действие ---_2 , если же оно не удовлетворено, то выбирается и выполняется действие ---_3 . Может случиться, что когда условие не удовлетворяется, не нужно производить никаких действий (на данном шаге). Тогда часть записи *в противном случае* ---_3 будем опускать.

Закончить процесс. После этого действия никакие другие действия данного алгоритма уже не выполняются, даже если это действие встретилось в середине какого-либо шага, и независимо от места этого шага в записи всего алгоритма. Это действие будем описывать одним словом *выход*.

Присвоить значение переменной. До сих пор мы записывали это действие в виде

$$\text{Принять } \text{---}_1 = \text{---}_2. \quad (1)$$

или

$$\text{Заменить значение } \text{---}_1 \text{ значением } \text{---}_2, \quad (2)$$

где вместо ---_1 стояло какое-то обозначение, а вместо ---_2 — некоторое выражение. Это действие сводится к тому, что вычис-

ляется значение выражения ---_2 и обозначение ---_1 впредь начинает обозначать это значение. Если раньше оно обозначало какое-то другое значение (в таком случае мы использовали форму (2)), то, начиная с текущего момента, это старое значение забывается или теряется. Но старого значения у обозначения ---_1 могло и не быть (при этом использовалась форма (1)). Существовало это значение или нет, содержание действия фактически оказывается одним и тем же. Так как подобные действия очень часто встречаются при записи алгоритмов, введем для них краткую форму записи:

$$\text{---}_1 := \text{---}_2.$$

Обозначение ---_1 , которое имеет смысл применять в такой записи, назовем *переменной*, а действие, представленное этой записью, — *присваиванием* значения переменной.

Условимся также в начале алгоритма перечислять исходные данные, к которым применяется алгоритм, и результаты его работы (значения, которые он вырабатывает), а также неформально описывать назначение алгоритма. Условимся, наконец, весь алгоритм обозначать какой-либо буквой. Будем ставить эту букву в начале записи алгоритма и писать ее вместо слова «шаг» рядом с номером каждого шага.

При этих соглашениях два рассмотренных ранее примера запишутся в виде:

Е. Даны два целых числа m и n , таких, что $m > n \geq 0$. Требуется найти их наибольший общий делитель d .

Е1. Если $n = 0$, то $d := m$, выход.

Е2. Вычислить остаток r от деления m на n , $m := n$, $n := r$, перейти на Е1.

С. Дано выражение $f(i)$, содержащее переменную i , и целое число n . Требуется определить значение s выражения

$$\sum_{i=1}^n f(i).$$

С1. $i := 1$, $s := 0$.

С2. Если $i > n$, то выход.

С3. $s := s + f(i)$, $i := i + 1$, перейти на С2.

Рассмотрим теперь задачу о решении уравнения

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

не выше второй степени. Мы, следовательно, допускаем, что любые из чисел a , b , c могут быть равны нулю. Теория этой задачи достаточно хорошо известна. Но прежде чем записывать алгоритм, введем понятие текстового значения и текстовой переменной. *Текстовым значением* (или *текстом*) будем называть произвольную

конечную последовательность знаков, заключенную в кавычки. Пример:

«0», «∞», «2в», «барабан».

Знаки текста могут выбираться из некоторого конечного набора, который мы определять не будем, но который, разумеется, не должен содержать знака кавычек. *Текстовая переменная* — это переменная, значением которой может быть любой текст.

Нам в этом алгоритме понадобится текстовая переменная s , которой при выполнении алгоритма будет присвоено одно из значений

«0», «1», «2в», «2к», «2р», «∞»,

означающих соответственно, что данное уравнение не имеет корней, имеет один корень, два различных вещественных корня, два комплексных корня, два равных корня, имеет бесконечно много корней (обращается в тождество). Переменные x_1 и x_2 (или одна из них) получат значения вещественных корней в случаях «1», «2в», «2р». В случае «2к» переменные r и i получат соответственно значения вещественной и мнимой части (знак i безразличен).

Q. Даны три числа a , b , c — коэффициенты алгебраического уравнения не выше второй степени. Требуется определить количество и тип корней уравнения и вычислить значения корней, используя переменные s , x_1 , x_2 , r , i .

Q1. Если $a = 0$, то перейти на Q5.

Q2. Если $b^2 - 4ac > 0$, то $s := \text{«2в»}$, $x_1 := (-b + \sqrt{b^2 - 4ac})/2a$, $x_2 := (-b - \sqrt{b^2 - 4ac})/2a$, выход.

Q3. Если $b^2 - 4ac < 0$, то $s := \text{«2к»}$, $r := -b/2a$, $i := \sqrt{4ac - b^2}/2a$, выход.

Q4. $s := \text{«2р»}$, $x_1 := -b/2a$, выход.

Q5. Если $b \neq 0$, то $s := \text{«1»}$, $x_1 := -c/b$, выход.

Q6. Если $c \neq 0$, то $s := \text{«0»}$, выход.

Q7. $s := \text{«∞»}$, выход.

Хорошо или плохо записан этот алгоритм? Если рассматривать его как сводку результатов, которые дает теория, то особых претензий к нему быть предъявлено не может. Но если подходить к нему, как к руководству по практическому вычислению корней, то этот алгоритм определенно плох. Так, например, при $a \neq 0$ выражение $b^2 - 4ac$ этот алгоритм предписывает вычислить три раза: первый раз при проверке условия $b^2 - 4ac > 0$, а второй и третий раз — либо при вычислении x_1 и x_2 , либо при проверке условия $b^2 - 4ac < 0$ и при вычислении i . Несколько раз вычисляется произведение $2a$.

К алгоритму Q можно предъявить и другую претензию — в нем используется слишком много переменных. Для того чтобы подчеркнуть, что значения r и i — это не корни данного уравнения, а компоненты комплексных корней в случае «2к», введение этих двух переменных может быть и оправдано. Но природа корней уравнения зафиксирована не в значениях этих переменных, а в значении текстовой переменной s . Если учесть это, то сохранить результаты вычислений на шаге Q3 можно, присвоив их тем же переменным x_1 и x_2 , которые используются в других случаях.

Наконец, сама запись алгоритма длинновата. Так, выражение $b^2 - 4ac$, которое, как мы заметили, вычисляется до трех раз, фигурирует в записи алгоритма целых пять раз. Два раза явно и еще два раза неявно выписано выражение $-b/2a$.

Если учесть эти замечания, то алгоритм можно переписать в следующем виде:

Q'. Даны коэффициенты a, b, c алгебраического уравнения степени не выше 2. Требуется определить тип, число и значения корней уравнения, используя переменные s, x_1, x_2 .

Q'1. Если $a = 0$, то перейти на Q'5, в противном случае $x_1 := -b/2a, d := x_1^2 - c/a$.

Q'2. Если $d > 0$, то $s := \text{«2в»}, d := \sqrt{d}, x_2 := x_1 - d, x_1 := x_1 + d, \text{выход.}$

Q'3. Если $d < 0$, то $s := \text{«2к»}, x_2 := \sqrt{-d}, \text{выход.}$

Q'4. $s := \text{«2р»}, \text{выход.}$

Q'5. Если $b \neq 0$, то $s := \text{«1»}, x_1 := -c/b, \text{выход.}$

Q'6. Если $c \neq 0$, то $s := \text{«0»}, \text{выход.}$

Q'7. $s := \text{«∞»}, \text{выход.}$

У п р а ж н е н и я. 1. Переписать алгоритм Q' в предположении, что значения коэффициентов после отыскания корней уже не нужны (используя для вычислений переменные a, b, c).

2. Описать алгоритм перевода чисел из «чужой» системы счисления с основанием q в «свою» систему счисления с основанием p . Исходная запись числа задана в виде значения текстовой переменной z . Результатом работы алгоритма должно быть значение числовой переменной x . В алгоритмах будем пользоваться следующими операциями над текстовыми значениями:

а) операцию выделения первого слева знака из текстового значения обозначим знаком $\lfloor$, так что

$$\lfloor \text{«3,14»} = \text{«3»};$$

б) операцию отбрасывания первого слева знака из текстового значения обозначим знаком $\lfloor$, так что

$$\lfloor \text{«3,14»} = \text{«,14»};$$

в) операцию соединения двух текстов в один обозначим знаком $\frown$, так что

$$\text{«вор»} \frown \text{«она»} = \text{«ворона»};$$

г) для перевода целых чисел из диапазона $0 \leq k < q$ в тексты, состоящие из одной цифры в системе с основанием q , и наоборот будем пользоваться введенными ранее операциями $d(k)$ и $n(\alpha)$, так что

$$d(1) = \langle 1 \rangle,$$

$$n(\langle 1 \rangle) = 1$$

(числовые значения изображаются их записью в своей системе счисления без кавычек).

3. Описать алгоритм перевода чисел из своей системы счисления в чужую. Исходным является значение числовой переменной x и число s знаков после запятой в результате. Результирующий текст присваивается текстовой переменной z .

При переходе от алгоритма Q к алгоритму Q' мы руководствовались соображениями экономии, которые неизменно будут методом нашего внимания. Мы стремились сэкономить:

- 1) время работы алгоритма, сокращая количество вычислений,
- 2) число используемых переменных,
- 3) длину алгоритма.

Нетрудно догадаться, что при переходе от алгоритмов (в том виде, как мы их записываем) к машинным программам два последних пункта означают экономию числа ячеек памяти, занятых промежуточными результатами вычислений и самой программой.

§ 4. Другие средства записи алгоритмов

Существует множество разных приемов записи алгоритмов. Способ, которым мы пользовались, это по существу обычный язык, но с тщательным отбором слов, так, чтобы лишних слов, синонимов и т. п. в записи алгоритма не было, дополненный обычными математическими обозначениями и некоторыми специальными соглашениями (о знаке присваивания, о нумерации шагов алгоритма). Среди других способов упомянем так называемые *блок-схемы*. Они строятся из *блоков* двух типов, соединенных стрелками. Стрелки изображают последовательность вычислений, внутри блоков записывается, в чем состоят сами вычисления.

Блоки первого типа называются *вычислительными* (или *арифметическими*) и изображаются прямоугольниками, от которых ведет лишь одна стрелка, показывающая, к какому блоку следует перейти после окончания вычислений, предписанных данным блоком.

Блоки второго типа называются *логическими*. Они изображаются овалами, внутри которых записано некоторое условие, которое может либо удовлетворяться, либо не удовлетворяться при текущих значениях переменных. От логического блока ведут две стрелки, помеченные словами «да» и «нет». Первая стрелка показывает переход в случае, когда условие удовлетворено, вторая — когда оно не удовлетворяется.

Примеры блоков приведены на рис. 4.4.1 и 4.4.2.

К каждому блоку должна вести по крайней мере одна стрелка, но их может быть и несколько. В блок-схеме должна быть ровно одна стрелка, не выходящая ни из какого блока. Она указывает на блок, с выполнения которого начинается работа алгоритма, поэтому

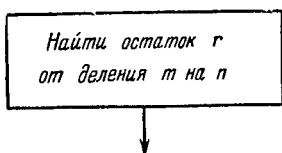


Рис. 4.4.1. Вычислительный блок.

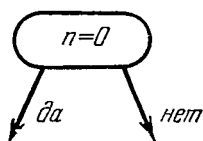


Рис. 4.4.2. Логический блок.

у ее начала может быть поставлено слово *Вход*. В схеме должна быть по крайней мере одна стрелка, не ведущая ни к какому блоку (или ведущая к слову *Выход*). Такие стрелки означают, что при необходимости перехода по ним работа алгоритма заканчивается.

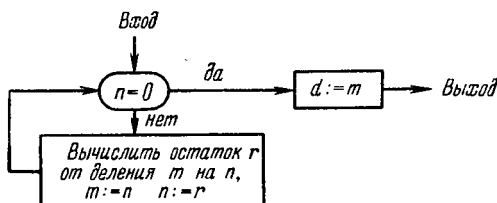


Рис. 4.4.3. Блок-схема алгоритма Евклида.

На рис. 4.4.3 и 4.4.4 представлены в качестве примеров блок-схемы алгоритма Евклида и алгоритма решения уравнения

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

Способ записи алгоритмов, изложенный в § 3, и метод представления алгоритмов блок-схемами страдают одним общим недостатком — они чересчур детальны. Этот недостаток почти не проявляется, пока алгоритмы столь просты, как это было в рассмотренных примерах. Однако если число шагов или блоков начинает насчитываться сотнями, то алгоритм становится почти необозримым, роль каждого отдельного шага (блока) — мало понятной, а схема их взаимодействия — невразумительной. Отчасти это вытекает из сложности алгоритма по существу, но если пользоваться более емкими средствами описания алгоритмов, то их обозримость улучшается.

Есть два основных приема, позволяющих подчеркнуть существо вычислительного процесса, определяемого алгоритмом. Один из

этих приемов заключается во включении в число средств записи алгоритмов способа описания циклических процессов без их расчленения на отдельные шаги. Например, к средствам, перечисленным в § 3, можно присоединить такой способ описания циклического повторения некоторого действия:

Пока —₁ *выполнять* —₂, где вместо —₁ стоит запись некоторого условия, а вместо —₂ описание действия. Выполнение

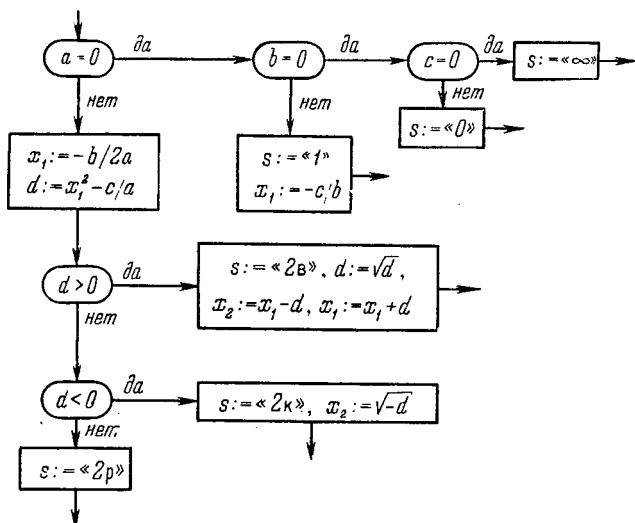


Рис. 4.4.4. Блок-схема решения уравнения $ax^2 + bx + c = 0$.

такого предписания можно изобразить следующей последовательностью шагов:

- A1. Если —₁, то на A2, в противном случае выход.
A2. —₂, перейти на A1.

Если воспользоваться данным приемом, то алгоритм Евклида можно переписать в виде одного шага:

- E1. Пока $n \neq 0$ выполнять $r := \text{остаток от деления } m \text{ на } n$
 $m := n, n := r$.

Результатом работы этого алгоритма будет последнее значение, присвоенное переменной m . В частности, если уже исходное значение переменной n равно нулю, то результатом будет исходное значение m , так как действие, стоящее после слова *выполнять*, не будет выполняться ни разу.

У п р а ж н е н и я. 1. Описать алгоритм вычисления остатка от деления неотрицательного числа m на положительное число n .

2. Описать с помощью понятия *Пока выполнять ...* алгоритм вычисления суммы $s = \sum_{i=1}^n f(i)$.

Второй прием, делающий описание алгоритма более наглядным и компактным, заключается в использовании рекурсивных функций.

Рассмотрим этот прием на примере вычисления суммы $\sum_{i=1}^n f(i)$.

Мы хотим определить функцию, обращение к которой записывалось бы в виде

$$\text{Сумма}(f, i, n), \quad (1)$$

где i — переменная (так называемая *переменная суммирования*), f — выражение, зависящее от переменной суммирования, n — верхняя граница значений, принимаемых переменной суммирования (нижняя граница считается равной 1), *Сумма* — наименование определяемой функции. Например, для вычисления суммы $\sum_{i=1}^{100} i^3$ следует обратиться к функции *Сумма* с помощью выражения

$$\text{Сумма}(i^2, i, 100).$$

Указание переменной суммирования в числе аргументов функции очень существенно. Так, например, выражение

$$\text{Сумма}(p^q, p, 5)$$

обозначает

$$1^q + 2^q + 3^q + 4^q + 5^q,$$

а выражение

$$\text{Сумма}(p^q, q, 5)$$

обозначает

$$p + p^2 + p^3 + p^4 + p^5.$$

Идея рекурсивного определения функции заключается в следующем. Из совокупности допустимых значений аргументов выделяются простые случаи, когда вычисление функции сводится к другим, ранее определенным или независимо определяемым понятиям. Например, если значение n неположительно, то следует считать, что значение выражения (1) равно нулю. В оставшихся случаях следует попытаться выразить значение функции через значение той же функции при других значениях аргументов, так, чтобы рано или поздно дело свелось к выделенным простым случаям. В рассматриваемом примере вычисление суммы n слагаемых можно, отщипнув последнее слагаемое, свести к вычислению суммы $n - 1$ слагаемого. Это приводит нас к следующему определению функции *Сумма*:

Сумма (f, i, n) = если $n \leq 0$, то 0,

в противном случае $f_{i=n} + \text{Сумма}(f, i, n - 1)$,

где $f|_{i=n}$ обозначает результат подстановки значения n переменной i в выражение f . Это определение дает способ вычисления суммы в любом конкретном случае. Например, для вычисления суммы

$$\text{Сумма } (i^2, i, 3)$$

оно предписывает воспользоваться следующей цепочкой равенств:

$$\begin{aligned} \text{Сумма } (i^2, i, 3) &= i^2|_{i=3} + \text{Сумма } (i^2, i, 3-1) = \\ &= 3^2 + \text{Сумма } (i^2, i, 2), \\ \text{Сумма } (i^2, i, 2) &= 2^2 + \text{Сумма } (i^2, i, 1), \\ \text{Сумма } (i^2, i, 1) &= 1^2 + \text{Сумма } (i^2, i, 0), \\ \text{Сумма } (i^2, i, 0) &= 0. \end{aligned}$$

У п р а ж н е н и е 3. Описать алгоритм нахождения наибольшего общего делителя целых чисел m и n , удовлетворяющих условию $m > n \geq 0$, с помощью двух рекурсивных функций: *НОД*, вычисляющей сам наибольший общий делитель, и *ОСТ*, вычисляющей остаток от деления $m \geq 0$ на $n > 0$.

Включение понятий цикла и рекурсивной функции в арсенал средств, используемых при записи алгоритмов, делает структуру алгоритма более сложной, но и более гибкой, более обозримой, чем простая последовательность пронумерованных шагов. Подобного рода понятия широко применяются в так называемых *алгоритмических языках*. В качестве примера алгоритмического языка в гл. 8 будет описан широко распространенный язык АЛГОЛ 60. А пока ограничимся общей характеристикой всех алгоритмических языков.

Чтобы определить алгоритмический язык, надо задать:

— множество значений, над которыми могут выполняться операции в этом языке и которые могут быть результатами этих операций;

— способ изображения этих первичных значений;

— набор первичных операций, которые могут выполняться над значениями;

— способ записи первичных операций;

— правила объединения записей отдельных операций в последовательности с указанием порядка выполнения этих операций.

В частности, алгоритмическим языком является язык любой вычислительной машины. Первичные значения в таком языке — это машинные слова и адреса ячеек памяти, изображаемые последовательностями нулей и единиц. Операции — машинные операции, свои для каждого типа машин, записываемые в виде машинных команд. Команды объединяются в программы выписыванием их подряд с указанием ячеек памяти, в которых они должны размещаться. Последовательность выполнения команд во времени, как правило, совпадает с последовательностью, в которой они написаны. Однако некоторые операции (операции перехода) определяют адрес следующей выполняемой команды особым образом.

Правила записи алгоритмов, сформулированные в § 3, еще не определяют язык, поскольку в них ничего не сказано о том, какие операции являются первичными и над какими значениями они могут выполняться. Эта неопределенность языка может оказаться полезной, как это будет показано в следующем параграфе. Алгоритмические языки, применяемые на практике, занимают промежуточное положение между машинными языками и аппаратом записи алгоритмов, введенным в предыдущем параграфе. Во многих из них также допускается известная свобода определения новых типов значений и операций над этими значениями. Правда, вновь вводимые значения не определяются независимо от первичных значений, а являются результатом их объединения в некоторые структуры. Операции над такими структурными значениями также определяются в конечном счете через первичные операции. Возможность вводить сложные значения и операции также является одним из средств сделать алгоритмы более компактными и обозримыми.

§ 5. Детализация алгоритмов

В § 3, озаглавленном «Средства, используемые при записи алгоритмов», были перечислены явно не все эти средства. Фактически речь шла только о средствах описания последовательности действий и введения или изменения обозначений.

Что касается самих действий, из выполнения которых складывается выполнение алгоритма, то обычно предполагается, что мы понимаем их смысл, умеем их выполнять и описывать. Таким образом, алгоритм — это предписание, сводящее выполнение некоторого действия к выполнению других действий, которые считаются более простыми. Так, например, при описании алгоритма Евклида такими действиями были проверка условия $n = 0$ и вычисление остатка от деления одного натурального числа на другое. Эти действия по отношению к данному алгоритму можно назвать *элементарными*.

Из требований, предъявляемых к элементарным действиям — понимание их смысла, умение их выполнять и записывать — особенно важно последнее. В самом деле, вычислительная машина выполняет арифметические действия гораздо быстрее и надежнее, чем человек, хотя она совершенно лишена способности понимать. Да и человека можно научить выполнять те или иные действия механически. Умение выполнять элементарные действия кажется более существенным. Но если вспомнить, что под алгоритмом мы условились понимать лишь запись, а не сам процесс, определяемый этой записью, то станет ясно, что написать алгоритм можно, умея лишь записывать элементарные действия.

Это не просто формальное рассуждение. Математика дает множество примеров, в которых такой подход оказывается оправданным

не только теоретически, но и практически. Вот один из них. Теория логарифмов позволяет свести выполнение операции умножения к операциям взятия логарифма числа (обозначим ее символом $\log$), отыскания числа по логарифму (обозначим ее символом $\exp$) и сложению.

Соответствующий алгоритм имеет вид:

- М. Даны числа a и b . Требуется найти их произведение c , используя таблицы логарифмов.
- М1. Если $a=0$ или $b=0$, то $c:=0$, выход.
- М2. Если $a>0$ и $b>0$ или $a<0$ и $b<0$, то $s:=1$, в противном случае $s:=-1$.
- М3. $p:=\log|a|$, $q:=\log|b|$, $r:=p+q$, $c:=\exp r$.
- М4. Если $s<0$, то $c:=-c$.
- М5. Выход.

В дальнейшем, если последний шаг алгоритма, подобно шагу М5 в этом примере, состоит из одного действия *выход*, такой шаг будем опускать.

Этот алгоритм имеет объективную ценность независимо от того, есть ли у нас под рукой таблицы логарифмов, обеспечивающие нужную точность вычислений. Практическая же ценность его в том, что, глядя на него (или держа его в уме), мы можем решить, воспользуемся ли мы им (и станем ли для этого разыскивать соответствующие таблицы), чтобы выполнить нужные нам вычисления, или будем вычислять произведения каким-либо иным способом.

Записать этот алгоритм мы смогли благодаря тому, что ввели специальные обозначения для входящих в него элементарных действий. В случае же алгоритма Евклида мы воспользовались для записи одного из элементарных действий обычным русским языком.

Действие, рассматриваемое в некотором алгоритме как элементарное, может быть само описано в виде алгоритма. Так, действие «вычислить остаток r от деления m на n », считавшееся элементарным при описании алгоритма Евклида, само может быть описано в виде, например, следующего алгоритма:

- Р. Найти остаток r от деления целого числа $m \geq 0$, на целое $n > 0$.
- Р1. Если $m < n$, то $r = m$, выход.
- Р2. $m := m - n$, перейти на Р1.

У п р а ж н е н и я. 1. Описать процесс вычисления остатка в виде алгоритма, более близкого к тому, который используется на практике.

2. Переделать алгоритм Е, считая элементарными те же действия, что в алгоритме Р.

После того как некоторые элементарные действия алгоритма сами описаны в виде алгоритмов, запись исходного алгоритма может

быть преобразована так, чтобы эти элементарные действия были из нее исключены. Этот прием, называемый *детализацией* алгоритма, очень полезен и даже необходим при построении мало-мальски сложных алгоритмов. Покажем еще на одном примере, как это делается.

§ 6. Алгоритм приведения подобных членов в многочлене

Этот пример выбран потому, что алгоритм, о котором пойдет речь, всем хорошо известен — настолько хорошо, что мало кто отдает себе отчет в алгоритмической стороне дела. Поэтому очень полезно в явной форме сформулировать алгоритм приведения подобных членов в многочлене.

Прежде всего, что такое многочлен? Многие, по-видимому, скажут, что многочлен — это сумма (или алгебраическая сумма) одночленов. В таком ответе есть доля истины, потому что он отражает наше желание рассматривать многочлен как результат сложения одночленов. Но, с другой стороны, сравним два задания: «Найти сумму чисел 2 и 3» и «Найти сумму одночленов x и y ». На первое задание мы отвечаем «5», на второе — « $x + y$ ». Фактически, второй ответ — это признание нашего бессилия выполнить задание. Этот ответ ничем не отличается от самого задания, кроме способа записи — ведь и задания можно было бы написать в виде: « $2 + 3 = ?$ » и « $x + y = ?$ », и тогда невозможность фактически выполнить второе задание становится особенно ясной.

С аналогичными ситуациями мы сталкиваемся так часто, что они являются скорее правилом, чем исключением:

- 23 — сумма 2 десятков и 3 единиц,
- $2/3$ — частное от деления 2 на 3,
- 3 — разность чисел 0 и 3,
- $\sqrt{2}$ — корень квадратный из 2,
- $2 + 3i$ — сумма числа 2 с произведением числа 3 на корень квадратный из —1

— все это примеры случаев, когда общепринятая символика отражает (а когда мы перестаем замечать ее условность — маскирует) невозможность выполнения тех или иных действий в рамках старых понятий. Разумеется, такая символика полезна не всегда. К ней прибегают лишь тогда, когда сами действия отражают какие-то реальные явления окружающего нас мира, когда для описания этих явлений нужны новые понятия (многозначные числа, дроби и т. д.), а для обозначения новых понятий нужна новая символика.

Итак, определим *многочлен* как последовательность выписанных подряд одночленов, перед каждым из которых стоит знак $+$ или $-$. Будем допускать возможность, когда в этой последовательности нет ни одного одночлена. Такую последовательность назовем *пустым* многочленом. Теперь надо определить, что такое одночлен.

В школьной алгебре отсутствует специальный термин для одночлена, не содержащего коэффициента, хотя без этого понятия невозможно определить, что называется приведением подобных членов. Исправим эту оплошность и введем следующее определение, такое же формальное, как данное выше определение многочлена.

Одночлен — это выписанные подряд коэффициент и терм. Одночлены подобны, если их термы тождественны. Что такое коэффициент и терм, пока определять не будем, так как мы можем описать алгоритм приведения подобных членов, не прибегая к этим определениям. Но мы должны предположить, что мы умеем выполнять некоторые операции над термами и коэффициентами, в частности, сравнивать термы на тождество, складывать и вычитать коэффициенты, определять, не равен ли коэффициент нулю. Впрочем, для облегчения нашей задачи предположим, что мы можем менять знак коэффициентов, и тогда операция вычитания коэффициентов нам не потребуется (а вернее, необходимость ее выполнения будет замаскирована новой символикой — отрицательными коэффициентами, без которых раньше мы могли бы обойтись).

После этих замечаний можно написать следующий алгоритм:

- S. Дан многочлен M . Построить многочлен M_1 , тождественный многочлену M , но не содержащий подобных членов.
- S1. Просмотреть знаки, стоящие перед одночленами многочлена M . Каждый знак — заменить на $+$, изменив при этом знак коэффициента следующего за ним одночлена.
- S2. Присвоить M_1 в качестве значения пустой многочлен.
- S3. Выделить из многочлена M первый одночлен O , оставшуюся часть снова обозначить через M .
- S4. Если среди одночленов M есть одночлен O_1 , подобный O , то исключить его вместе с предшествующим ему знаком $+$ из M , коэффициент O заменить суммой коэффициентов O и O_1 , снова перейти на S4.
- S5. Если коэффициент O отличен от нуля, то приписать к M_1 знак $+$, а за ним одночлен O .
- S6. Если многочлен M не пуст, то перейти на S3, в противном случае — выход.

Можно ли считать действия, в терминах которых описан этот алгоритм, элементарными? В некоторых случаях, по-видимому, да. Так, например, шаг S2, который можно переписать в виде:

S2: $M_1 := \text{пустой многочлен},$

вероятно, не нуждается в дальнейшей детализации, так как понятие *пустой многочлен* трудно свести к чему-либо более простому. Шаг S3 при принятом определении многочлена также едва ли нужно детализировать (в самом деле, существуют ли какие-либо более простые операции над конечными последовательностями элементов,

чем выделение или вычеркивание первого элемента такой последовательности?). Условие, проверяемое на шаге S6, также достаточно элементарно. Шаг S5 уже может вызвать сомнения. При желании его можно детализировать следующим образом:

S5.1. $K := \text{коэффициент } O$.

S5.2. Если $K = 0$, то выход.

S5.3. $M_1 := M_1 \hat{ } \langle + \rangle \hat{ } O$.

Здесь знаком $\hat{ }$ обозначена операция выписывания текстов подряд. Выход означает выход из шага S5, а не из алгоритма S.

Обратимся к шагу S1. Здесь уже, бесспорно, необходима детализация, которая впрочем выполняется без труда.

S1.1. $M_1 := \text{пустой многочлен}$.

S1.2. $Z := \text{начальный знак } M$, $O := \text{первый одночлен } M$,
 $M := \text{остаток } M$.

S1.3. Если $Z = \langle + \rangle$, то перейти на S1.5.

S1.4. $K := \text{коэффициент } O$, $T := \text{терм } O$, $K := -K$,
 $O := K \hat{ } T$.

S1.5. $M_1 := M_1 \hat{ } \langle + \rangle \hat{ } O$.

S1.6. Если многочлен M не пуст, то перейти на S1.2, в противном случае $M := M_1$.

Здесь по аналогии с тем, что говорилось выше, нужды в дальнейшей детализации уже нет. Рассмотрим наконец, шаг S4. Если при его детализации следовать буквально тому, что написано в этом шаге, то получим примерно следующий результат:

S4.1. $M_2 := \text{пустой одночлен}$, $K := \text{коэффициент } O$,
 $T := \text{терм } O$.

S4.2. Если многочлен M пуст, то $M := M_2$, выход.

S4.3. $O_1 := \text{первый одночлен } M$, $M := \text{остаток } M$,
 $K_1 := \text{коэффициент } O_1$, $T_1 := \text{терм } O_1$.

S4.4. Если $T_1 = T$, то $M := M_2 \hat{ } M$, $K := K + K_1$,
 $O := K \hat{ } T$, перейти на S4.1.

S4.5. $M_2 := M_2 \hat{ } \langle + \rangle \hat{ } O_1$, перейти на S4.2.

Ясно, однако, что так описанный шаг S4.4 заставит при выполнении алгоритма проделать много лишней работы. Ведь в M_2 собираются только те одночлены M , которые не подобны O , поэтому незачем присоединять их к остатку M и начинать всю работу заново. При детализации (в отличие от выполнения) алгоритма надо руководствоваться не буквой, а духом сказанного в тексте алгоритма. Гораздо разумнее записать шаг S4.4 следующим образом:

S4.4. Если $T_1 = T$, то $K := K + K_1$, перейти на S4.2
и включить операцию $O := K \hat{ } T$ в шаг S4.2 перед выходом.

У п р а ж н е н и е 1. Переписать алгоритм S, используя описанную выше детализацию шагов.

Можно ли на этом считать детализацию шага 4, а вместе с тем и всего алгоритма S законченной? В рамках введенных понятий и операций — да. Если же вспомнить, что понятия коэффициента и терма, а следовательно, и смысл операций сравнения термов, сложения коэффициентов, изменения знака коэффициента не были определены, то, конечно, дальнейшая детализация необходима. Она будет зависеть, однако, от того, как будут определены эти понятия.

Пусть, например, принято следующее определение. *Терм* — это последовательность (возможно, пустая) выписанных друг за другом множителей. Что такое множитель, пока не определяем. Будем считать, далее, два терма тождественными, если они образованы из одного и того же набора множителей, даже если эти множители выписаны в разном порядке. Это последнее определение явно не алгоритмично. Опишем в виде алгоритма то, что мы под этим понимаем.

- R. Даны два терма T и T_1 . Выработать ответ «да», если эти термы мы считаем тождественными, и «нет» — в противном случае.
- R1. Если терм T пуст, то перейти на R7.
- R2. Выделить первый множитель D терма T , обозначить остаток через T_2 : $T_2 :=$ пустой терм.
- R3. Если T_1 пуст, то ответ $:=$ «нет», выход.
- R4. Выделить первый множитель D_1 терма T_1 , обозначить остаток через T_1 .
- R5. Если $D = D_1$, то $T_1 := T_2 \hat{\ } T_1$, перейти на R1.
- R6. $T_2 := T_2 \hat{\ } D_1$, перейти на R3.
- R7. Если T_1 пуст, то ответ $:=$ «да», в противном случае ответ $:=$ «нет».

Для дальнейшей детализации надо уточнить понятие множителя, определить, в каком случае два множителя считаются тождественными, и т. д.

У п р а ж н е н и е 2. Будем понимать под множителем переменную, к которой справа сверху приписан показатель степени. Пусть переменная — это буква, показатель — число. Тогда алгоритм сравнения множителей D и D_1 крайне прост:

- Q1. Если переменная $D \neq$ переменная D_1 , то ответ $:=$ «нет», выход.
- Q2. Если показатель $D \neq$ показатель D_1 , то ответ $:=$ «нет», выход.
- Q3. Ответ $:=$ «да».

Однако в этом случае предыдущее определение тождества термов нельзя считать вполне удовлетворительным, так как термы $x^1y^2x^2y^{-1}$ и $y^1x^2x^1$ не тожде-

ственны в смысле этого определения, хотя в обычной трактовке их следует признать тождественным. Изменить алгоритм R так, чтобы он давал определение тождества термов, более близкое к обычному пониманию.

В дальнейшем нас будут интересовать алгоритмы, предназначенные для выполнения на электронных вычислительных машинах. Такие алгоритмы надо детализировать до тех пор, пока они не окажутся выраженными в терминах действий, доступных этим машинам. Итак, нам предстоит подробнее изучить возможности ЭВМ, ознакомиться с операциями, которые они умеют выполнять. Различные марки машин имеют много общего. Поэтому знакомиться с ЭВМ мы будем на примере сравнительно простой машины М-20. Более того, из операций, доступных этой машине, мы изучим не все, а только наиболее полезные и часто применяемые, и не сразу, а по мере необходимости.

Глава 5. СИСТЕМА КОМАНД

§ 1. Машины типа М-20

Совокупность всех операций, которые может выполнять электронная вычислительная машина, называется *системой команд* этой машины (*кодом* машины, ее *языком*).

Системы команд различных машин, даже при одном и том же количестве адресов в команде, обычно более или менее значительно отличаются одна от другой. Однако, овладев приемами программирования для одной машины, легко научиться составлять программы и для любой другой. Правила программирования остаются теми же, наиболее распространенные операции обычно применяются с несущественными отличиями на большинстве ЭВМ.

Эти операции можно разбить на несколько групп: арифметические операции (операции над числами), операции преобразования команд, операции над машинными словами, рассматриваемыми просто как последовательности двоичных цифр, операции управления и др. Ниже мы последовательно рассмотрим эти группы и опишем, в чем состоит каждая из операций на примере машин типа М-20. К этому типу относятся трехадресные электронные вычислительные машины М-20, БЭСМ-3М, БЭСМ-4, М-220, М-222, имеющие (если пренебречь мелкими различиями) одну и ту же систему команд. Они работают в двоичной системе счисления, каждое *машинное слово* (содержимое ячейки памяти) содержит 45 двоичных разрядов. Разряды будем нумеровать справа налево (от младших к старшему).

Коды чисел и команд занимают целое машинное слово. Для представления чисел разряды слова используются следующим образом. Младшие 36 разрядов — с 1-го по 36-й — служат для изображения абсолютной величины мантиссы числа (в прямом коде), которая, следовательно, может лежать в пределах от нуля до $1-2^{-36}$ (для нормализованных чисел — от $\frac{1}{2}$ до $1-2^{-36}$). В разрядах с 37-го по 43-й записывается *код порядка* числа. Код порядка числа — это двоичное число, равное порядку числа плюс 64_{10} (или $1000\ 000_2$). Так, порядок, равный 2, запишется в этих разрядах как 1 000 010, порядок, равный —1, будет выглядеть как 0 111 111.

Порядки чисел могут лежать в пределах от -64 (код порядка 0 000 000) до $+63$ (код порядка 1 111 111). Заметим, что порядку нуль соответствует код 1 000 000. Следовательно, все неотрицательные порядки начинаются с единицы в 43 разряде, отрицательные имеют в этом разряде 0. Другими словами, можно считать, что 43-й разряд изображает знак порядка, но в отличие от традиционного представления знак $+$ изображается цифрой 1, а знак $-$ изображается цифрой 0. Сам порядок изображается в дополнительном коде по модулю 2^8 . Такое представление порядка числа удобно тем, что при монотонном возрастании порядка в указанных пределах код порядка возрастает также монотонно и изменение порядка численно совпадает с изменением кода порядка.

Знак числа кодируется цифрой 44-го разряда, причем для знака $+$ принят код 0, для знака $-$ принят код 1.

В 45-м разряде может быть помещен дополнительный признак числа, который можно использовать, чтобы выделить некоторую группу чисел в памяти. Этот признак не имеет арифметического значения и не влияет на выполнение арифметических операций над числами.

Рассмотрим распределение разрядов слова при записи команд. Целые двоичные числа, записанные в разрядах $36 \div 25$, $24 \div 13$, $12 \div 1$, называются *адресами* (соответственно первым, вторым и третьим). В $37 \div 42$ разрядах помещается целое двоичное число, называемое *кодом операции* (КОП). В 45, 44 и 43 разрядах записываются *признаки модификации* адресов (обозначаются π_1, π_2, π_3). О значении этих признаков мы будем говорить позже, пока будем рассматривать команды с $\pi_i = 0$.

Для сокращения записи кодов чисел и команд будем пользоваться не двоичной, а восьмеричной системой счисления, объединяя по три двоичных цифры в одну восьмеричную цифру:

Двоичные числа (группы цифр)	000	001	010	011	100	101	110	111
Восьмеричные цифры	0	1	2	3	4	5	6	7

Рассмотрим примеры записи чисел. Мантисса числа, записанная восьмеричными цифрами $\alpha_1\alpha_2 \dots \alpha_{11}\alpha_{12}$, обозначает правильную восьмеричную дробь $0,\alpha_1\alpha_2 \dots \alpha_{12}$, т. е. число

$$\frac{\alpha_1}{8} + \frac{\alpha_2}{8^2} + \dots + \frac{\alpha_{12}}{8^{12}}.$$

Восьмеричная запись признака, знака и порядка числа расшифровывается несколько сложнее. Следующая таблица показывает,

как изменяется восьмеричный код в разрядах 45—37 при различных значениях признака числа, его знака и порядка;

	Знак	Порядок	
		$-64_{10} \div -1$	$0 \div 63_{10}$
0	+	$000_8 \div 077_8$	$100_8 \div 177_8$
0	—	$200_8 \div 277_8$	$300_8 \div 377_8$
1	+	$400_8 \div 477_8$	$500_8 \div 577_8$
1	—	$600_8 \div 677_8$	$700_8 \div 777_8$

Пр и м е р ы кодов чисел (всюду признак равен нулю).

1. Число 1. Двоичная нормализованная запись (ДНЗ): $2^1 \times 0,1$.
Двоичный код (ДК): 0 0 1 000001 100 ... 0.
Восьмеричный код (ВК): 101 4000 0000 0000.
2. Число -10 . ДНЗ: $-2^4 \times 0,101$;
ДК: 0 1 1 000100 10100 ... 0; ВК: 304 5000 0000 0000.
3. Число $11/16$. ДНЗ: $2^0 \times 0,1011$;
ДК: 0 0 1 000000 101100 ... 0; ВК: 100 5400 0000 0000.
4. Число $-1/3$. ДНЗ: $-2^{-1} \times 0,101010 \dots$;
ДК (с округлением): 0 1 0 111111 101010 ... 1011;
ВК (с округлением): 277 5252 5252 5253.
5. Число 2^{-65} . ДНЗ: $2^{-64} \times 0,1$;
ДК: 0 0 0 000000 100 ... 0; ВК: 000 4000 0000 0000.

Код с нулевой мантиссой изображает число нуль, независимо от порядка. Принято считать, что нормализованным представлением нуля является код числа с нулями во всех двоичных разрядах, кроме, может быть, разряда признака, иначе говоря, код с порядком -64_{10} и мантиссой, равной нулю, — так называемый *машинный нуль*. Наименьшим по абсолютной величине нормализованным числом, отличным от нуля, является число из примера 5.

- У п р а ж н е н и я. 1. Какое наименьшее положительное число (не обязательно нормализованное) может быть представлено словом машины М-20?
2. Какое наибольшее число может быть записано в ячейке машины М-20?
 3. Какое число представлено восьмеричным кодом 305 4200 0000 0000?
 4. Написать восьмеричный код числа $1/7$.

При записи команд восьмеричными цифрами совокупность трех признаков модификации изображается одной цифрой от 0 до 7, код операции — двузначным восьмеричным числом от 00 до 77_8 , каждый из трех адресов — четырехзначным восьмеричным числом от 0000 до $7777_8 = 4095_{10}$. Эти компоненты кода команды будем обозначать: π , КОП, A_1 , A_2 , A_3 . Иногда вместо π будем писать $\pi_1\pi_2\pi_3$, где π_i , $i = 1, 2, 3$, — двоичные цифры, представляющие признаки модификации отдельных адресов.

Пример. В команде 652 0001 0000 2205 компоненты имеют значения: $\pi = 6$ ($\pi_1 = \pi_2 = 1$, $\pi_3 = 0$), КОП = 52, $A_1 = 0001$, $A_2 = 0000$, $A_3 = 2205$.

Чаще всего адрес в команде является адресом, т. е. номером, ячейки памяти (хотя встречаются исключения из этого правила). Если это так, то почти всегда адреса A_1 и A_2 — это адреса, по которым происходит считывание из ячеек памяти (например, считываются операнды), а по адресу A_3 производится запись (результата). Такая система адресации позволяет непосредственно адресовать 4096₁₀ ячеек. Фактически оперативная память машины (ОЗУ) имеет 4095 ячеек — ячейки с адресом 0000 не существует. При попытке считать содержимое ячейки с этим адресом считывается код, все 45 разрядов которого равны нулю. Как мы знаем, такой код изображает число ноль.

Прежде чем перейти к описанию операций машины, отметим одно свойство машины М-20. При выполнении операций машина помимо основного результата вырабатывает управляющий сигнал, так называемый признак ω . Этот признак может быть равен нулю или единице. Он используется при работе команд передачи управления. При описании операций машины мы будем говорить, в каких случаях какое значение получит ω .

Перейдем к описанию операций машины. После словесного наименования операции будем в скобках указывать ее сокращенное символическое обозначение, используемое для записи кода в программе, и ее восьмеричный (цифровой) код. Можно было бы пользоваться только цифровым кодом операции, но это несколько затрудняет понимание программ.

§ 2. Арифметические операции

Сложение (+; 01). Операция состоит в том, что число, вызванное из ячейки ОЗУ, номер которой указан в первом адресе команды (коротко: число по первому адресу), складывается с числом по второму адресу и сумма засылается в ячейку ОЗУ, номер которой указан в третьем адресе команды (коротко: засылается по третьему адресу). Если результат больше нуля или равен нулю, вырабатывается сигнал $\omega = 0$, если меньше нуля, то $\omega = 1$.

Вычитание (—; 02). Из числа по первому адресу вычитается число по второму адресу, разность засылается по третьему адресу. Если результат неотрицателен, то $\omega := 0$, в противном случае $\omega := 1$.

Умножение ($\times$; 05). Число по первому адресу умножается на число по второму адресу, результат засылается по третьему адресу. Если порядок результата больше нуля (код порядка больше 100), то $\omega := 1$, в противном случае $\omega := 0$. Заметим, что для нормализованного результата условие выработки $\omega = 1$ равносильно

условию: абсолютная величина результата больше или равна единице.

Деление (/; 04). Число по первому адресу делится на число по второму адресу, частное засылается по третьему адресу. Условия выработки признака ω те же, что для операции умножения.

Вычитание модулей (|—|; 03). Из абсолютной величины числа по первому адресу вычитается абсолютная величина числа по второму адресу, разность засылается по третьему адресу. Если результат неотрицателен, то $\omega := 0$, в противном случае $\omega := 1$.

Извлечение квадратного корня ($\sqrt{}$; 44). Из числа по первому адресу извлекается квадратный корень и результат засылается по третьему адресу. Содержимое разрядов второго адреса не влияет на выполнение операции. Условия выработки признака ω те же, что для операции умножения.

Сложение порядка с адресом (+ АП; 06). При выполнении этой команды содержимое ее 25 ÷ 36 разрядов (т. е. первого адреса) является не адресом операнта, а непосредственно одним из операндов. Код k , записанный в 25 ÷ 31 разрядах (содержимое остальных разрядов первого адреса не влияет на выполнение операции), рассматривается как код порядка. Соответствующий порядок (т. е. $k - 100_8$) складывается с порядком числа по второму адресу. По третьему адресу засылается число, имеющее полученный порядок, его мантисса и знак берутся от числа по второму адресу. Условия выработки признака ω те же, что для операции умножения.

Вычитание адреса из порядка (—АП; 46). Выполняется аналогично предыдущей, только из порядка числа по второму адресу вычитается порядок, код которого записан в 25 ÷ 31 разрядах команды (условия выработки признака ω те же).

Для всех арифметических операций для точной записи мантиссы результата требуется, вообще говоря, больше двоичных разрядов, чем в мантиссах операндов. При делении и извлечении корня мантисса может оказаться даже бесконечнозначной. Но в ячейках памяти можно хранить числа с мантиссой определенной длины. Кроме того, при сложении и вычитании алгебраическая сумма мантисс операндов может получиться больше или равной 1. Поэтому результат арифметической операции нормализуется (представляется в виде нормализованного числа) и округляется, т. е. заменяется ближайшим двоичным числом с 36-разрядной мантиссой.

Как при выполнении самих арифметических операций, так и в процессе нормализации и округления результата могут встретиться особые случаи.

Порядок результата может оказаться больше $+63_{10}$, такой порядок не может быть изображен в отведенных для него разрядах. Если при этом мантисса результата отлична от 0, машина автоматически прекращает работу (происходит «Авост»). В частности,

«Авост» происходит, если при делении делитель оказывается равным нулю.

Если порядок результата окажется меньше -64_{10} или мантисса результата равна нулю, то в качестве результата выработается нормализованный нуль.

Все арифметические операции выполняются машиной в предположении, что операнды — нормализованные числа. Результат при этом также всегда оказывается нормализованным. Однако почти любую операцию можно выполнить и над ненормализованными операндами. Исключение составляет операция деления, выполняемая лишь в том случае, когда мантисса делимого меньше удвоенной мантиссы делителя (для нормализованных операндов это условие всегда выполняется).

Если хотя бы один операнд арифметической операции (единственный операнд операций $\sqrt{\quad}$, $+AP$ и $-AP$) был снабжен признаком (1 в 45 разряде), то и результат снабжается признаком; если ни один из операндов не имел признака, то и результат его не получает.

§ 3. Операции преобразования кодов

Сложение адресных частей команд ($+M$; 13). Эта операция, наиболее употребительная для преобразования кодов команд, уже упоминалась. В результате ее выполнения формируется и засылается по третьему адресу команда, имеющая тот же код операции и признаки модификации, что и команда, вызванная по первому адресу. Адресная часть команды-результата равна сумме адресных частей (рассматриваемых как целые 36-разрядные двоичные числа) команд по первому и второму адресам. Если эта сумма окажется больше или равной 2^{36} , то в ней сохраняются лишь 36 двоичных разрядов, а 1 в 37, разряде отбрасывается (в этом случае $\omega := 1$).

П р и м е р 1. При выполнении операции $+M$ из памяти были вызваны коды команд:

по 1-му адресу	3 01 2615 3730 1147
по 2-му адресу	4 34 7772 0004 7776
Результат операции:	3 01 2607 3735 1145 ($\omega = 1$).

Код операции команды, вызванной по второму адресу, не влияет на выполнение операции $+M$ и на ее результат.

Вычитание адресных частей команд ($-M$; 33). Выполняется аналогично операции $+M$, но из адресной части команды, вызванной по первому адресу, вычитается адресная часть команды, вызванной по второму адресу. Если адресная часть второй команды больше адресной части первой команды, вырабатывается признак $\omega = 1$.

Пример 2. При выполнении операции — M над теми же командами, что в предыдущем примере, получим результат:

3 01 2623 3723 1151 ($\omega = 1$).

Поскольку разряды адресной части команды совпадают с разрядами мантиссы числа (точнее, ее абсолютной величины), операции $+M$ и $-M$ называют иногда *сложением мантисс* и *вычитанием мантисс* (отсюда и их символические обозначения). Единственное достоинство этих названий — их краткость.

Пересылка ($ПЧ$; 00). При этой операции слово, вызванное по первому адресу, без всяких изменений засылается в ячейку по третьему адресу. Содержимое второго адреса команды $ПЧ$ не влияет на ее выполнение. Эта операция применяется, когда число или иное слово нужно поместить в определенную ячейку памяти вместо кода, находившегося там раньше. Операция пересылки не вырабатывает управляющий сигнал ω , а сохраняет его прежнее (т. е. выработанное при исполнении предшествующей команды) значение.

Сравнение ($\neq$; 15). Впоследствии с этой операцией мы познакомимся подробнее. Пока что будем иметь в виду лишь то, что при выполнении этой операции слово, прочитанное по первому адресу, сравнивается со словом по второму адресу. Если эти слова совпадают во всех 45 разрядах, то вырабатывается признак ω , равный 1, если они различаются хоть в одном разряде, то выработанное значение признака ω будет равно нулю. В ближайших главах эту операцию мы будем применять только с нулевым третьим адресом.

§ 4. Операции управления

Выше уже упоминался нормальный порядок выборки команд из памяти и их выполнения, при котором за командой, хранившейся в ячейке ОЗУ с адресом k , выполняется команда из ячейки с адресом $k + 1$.

В некоторых случаях этот порядок должен быть изменен. Это может потребоваться, например, когда после окончания вычислений по одной из частей программы нужно перейти к другой ее части, которая размещена в памяти машины не непосредственно вслед за первой, а где-либо в другом месте. Такого рода переходы называются *безусловными переходами*. Это название означает, что всегда, при всех условиях, по окончании работы данной части программы требуется перейти ко вполне определенному другому месту в программе. Но бывают и другие ситуации. В гл. 4 мы часто встречались с шагами вида:

Если «условие», то перейти на шаг «номер шага».

В такой записи необходимость перехода ставится в зависимость от

выполнения условия. Если указанное условие (например, некоторое соотношение между текущими значениями переменных) удовлетворяется, то требуется перейти к шагу с указанным номером. Если условие не удовлетворяется, то после шага, содержащего это условие, выполняется следующий по порядку шаг.

Такие переходы (связанные с проверкой выполнения условий) называются *условными переходами*. Перейдем к описанию команд, реализующих переходы, называемых также командами *передачи управления*. Это название связано с тем, что команды управляют последовательностью выполнения операций. *Передать управление* команде с адресом A означает: вслед за данной операцией перейти к выполнению операции, описываемой командой, хранящейся в ячейке памяти с адресом A .

Безусловная передача управления (PB ; 56). После выполнения этой команды очередная команда выбирается из ячейки ОЗУ, указанной во втором адресе команды PB . В этом заключается основной смысл операции PB . Но при этом остаются неиспользованными два адреса команды: первый и третий. Это нецелесообразно. Поэтому при выполнении команды PB одновременно с передачей управления по второму адресу код, вызванный по первому адресу, засылается в ячейку ОЗУ по третьему адресу.

Возможность пересылки кода из одной ячейки памяти в другую параллельно с передачей управления иногда оказывается полезной. Но можно и не использовать, для этого достаточно поместить нули во все разряды первого и третьего адресов команды PB .

Передача управления с возвратом (PB ; 16). После выполнения этой команды очередная команда выбирается также по второму адресу. В ячейку ОЗУ по третьему адресу засылается команда передачи управления с возвратом (PB) со вторым адресом, равным первому адресу выполняемой команды PB . Первый и третий адрес, а также признаки модификации адресов засылаемой команды равны нулю. Короче говоря, если выполняемая команда имеет вид

0 16 A_1 A_2 A_3

то в ячейку A_3 будет записана команда

0 16 0 A_1 0

и выполнение программы будет продолжаться с команды с адресом A_2 .

Таким образом, операция PB — это также операция безусловной передачи управления, но она позволяет одновременно с переходом на выполнение некоторой части программы (подпрограммы) заслать в конец этой части команду передачи управления на новое место программы. В частности, после выполнения подпрограммы можно вернуться к тому месту программы, откуда был осуществлен переход на подпрограмму. Для этого следует, если команда PB

хранится в ячейке памяти с адресом k , сделать ее первым адресом адрес $k + 1$. Операция *ПВ* очень удобна для организации обращений к *подпрограммам* — таким участкам программы, которые нужно выполнить несколько раз по ходу выполнения основной программы, в разных ее местах.

Для организации условных передач управления машина М-20 имеет две команды. Условие, которое они проверяют, состоит в сравнении текущего значения признака ω с нулем или с единицей. Однако, благодаря тому, что очень многие операции вырабатывают значение этого признака, в программах для машины М-20 легко организовать проверку самых разнообразных условий.

Условный переход по $\omega = 0$ (П0; 76). Если текущее значение ω равно нулю, то очередная команда выбирается по второму адресу, в противном случае сохраняется обычный порядок выполнения команд, т. е. следующей выполняется команда из ячейки с адресом, на единицу большим адреса выполняемой команды условного перехода. В обоих случаях слово, прочитанное по первому адресу, засылается по третьему адресу.

Условный переход по $\omega = 1$ (П1; 36). Эта команда выполняется так же, как предыдущая, но переход по второму адресу происходит при $\omega = 1$, а переход к следующей по порядку команде программы — при $\omega = 0$.

Сами операции перехода не меняют значение признака ω , оставшееся от предыдущей команды.

Останов (СТОП, 77). При выполнении этой команды выполнение программы прекращается.

§ 5. Запись программ

Запись программы должна быть удобной для работы с ней. В конечном счете всякая программа должна быть введена в машину, а для этого ее предварительно надо отперфорировать — перенести на перфокарты, с которых ее сможет прочесть читающее устройство. Для перфорации команды программы должны быть записаны в восьмеричном коде. Машина не прощает человеку ни малейшей его ошибки — она попросту не способна их обнаружить. Поэтому единственный, но зато надежный способ заставить машину правильно выполнить программу — это написать ее без ошибок. Для этого программист должен быть предельно внимательным и аккуратным и, разумеется, должен точно знать правила программирования, иными словами, знать реакцию машины на каждый значок, на любую мельчайшую деталь записи программы. Этому содействует четкая, точно определенная форма записи программ.

Обычно программа записывается на бланках определенной формы. Для машины М-20 форма бланка может быть примерно следующей:

Адрес команды	Признаки модификации и код операции	1-й адрес	2-й адрес	3-й адрес	Пояснение смысла команды (результат операции)
А	п КОП	А ₁	А ₂	А ₃	Примечание

В первой графе бланка записывается четырехзначный восьмеричный номер ячейки ОЗУ, где хранится данная команда. В следующей графе помещаются три восьмеричные цифры. Первая из них обозначает совокупность признаков модификации адресов, остальные две — код операции. В третьей, четвертой и пятой графах бланка записываются четырехзначные адреса команды.

Для чего нужна последняя, шестая графа? Очевидно, не для перфорации — содержимое этой графы (как, впрочем, и первой) не перфорируется. Но работа с программой не сводится лишь к ее перфорации. Программа должна быть сначала составлена, тщательно проверена и только после этого она может быть передана на перфорацию. К сожалению, проверка не гарантирует правильности программы. Поэтому после первого ввода программы в машину начинается период ее *отладки* — оценки программы по результатам ее работы. В ходе отладки программисту может потребоваться внести в программу ряд исправлений и изменений. При составлении, проверке, отладке и изменении программы программисту приходится проследивать ее работу. Он должен четко представлять себе, какие промежуточные результаты выработала программа и где они хранятся на любом этапе работы программы. Построчные примечания к программе очень способствуют такому пониманию. Особенно трудно бывает разобраться в чужой программе (например, студенту — в программе, составленной преподавателем, и наоборот). И здесь примечания очень полезны.

Из других средств, облегчающих составление и проверку правильности программ, пока отметим только необходимость предварительной записи алгоритма в другой, более наглядной форме, чем программа, даже снабженная примечаниями, — например, в виде блок-схемы или пошаговой записи. Малоопытные программисты часто пренебрегают этим этапом и, как правило, бывают наказаны. В лучшем случае отладка программы сильно затягивается, в худшем — ее вообще не удастся довести до конца. При наличии наглядно записанного алгоритма упрощается и составление примечаний — они могут ссылаться на те или иные действия алгоритма и использовать введенные там обозначения.

Глава 6. ПРОГРАММИРОВАНИЕ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ И ПРИСВАИВАНИЙ

Знакомство с составлением машинных программ начнем с программирования арифметических (со школьной точки зрения — алгебраических) выражений и формул — присваиваний значений таких выражений переменным. Действия, применяемые при вычислении арифметических выражений, — это арифметические операции.

Примеры арифметических присваиваний

$$\begin{aligned}x &:= a^2 + b^2 + c^2, \\y &:= ax^2 + bx + c, \\z &:= \frac{(a+x)^3 + (a-y)^3}{(a+x)^2 + (a-y)^2}, \\ \Delta &:= \frac{1}{2} \left(\frac{x}{y} - y \right), \quad y := y + \Delta.\end{aligned}$$

В последнем примере предполагается, что оба присваивания составляют одно действие алгоритма. Арифметические присваивания — это, как правило, части более крупных алгоритмов, и программы, которые мы будем писать, следует считать частями более крупных программ.

В арифметическом выражении обычно более или менее ясен бывает порядок действий — порядок выполнения арифметических операций. В то же время общепринятые правила, определяющие этот порядок, оставляют некоторую свободу. Этой свободой следует пользоваться, чтобы программа получилась более *экономичной* — чтобы она содержала по возможности меньше команд и занимала возможно меньшее количество *рабочих ячеек* — так называются ячейки, в которые помещаются промежуточные результаты вычислений. Как это делается, покажем на примерах.

Пример 1.

$$x := a^2 + b^2 + c^2. \quad (1)$$

Предположим, что перед началом вычислений числа a , b и c хранятся в ячейках памяти с адресами 0041, 0042 и 0043 соответственно и должны быть сохранены в этих ячейках для использования при

вычислениях по другим формулам. Пусть число x должно быть помещено в ячейку 0050. В качестве рабочих можно использовать ячейки, начиная с 0101. Программу следует разместить, начиная с ячейки 0001.

Человек, считающий вручную, вероятно, нашел бы сначала квадраты чисел a , b и c , а затем их сумму. Посмотрим, как выглядит программа при таком порядке действий:

Программа 1

А	π	КОП	A ₁	A ₂	A ₃	Примечание
0001	0	05	0041	0041	0101	a^2
0002	0	05	0042	0042	0102	b^2
0003	0	05	0043	0043	0103	c^2
0004	0	01	0101	0102	0101	$a^2 + b^2$
0005	0	01	0101	0103	0050	$x := a^2 + b^2 + c^2$

Количество операций в этой программе сокращено быть не может. Но можно сэкономить две рабочие ячейки. С точки зрения экономии места в памяти в программе допущены две ошибки. Результат первой же операции (a^2) можно поместить сразу в ту ячейку, где будет размещен результат вычислений по заданной формуле — в ячейку 0050 (это не общее правило, а лишь замечание к данной программе). Результат второй операции был использован не тотчас же, а лишь в четвертой операции, а результат третьей операции — в пятой. Это потребовало занять лишнюю рабочую ячейку. Целесообразнее было бы третью и четвертую операции переставить. С учетом сделанных замечаний программу следует привести к следующему виду.

Программа 2

А	π	КОП	A ₁	A ₂	A ₃	Примечание
0001	0	05	0041	0041	0050	a^2
0002	0	05	0042	0042	0101	b^2
0003	0	01	0050	0101	0050	$a^2 + b^2$
0004	0	05	0043	0043	0101	c^2
0005	0	01	0050	0101	0050	$x := a^2 + b^2 + c^2$

В задании на составление машинной программы кроме алгоритма, по которому ведутся вычисления, обычно ничего не задается. Адреса ячеек для размещения исходных величин и результатов вычислений, адреса рабочих ячеек и команд самой программы выбирает сам программист. Однако если составляется лишь часть программы, то все перечисленные адреса, определяющие так называемое *распределение памяти*, могут быть заданы, чтобы эта часть была согласована с другими. Задать размещение исходных и окончательных данных (явно фигурирующих в алгоритме) проще всего в виде

таблицы соответствия между буквенными обозначениями и адресами. Так, для примера 1 такая таблица имеет вид:

a	0041
b	0042
c	0043
x	0050

При этом буква a продолжает оставаться связанной с некоторым числовым значением, которое она представляет в формуле (1). Будем говорить, что буква a *обозначает* это числовое значение и *соответствует* адресу 0041 или *заменяет* его.

Заметим, что буквенное обозначение само по себе не имеет ничего общего с числом, которое оно обозначает. Точно так же адрес ячейки (являющийся целым числом) не имеет ничего общего с числом, которое помещено в ячейку с этим адресом. В обоих случаях речь идет об условных обозначениях, которые временно в определенной ситуации связываются со значениями, которые они представляют. Буквенные обозначения более привычны и удобны человеку. Восьмеричные обозначения более естественны для машины. В остальном они совершенно эквивалентны. Запись

$$x := a + b$$

означает: «число, обозначенное буквой a , сложить с числом, обозначенным буквой b , и результат обозначить буквой x ». Запись

001 0041 0042 0050

означает: «число, хранящееся в ячейке с адресом 0041, сложить с числом, хранящимся в ячейке с адресом 0042, и результат поместить в ячейку с адресом 0050». Аналогия, выраженная табличкой:

обозначить	поместить в ячейку
буква	адрес
a	0041
x	0050
...	...

становится предельно ясной.

Пример 2.

$$y := ax^2 + bx + c,$$

распределение памяти задается таблицей

a	0041
b	0042
c	0043
x	0050
y	0051

(2)

В этом примере последовательно вычислять и суммировать члены выражения уже не выгодно ни с точки зрения количества операций, ни с позиций экономии числа рабочих ячеек. Разумнее всего воспользоваться схемой Горнера, в которой порядок вычислений определяется записью

$$y := (ax + b)x + c.$$

Все вычисления можно вести в той же ячейке 0051, куда будет помещен окончательный результат (в действительности вычисления производятся в арифметическом устройстве, использованный оборот речи означает, что результат каждой операции засылается в одну и ту же ячейку памяти).

Программа будет иметь вид:

Программа 3

Л	π	КОП	A ₁	A ₂	A ₃	Примечание
0001	0	05	0041	0050	0051	ax
0002	0	01	0051	0042	0051	$ax + b$
0003	0	05	0051	0050	0051	$(ax + b)x$
0004	0	01	0051	0043	0051	$y := (ax + b)x + c$

Раз уж мы договорились о соответствии обозначений, выраженном табличкой (2), то ту же самую программу можно записать и в буквенных обозначениях вместо цифровых обозначений адресов. Заодно и коды операций обозначим символически (с этой целью в гл. 5 для каждой машинной операции было указано символическое обозначение ее кода). Далее, адрес первой команды обозначим буквой, например, p . В нашем случае $p = 0001$. Тогда адреса других команд можно обозначить $p + 1$, $p + 2$,...

Программа примет вид:

Программа 4

Л	π	КОП	A ₁	A ₂	A ₃	Примечание
p	0	×	a	x	y	ax
$p+1$	0	+	y	b	y	$ax + b$
$p+2$	0	×	y	x	y	$(ax + b)x$
$p+3$	0	+	y	c	y	$y := (ax + b)x + c$

(при такой записи для π лучше завести отдельную графу).

В дальнейшем мы, как правило, будем записывать программы только в символических (буквенных) обозначениях. Однако надо твердо помнить, что в памяти машины и на ее регистрах коды команд и чисел представляются в виде наборов нулей и единиц. В § 1 гл. 5 мы уже договорились, что такие наборы для большей компактности мы записываем в восьмеричном коде. Восьмеричный и двоичный коды находятся между собой во взаимно однозначном соответствии.

Сейчас, переходя к символическим обозначениям, мы делаем следующий шаг на пути к более наглядной записи. Но здесь нас подстерегает опасность — между символическими и восьмеричными (и двоичными) обозначениями нет естественного взаимно однозначного соответствия. Поэтому не всегда может быть ясно, чему именно соответствует тот или иной символ. Чтобы исключить возможность разных толкований, примем следующие

С о г л а ш е н и я о з а п и с и п р о г р а м м в с и м в о л и ч е с к и х о б о з н а ч е н и я х

1. Каждая буква или буква с индексом всюду в пределах одной программы заменяет один и тот же адрес. Разные буквы обычно заменяют разные адреса, но иногда допускается совпадение адресов, соответствующих различным буквенным обозначениям.

2. Коды команд, выписанные подряд, всегда располагаются в последовательных ячейках памяти, образуя отдельный связный участок программы. Участки, написанные раздельно, могут располагаться в разных местах памяти, если их взаимное расположение особо не оговорено.

3. Если буква или иное обозначение стоит слева от кода команды или какого-нибудь иного кода, то оно заменяет адрес ячейки, в которой хранится этот код. Так, например, в программе 4 буква p заменяет адрес ячейки, в которой помещается команда

$$0 \times a x y$$

4. При программировании арифметических выражений буква, встречающаяся в этом выражении в качестве переменной, заменяет в программе адрес ячейки, в которой размещается текущее значение этой переменной. Так, в программе 4 буква a заменяет адрес ячейки, в которой хранится значение коэффициента, обозначенного той же буквой в выражении

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

5. Если адрес представлен некоторым выражением, то этот адрес равен значению выражения, которое может быть вычислено после замены всех буквенных обозначений, содержащихся в этом выражении, их числовыми эквивалентами. Так, если в программе 4 буква p заменяет адрес 0001, то выражению $p + 3$ соответствует адрес 0004.

6. Если символическое обозначение не подходит ни под одну из предыдущих категорий, то оно заменяет адрес некоторой рабочей ячейки, куда помещаются промежуточные результаты вычислений. Так, в программе 1 при переходе к буквенным обозначениям адреса 0101, 0102 и 0103 можно было бы заменить символами r_1 , r_2 и r_3 , не занятыми в программируемом выражении.

В дальнейшем мы будем постепенно расширять перечень этих соглашений.

В символических обозначениях писать программу, проверять и понимать ее гораздо легче, чем в восьмеричных обозначениях. Переход от одних обозначений к другим, т. е. от программы 4 к программе 3, когда задана таблица соответствия (2), никакой сложности не представляет и может быть выполнен механически. Впредь мы будем составлять программы только в символических обозначениях и лишь на следующем примере продемонстрируем, как переходить от этих обозначений к цифровым.

Пример 3.

$$z = \frac{(a+x)^3 + (a-y)^3}{(a+x)^2 + (a-y)^2}.$$

Продумаем порядок вычислений. Очевидно, начать можно с вычисления $a + x$ (поместим этот результат в рабочую ячейку с адресом r). Затем вычислим $(a + x)^2$. Этот результат надо поместить в другую рабочую ячейку (ее адрес запишем в виде r_1), так как значение $a + x$ в ячейке r надо сохранить для вычисления $(a + x)^3$. Как продолжить вычисления — закончить вычисление знаменателя или получить $(a + x)^3$? Разумеется, лучше сделать последнее, так как хранить промежуточный результат $a + x$, нужный только для вычисления $(a + x)^3$, далее нецелесообразно. Дальнейший порядок вычислений ясен. Получаем программу:

Программа 5

А	п	КОП	A ₁	A ₂	A ₃	Примечание
p	0	+	a	x	r	$a + x$
$p+1$	0	×	r	r	r_1	$(a + x)^2$
$p+2$	0	×	r	r_1	r	$(a + x)^3$
$p+3$	0	—	a	y	r_2	$a - y$
$p+4$	0	×	r_2	r_2	z	$(a - y)^2$
$p+5$	0	+	r_1	z	r_1	$(a + x)^2 + (a - y)^2$
$p+6$	0	×	r_2	z	r_2	$(a - y)^3$
$p+7$	0	+	r	r_2	r	$(a + x)^3 + (a - y)^3$
$p+10_8$	0	/	r	r_1	z	z

Полученный вариант программы — не единственный. В нем можно было бы переставить команды в ячейках $p + 5$ и $p + 6$, можно было бы использовать ячейку z в качестве рабочей в первой же операции, но все это не привело бы ни к сокращению количества операций, ни к уменьшению числа рабочих ячеек.

Экономии числа рабочих ячеек можно получить, если после вычисления z уже не нужны для дальнейших вычислений числа a , x и y . Тогда в качестве рабочих можно использовать ячейки, где первоначально хранятся эти числа.

Программа 6

A	π	КОП	A ₁	A ₂	A ₃	Примечание
p	0	+	a	x	x	$a + x$
p+1	0	-	a	y	y	$a - y$
p+2	0	×	x	x	a	$(a + x)^2$
p+3	0	×	x	a	x	$(a + x)^3$
p+4	0	×	y	y		$(a - y)^2$
p+5	0	×	y	z	y	$(a - y)^3$
p+6	0	+	a	z	a	$\text{знам} := (a + x)^2 + (a - y)^2$
p+7	0	+	x	y	x	$\text{числ} := (a + x)^3 + (a - y)^3$
p+10	0	/	x	a	z	$z := \text{числ} / \text{знам}$

При этом, чтобы побыстрее освободить эти ячейки, потребовалось изменить порядок операций. Однако этот резерв экономии можно использовать лишь в том случае, если заранее точно известно взаимодействие данного отрезка программы с другими частями. Это условие редко соблюдается на практике. Поэтому каждый участок программы стараются писать так, чтобы он не зависел от общей компоновки программы и от того, как составлены другие участки. Поэтому программа 5 предпочтительнее программы 6. Распределение памяти обычно делается тогда, когда все участки программы уже составлены независимо друг от друга в буквенных обозначениях. Пусть при этом выяснилось, что программа 5 работает в условиях, когда ячейки a , x и y по мере их освобождения в этом участке программы можно занимать под рабочие. Порядок операций уже не позволяет сделать этого оптимальным образом, как в программе 6. Но можно совместить ячейки r и x , r_2 и y . Таким образом, распределение памяти может быть, например, таким:

a	3001
x	3010
y	3011
z	3025

(эти адреса определились при распределении памяти под основные переменные программы),

r	3010
r ₁	4001
r ₂	3011
p	2000

Ячейка r_1 может быть совмещена с рабочими ячейками других участков программы (если они, подобно r_1 , не хранят величин, передаваемых от участка к участку). При таком распределении программа в восьмеричных обозначениях будет выглядеть так:

Программа 7

А	π	КОП	A ₁	A ₂	A ₃	Примечание
2000	0	01	3001	3010	3010	$a + x$
2001	0	05	3010	3010	4001	$(a + x)^2$
2002	0	05	3010	4001	3010	$(a + x)^3$
2003	0	02	3001	3011	3011	$a - y$
2004	0	05	3011	3011	3025	$(a - y)^2$
2005	0	01	4001	3025	4001	$знам := (a + x)^2 + (a - y)^2$
2006	0	05	3011	3025	3011	$(a - y)^3$
2007	0	01	3010	3011	3010	$числ := (a + x)^3 + (a - y)^3$
2010	0	04	3010	4001	3025	$z := числ/знам$

Пример 4.

$$\Delta := \frac{1}{2} \left(\frac{x}{y} - y \right), \quad y := y + \Delta.$$

Никаких сложностей этот пример на первый взгляд не представляет. Его программа может быть, например, такой (начиная с этого примера, мы перестаем писать заголовки граф программы):

Программа 8

p	0	$/$	x	y	Δ	x/y
$p+1$	0	$-$	Δ	y	Δ	$x/y - y$
$p+2$	0	$/$	Δ	c	Δ	$\Delta := (x/y - y) / 2$
$p+3$	0	$+$	y	Δ	y	$y := y + \Delta$

Здесь буква *c* заменяет адрес ячейки, хранящей число 2. Но чаще мы будем применять другой способ.

Соглашения о символических обозначениях адресов (продолжение)

7. Число, заключенное в угловые скобки, заменяет адрес ячейки, в которой постоянно хранится это числовое значение. Например, команду $p + 2$ в программе 8 можно написать в виде

$p + 2 \quad 0 \quad / \quad \Delta \quad \langle 2 \rangle \quad \Delta$

Это соглашение избавляет нас от необходимости вводить дополнительные буквенные эквиваленты адресов вроде буквы *c* в программе 8 и разъяснять их назначение. При распределении памяти восьмеричные адреса ячеек, хранящих числовые константы, задаются обычным способом, например

$$\begin{array}{l|l} \langle 10 \rangle & 7760 \\ \langle 1/2 \rangle & 7764 \end{array}$$

Заметим, что обозначение $\langle 2 \rangle$ отнюдь не заменяет адрес 0002. В ячейке с адресом 0002 может находиться любое представимое

в машине число, а обозначению $\langle 2 \rangle$ может соответствовать любой восьмеричный адрес. Соответствие

$$\langle 2 \rangle | 0002$$

может оказаться лишь случайным.

Второе замечание, которое можно сделать в связи с этой программой, относится к выбору одной из нескольких операций, дающих одинаковый результат. При прочих равных условиях надо выбирать ту операцию, которая выполняется быстрее. Времена выполнения операций обычно указываются в описании системы команд машины. Мы не будем их приводить (они немного различаются для разных марок машин с системой команд типа М-20), заметим только, что операции $+$, $-$ и $| - |$ выполняются в среднем быстрее всех других, $+АП$ и $-АП$ — медленнее, затем следуют $\times$, $/$ и $\sqrt{}$ (каждая из трех последних заметно медленнее предыдущей). Поэтому вместо команды

$$p+2 \quad 0 \quad / \quad \Delta \quad \langle 2 \rangle \quad \Delta$$

лучше воспользоваться командой

$$p+2 \quad 0 \quad \times \quad \Delta \quad \langle 1/2 \rangle \quad \Delta$$

а еще лучше — командой

$$p+2 \quad 0 \quad +АП \quad 0077 \quad \Delta \quad \Delta$$

или

$$p+2 \quad 0 \quad -АП \quad 0101 \quad \Delta \quad \Delta$$

Последние два варианта позволяют обойтись без ячейки, хранящей константу 2 или $1/2$ (если, конечно, такая константа не потребуется где-то в другой части программы для сложения или вычитания).

Заметим, что адреса 0077 и 0101 в этих вариантах команды не заменяются буквенными эквивалентами. Это можно было бы сделать, включив в таблицу соответствия строки

$$\begin{array}{l|l} q_1 & 0077 \\ q_2 & 0101 \end{array}$$

Но запись команды $p+2$ в виде

$$p+2 \quad 0 \quad +АП \quad q_1 \quad \Delta \quad \Delta$$

или

$$p+2 \quad 0 \quad -АП \quad q_2 \quad \Delta \quad \Delta$$

затемняла бы, а не разъясняла смысл операции.

В дальнейшем восьмеричные коды, когда потребуется их явно выписать в том или ином адресе команды, обычно будем писать, опуская начальные нули (77 вместо 0077, 0 вместо 0000, и т. п.).

Глава 7. ЦИКЛЫ

§ 1. Виды циклов

При программировании арифметических выражений и присваиваний получаются программы, каждая команда которых выполняется в процессе вычислений только один раз. Такие программы могут использоваться лишь в качестве составных частей более сложных программ. В процессе работы реальных программ почти каждый ее участок выполняется, как правило, очень много раз. Иначе не может и быть. Действительно, программа, каждая команда которой выполняется всего один раз, на любой ЭВМ выполнится за секунду или еще быстрее — таково соотношение между объемом памяти и быстродействием современных машин. На составление, проверку и подготовку ко вводу в машину такой программы программист затратит не меньше времени и труда, чем ушло бы на выполнение тех же вычислений на клавишных машинах (арифмометрах).

Ясно, что такое использование ЭВМ было бы совершенно бессмысленно. Поэтому на ЭВМ решаются лишь такие задачи, которые требуют многократного повторения однотипных действий. При каждом повторении работает одна и та же часть программы, но при других значениях исходных величин. Задание новых исходных величин осуществляется либо путем их засылки в те ячейки, откуда их извлекает данная часть программы, либо изменением в командах этой части адресов ячеек, хранящих эти величины.

Всякая часть программы (группа последовательных команд), заканчивающаяся командой передачи управления на начало этой части программы, называется *циклом*. Обычно для обеспечения работы цикла нужно выполнить несколько подготовительных операций, которые также относят к данному циклу. Аналогично, *циклом* в алгоритме называется группа последовательных шагов, завершающаяся переходом на первый шаг этой группы. И здесь, как правило, кроме шагов собственно цикла нужен подготовительный шаг или шаги.

Если внутри цикла нет команд передачи управления более ранним командам того же цикла (проще, нет других циклов), то цикл

называется *простым*. Если же внутри цикла содержатся другие циклы, то он называется *кратным* или *сложным*.

Циклы составляют основу программирования. Уметь правильно составить программу для циклов различных типов — это значит постичь азы программирования.

Существуют два основных типа циклов. Циклы первого типа не содержат операций над командами цикла или над отдельными адресами этих команд. Циклы такого типа назовем *итерационными*. Чаще всего подобные циклы реализуют какой-либо процесс последовательных приближений, например, вычисление суммы ряда, заканчивающееся при условии, что очередной член ряда (или другая величина, по которой можно судить о погрешности вычисленной суммы) становится достаточно мал. Другой пример — вычисление

суммы $\sum_{i=1}^n f(i)$, где n — заданное число, а $f(i)$ — арифметическое выражение, в котором i появляется только в качестве операнта арифметических операций.

Но возьмем такой пример, очень похожий на предыдущий: вычисление суммы $\sum_{i=1}^n x_i$, где $x_1, \dots, x_n$ — ранее вычисленные величины, располагающиеся в последовательных ячейках памяти. Если не задумываться над реализацией этого процесса на машине, то этот пример ничем не отличается от предыдущего. Его можно описать алгоритмом

S. Вычисление суммы $s = \sum_{i=1}^n x_i$.

S1. $i := 1, \quad s := 0$.

S2. $s := s + x_i, \quad i := i + 1$.

S3. Если $i \leq n$, то перейти на S2.

Обратим внимание на присваивание $s := s + x_i$, встречающееся на шаге S2 этого алгоритма. В нем участвует величина x_i , не значение, а адрес которой зависит от переменной i . Если мы попытаемся описать это присваивание командой

0 + s x_i s,

то окажется, что при разных значениях i у этой команды будут разные адреса, иначе говоря, каждому значению i должна соответствовать своя команда. Попытка унифицировать вид этой команды, потребовав, чтобы все значения x_i поочередно засылались в одну и ту же ячейку y , не спасает положения — для такой пересылки понадобится команда

0 ПЧ x_i 0 y,

также изменяющаяся с изменением i .

Циклы такого типа, в которых требуется при каждом повторении цикла менять адреса величин, участвующих в вычислениях, назовем *структурными* циклами или циклами с *переадресацией*.

Знакомство с техникой программирования циклов начнем с простых итерационных циклов.

§ 2. Программирование простых итерационных циклов

В предыдущем параграфе мы уже говорили, что к простому итерационному циклу приводит задача о вычислении суммы $\sum_{i=1}^n f(i)$, где $f(i)$ — арифметическое выражение, зависящее от i . Рассмотрим случай конкретного выражения $f(i)$.

Пример 1.

$$s := \sum_{i=1}^n \frac{1}{i}.$$

Алгоритм для этого примера нам хорошо известен:

S. Вычисление суммы $s = \sum_{i=1}^n \frac{1}{i}$.

S1. $s := 0, i := 1$.

S2. $s := s + 1/i, i := i + 1$.

S3. Если $i \leq n$, то перейти на S2.

Программирование первых двух шагов никакой сложности не представляет:

p	0	ПЧ	0	0	s	$s := 0$	} S1
$p+1$	0	ПЧ	$\langle 1 \rangle$	0	i	$i := 1$	
$p+2$	0	/	$\langle 1 \rangle$	i	r	$1/i$	} S2
$p+3$	0	+	s	r	s	$s := s + 1/i$	
$p+4$	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$	

Чтобы запрограммировать шаг S3, вспомним, что в нашем распоряжении есть две операции

0 П0 0 $p+2$ 0 Если $\omega = 0$, то перейти на $p+2$ (1)

и

0 П1 0 $p+2$ 0 Если $\omega = 1$, то перейти на $p+2$

Любая из них передает управление первой команде участка, реализующего шаг S2, и пригодна для завершения шага S3, если перед этим будет выполнена команда, в которой условие выработки соответствующего значения ω совпадает с условием $i \leq n$. Очевидно, для этого годится команда

0 — n i r $r := n - i$, если $r \geq 0$ ($n \geq i$), то $\omega := 0$

в сочетании с командой (1). Заметим, что значение r , выработанное этой командой, в дальнейших вычислениях не потребуется, поэтому адрес r можно заменить нулевым адресом. Значение разности $n - i$ никуда не зашлетя, но значение ω (0 при $n \geq i$, 1 при $n < i$) сохранится и будет использовано следующей командой для организации условного перехода. Окончательно, программа примет вид:

Программа 1

p	0	ПЧ	0	0	s	$s := 0$
$p+1$	0	ПЧ	$\langle 1 \rangle$	0	i	$i := 1$
$p+2$	0	/	$\langle 1 \rangle$	i	r	$1/i$
$p+3$	0	+	s	r	s	$s := s + 1/i$
$p+4$	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
$p+5$	0	-	n	i	0	Если $n \geq i$, то перейти на $p+2$ } S3
$p+6$	0	П0	0	$p+2$	0	

В примечаниях к командам $p+5$ и $p+6$ оставлена только информация, существенная для понимания действия этих команд. В них, правда, не сказано, что когда команда $p+5$ выработает значение $\omega = 1$ (при $i > n$), то после команды $p+6$ управление перейдет к следующей команде (в ячейке $p+7$), но это в подобных случаях всегда неявно подразумевается (так же, как в описании алгоритма подразумевается, что при $i > n$ за шагом S3 следует выход из алгоритма).

Пример 2. Вычисление квадратного корня.

Забудем на время, что в машине М-20 есть операция извлечения квадратного корня. Составим алгоритм, а затем программу извлечения корня, исходя из следующих соображений.

Пусть дано число $x > 0$ и выбрано некоторое начальное приближение $y_0 > 0$ к $\sqrt{x}$. Построим последовательность $y_0, y_1, y_2, \dots$ по следующему закону

$$y_{i+1} = \frac{1}{2} \left(y_i + \frac{x}{y_i} \right). \quad (2)$$

Для исследования поведения элементов этой последовательности рассмотрим величину

$$\alpha_i = \frac{y_i - \sqrt{x}}{y_i + \sqrt{x}}, \quad (3)$$

через которую y_i выражается по формуле

$$y_i = \frac{1 + \alpha_i}{1 - \alpha_i} \sqrt{x}.$$

Используя выражения (2) и (3), легко показать, что

$$\alpha_{i+1} = \alpha_i^2,$$

откуда по индукции следует, что

$$\alpha_i = \alpha_0^{2^i}.$$

При $x > 0$, $y_0 > 0$ справедливо неравенство

$$|\alpha_0| < 1,$$

из которого следует, что

$$\lim_{i \rightarrow \infty} \alpha_i = 0$$

и

$$\lim_{i \rightarrow \infty} y_i = \sqrt{x}.$$

Нетрудно также получить соотношение

$$y_{i+1} - \sqrt{x} = \alpha_i (y_i - y_{i+1}),$$

означающее, что погрешность приближения y_{i+1} к $\sqrt{x}$ не превосходит по абсолютной величине разность Δ_i между этим и предыдущим приближением y_i (более того, она составляет очень маленькую долю этой разности).

Все это дает основание написать следующий алгоритм для приближенного вычисления $\sqrt{x}$ с заданной погрешностью ε . В этом алгоритме, разумеется, вместо последовательности $y_0, y_1, y_2, \dots$ введем переменную y , поочередно принимающую значения элементов этой последовательности.

А. Даны числа $x > 0$ и $\varepsilon > 0$. Найти приближение y к $\sqrt{x}$, погрешность которого не превосходит ε .

A1. $y := 1$.

A2. $\Delta := \frac{1}{2} \left(\frac{x}{y} - y \right)$, $y := y + \Delta$.

A3. Если $|\Delta| > \varepsilon$, то перейти на A2.

Команды, реализующие «рабочий» шаг A2, были составлены в примере 6.4. Их надо дополнить командами, реализующими подготовку цикла к работе — шаг A1, и проверку возможности завершить работу цикла — шаг A3. Начальная команда очевидна:

0 ПЧ <1> 0 y y:=1,

а шаг A3 программируется тоже просто:

0 |—| ε Δ 0 Если $|\Delta| > |\varepsilon|$, то ($\omega := 1$)
0 П1 0 A2 0 (Если $\omega = 1$, то) перейти на A2.

Здесь A2 во втором адресе последней команды — это адрес первой команды участка, реализующего шаг A2 алгоритма. Собирая все команды вместе, получим программу:

Программа 2

p	0	$\Pi 1$	$\langle 1 \rangle$	0	y	$y := 1$	A1
$p+1$	0	/	x	y	Δ	x/y	} A2
$p+2$	0	—	Δ	y	Δ	$x/y - y$	
$p+3$	0	$+ A\Pi$	77	Δ	Δ	$\Delta := (x/y - y) / 2$	
$p+4$	0	+	y	Δ	y	$y := y + \Delta$	
$p+5$	0	$ - $	ε	Δ	0	Если $ \Delta > \varepsilon$, то	} A3
$p+6$	0	$\Pi 1$	0	$p+1$	0	перейти на A2	

§ 3. Программирование разветвлений в программах

Единственная новая проблема, с которой мы столкнулись в предыдущем параграфе, — это программирование разветвлений, т. е. таких действий, запись которых в составе алгоритма имеет вид:

Если «условие», то «действие 1», в противном случае «действие 2». (1)

Впрочем, мы имели дело лишь с очень простым частным случаем этой проблемы, когда «условие» — это просто неравенство, «действие 1» — это *перейти на шаг ...*, а «действие 2» вообще отсутствует. Рассмотрим более общий случай произвольных действий 1 и 2. Легко убедиться, что он сводится к упомянутому частному случаю. Достаточно заменить действие (1) следующим алгоритмом.

C. *Выполнение разветвляющегося действия.*

C1. *Если «условие», то перейти на C3.*

C2. *«Действие 2», выход.*

C3. *«Действие 1».*

Выход здесь означает переход на шаг, следующий в алгоритме за действием (1). Возможны другие варианты алгоритма, в которых порядок действий в записи совпадает с их порядком в (1), но при этом приходится либо вставить лишний переход, либо изменить смысл условия на противоположный. Вот один из этих вариантов:

D. *Выполнение разветвляющегося действия.*

D1. *Если «условие», то перейти на D3.*

D2. *Перейти на D4.*

D3. *«Действие 1», выход.*

D4. *«Действие 2».*

Другой вариант таков:

E. *Выполнение разветвляющегося действия.*

E1. *Если не «условие», то перейти на E3.*

E2. *«Действие 1», выход.*

E3. *«Действие 2».*

Таким образом, надо научиться программировать действия вида

Если «условие», то перейти на «номер шага»

для различных условий. Рассмотрим сначала условия, имеющие вид неравенств между двумя числами, т. е.

$$x < y, \quad (2)$$

$$x \leq y, \quad (3)$$

$$x \geq y, \quad (4)$$

$$x > y. \quad (5)$$

Очевидно, условия (4) и (5) перестановкой сторон неравенства сводятся к (3) и (2), так что достаточно рассмотреть, как программируются эти два случая. Они очень просты. Программа действия

Если $x < y$, то перейти на N

состоит из команд:

0 — x y 0 Если $x < y$, то ($\omega := 1$)
0 П1 0 N 0 (Если $\omega = 1$, то) перейти на N

Действие

— Если $x \leq y$, то перейти на N

реализуется командами:

0 — y x 0 Если $x \leq y$, то ($\omega := 0$)
0 П0 0 N 0 (Если $\omega = 0$, то) перейти на N

Наличие в системе команд М-20 операции $| - |$ делает столь же простой проверку условий вида $|x| < |y|$ или $|x| \leq |y|$. Соответствующие участки программ имеют вид:

0 $| - |$ x y 0 Если $|x| < |y|$, то ($\omega := 1$)
0 П1 0 N 0 перейти на N

и

0 $| - |$ y x 0 Если $|x| \leq |y|$, то ($\omega := 0$)
0 П0 0 N 0 перейти на N

Рассмотрим теперь условия вида $x = y$ и $x \neq y$. Их можно переписать в виде $|x - y| \leq 0$ и $|x - y| > 0$. Отсюда получаем для действия

Если $x = y$, то перейти на N

программу

0 — x y r x-y
0 $| - |$ 0 r 0 Если $|x - y| \leq 0$, то ($\omega := 0$)
0 П0 0 N 0 перейти на N (6)

Заменив в последней команде код операции П0 на П1, получим, очевидно, программу для действия

Если $x \neq y$, то перейти на N.

Иногда возможно реализовать эти действия проще, используя операцию $\neq$, вырабатывающую признак $\omega = 1$ только в том случае, когда машинные представления ее операндов совпадают. Так, действие

Если $x = y$, то перейти на N

реализуется командами:

$$\begin{array}{llllll} 0 & \neq & x & y & 0 & \text{Если } x=y, \text{ то } (\omega := 1) \\ 0 & \Pi 1 & 0 & N & 0 & \text{перейти на N.} \end{array} \quad (7)$$

Следует считаться с тем, что в машине числа представляются и арифметические операции выполняются приближенно. Поэтому, если некоторые вычисления теоретически дают одинаковые значения x и y , на практике может оказаться, что машинные представления этих значений слегка различаются. Если мы согласны все же считать такие значения равными, то ни программа (6), ни программа (7) нас не удовлетворит. Последнюю исправить уже невозможно, так как она написана в расчете на точное совпадение машинных представлений x и y . Программа же (6) переделывается сравнительно просто — достаточно заменить условие $|x - y| \leq 0$ условием $|x - y| \leq \varepsilon$, где ε — маленькое число. Какое число следует считать маленьким? Оно не может быть одним и тем же на все случаи жизни и должно выбираться в зависимости от ожидаемого значения x и y . Мало различающиеся числовые значения в машинном представлении — это значения, у которых совпадают порядки и большинство разрядов мантиссы, кроме самых младших. Единича младшего разряда мантиссы нормализованного числового значения — это число, приблизительно в 2^{36} раз меньшее, чем само это значение. Следовательно, если ожидаемое значение x , при котором оно должно оказаться равным y , равно x_0 , то в качестве ε , о котором шла речь, следует принять число несколько большее, чем $x_0 \cdot 2^{-36}$, например, $x_0 \cdot 2^{-34}$, или $x_0 \cdot 2^{-32}$, или даже еще большее число, если значение x_0 нам точно неизвестно. Вообще же следует избегать алгоритмов, работа которых критически зависит от возможного равенства или неравенства приближенных чисел.

Пример 1. Пусть требуется вычислить сумму квадратных корней из членов арифметической прогрессии 0,01, 0,02, ..., 0,99, 1,00. Рассмотрим такой вариант алгоритма

A. *Вычисление суммы $\sqrt{0,01} + \sqrt{0,02} + \dots + \sqrt{1,00}$.*

A1. $s := 0, x := 0,01$.

A2. $s := s + \sqrt{x}, x := x + 0,01$.

A3. *Если $x \leq 1$, то перейти на A2.*

Формально он правилен, так как в последовательности значений, присваиваемых переменной x , должно встретиться значение 1 и при этом значении произойдет последний переход на шаг A2. Но

если этот алгоритм (вернее, соответствующая программа) выполняется на машине, то из-за погрешностей вычислений сотое по порядку значение, присвоенное переменной x (засланное в ячейку x), может оказаться чуть больше 1 и на шаг А2 мы уже не попадем. Вычисленное значение s окажется на 1 (грубо говоря) меньше требуемого. Поэтому условие в шаге А3 следует заменить, например, на $x \leq 1,005$.

У п р а ж н е н и е 1. Написать соответствующую программу.

Машина М-20 обладает следующим свойством. Целые числа n из диапазона $-2^{36} < n < 2^{36}$ могут быть в ней представлены точно (это значит, что если двоичный порядок такого числа равен p , $p \leq 36$, то только в старших p разрядах мантиссы могут встречаться цифры, отличные от нуля, младшие же $36 - p$ цифр равны нулю). Более того, если над такими числами выполняются арифметические операции, дающие целый же результат из того же диапазона, то и этот результат оказывается представленным точно.

Благодаря этому свойству программы типа программы 7.2.1, где используется сравнение целых чисел n и i , работают правильно (если $n < 2^{36}$).

П р и м е р 2. Решение уравнения степени не выше второй.

Алгоритм решения уравнения второй (или более низкой) степени был описан в § 3 гл. 4. Возьмем его в несколько измененном виде, заменив текстовую переменную s числовой переменной, которой будем присваивать значения: 0 — при отсутствии корней, 1 — при наличии одного корня, 2 — при наличии двух различных вещественных корней, 3 — в случае комплексных корней, 4 — в случае одного кратного корня, 5 — если уравнение обращается в тождество.

Видоизмененный алгоритм имеет вид:

Q. Решение уравнения $ax^2 + bx + c = 0$.

Q1. Если $a = 0$, то перейти на Q7.

Q2. $x_1 := -b/2a$, $d := x_1^2 - c/a$.

Q3. Если $d \leq 0$, то перейти на Q5.

Q4. $d := \sqrt{d}$, $x_2 := x_1 - d$, $x_1 := x_1 + d$, $s := 2$, выход.

Q5. Если $d < 0$, то $x_2 := \sqrt{-d}$, $s := 3$, выход.

Q6. $s := 4$, выход.

Q7. Если $b \neq 0$, то $x_1 := -c/b$, $s := 1$, выход.

Q8. Если $c \neq 0$, то $s := 0$, выход.

Q9. $s := 5$.

При программировании следует применить описанный в начале параграфа прием и заменить, например, шаг Q5 двумя шагами:

Q5.1. Если $d \geq 0$, то перейти на Q6.

Q5.2. $x_2 := \sqrt{-d}$, $s := 3$, выход.

С учетом этого замечания программа будет иметь вид:

Программа 1						
Q1	0	$\neq$	a	0	0	Если $a = 0$, то ($\omega := 1$)
	0	П1	0	Q7	0	перейти на Q7
Q2	0	+АП	0101	a	x_1	$2a$
	0	/	b	x_1	x_1	$b/2a$
	0	—	0	x_1	x_1	$x_1 := -b/2a$
	0	$\times$	x_1	x_1	d	x_1^2
	0	/	c	a	x_2	c/a
	0	—	d	x_2	d	$d := x_1^2 - c/a$
Q3	0	—	0	d	0	Если $d \leq 0$, то ($\omega := 0$)
	0	П0	0	Q5	0	перейти на Q5
Q4	0	$\sqrt{}$	d	0	d	$d := \sqrt{d}$
	0	—	x_1	d	x_2	$x_2 := x_1 - d$
	0	+	x_1	d	x_1	$x_1 := x_1 + d$
	0	ПБ	$\langle 2 \rangle$	выход	s	$s := 2$, выход
Q5	0	—	d	0	0	Если $d \geq 0$, то ($\omega := 0$)
	0	П0	0	Q6	0	перейти на Q6
	0	—	0	d	d	$-d$
	0	$\sqrt{}$	d	0	x_2	$x_2 := \sqrt{-d}$
	0	ПБ	$\langle 3 \rangle$	выход	s	$s := 3$, выход
Q6	0	ПБ	$\langle 4 \rangle$	выход	s	$s := 4$, выход
Q7	0	$\neq$	b	0	0	Если $b = 0$, то ($\omega := 1$)
	0	П1	0	Q8	0	перейти на Q8
	0	/	c	b	x_1	c/b
	0	—	0	x_1	x_1	$x_1 := -c/b$
	0	ПБ	$\langle 1 \rangle$	выход	s	$s := 1$, выход
Q8	0	$\neq$	c	0	0	Если $c = 0$, то ($\omega := 1$)
	0	П1	0	Q9	0	перейти на Q9
	0	ПБ	0	выход	s	$s := 0$, выход
Q9	0	ПЧ	$\langle 5 \rangle$	0	s	$s := 5$
выход						

В этой программе мы отказались от сплошной нумерации команд и ставили обозначения Q1, Q2, ..., Q9 на месте адресов первых команд блоков (так для краткости будем называть участки программ), реализующих шаги алгоритма с теми же номерами. Этого оказалось достаточно, чтобы можно было записать почти все команды перехода, встретившиеся в программе. Исключение составляют команды перехода, в которых адрес перехода (A_2) заменен словом *выход*. Под этим понимается адрес команды, следующей за последней командой программы (иначе говоря, адрес Q9 + 1). В связи с этим необходимо сделать некоторые разъяснения.

Уже говорилось, что любой алгоритм можно и следует рассматривать, как часть — один шаг — какого-то другого более общего

алгоритма, решающего более крупную задачу. С этой точки зрения действие *выход* в описании алгоритма эквивалентно действию *перейти на следующий шаг охватывающего алгоритма*. Точно так же программы, которые мы пишем, — это лишь части более крупных программ.

В рассматриваемом примере эта более крупная программа перед началом работы программы 1 должна подготовить ее исходные данные (коэффициенты a , b и c), а после окончания — использовать, обработать ее результаты (значения s , x_1 , x_2). Выход из нашей программы естественно заключается в передаче управления команде, следующей за последней командой программы. В программе это отражено следующим образом. Действию *выход* соответствует команда

0 ПБ 0 *выход* Q,

где адрес *выход* — это и есть адрес упомянутой команды. Благодаря особенности операции ПБ машины М-20 эту операцию удастся совместить с присваиванием требуемого значения переменной s .

Рассмотрим теперь возможности улучшения программы. Обратим внимание на команду с адресом Q6. Непосредственно перед ней стоит команда безусловной передачи управления. Это значит, что команда Q6 может быть выполнена только, если ей будет передано управление какой-то другой командой. Единственная команда, которая может это сделать, — это команда с адресом $Q5 + 1$ (никакая другая команда не имеет Q6 своим вторым адресом). Команда $Q5 + 1$ только и делает, что передает управление команде Q6 при $\omega = 0$. Немедленно после этого команда Q6 передает управление по адресу *выход* и одновременно засылает число 4 в ячейку s . Можно совместить все эти действия в одной команде $Q5 + 1$, записав ее в виде:

0 П0 (4) *выход* s

и выбросив команду Q6. Правда, при этом значение 4 будет присвоено переменной s и при $\omega = 1$ ($d < 0$), но это не отразится на работе блока Q5, а последняя команда этого блока выработает другое, нужное значение s .

Аналогичная ситуация возникает с командами $Q8 + 1$ и Q9, с той разницей, что последняя не передает управления, а осуществляет только засылку значения в ячейку s . Если мы совместим эту пересылку с передачей управления команде, следующей за командой Q9 (т. е. по адресу *выход*), в команде $Q8 + 1$, то окончательные результаты работы программы не изменятся. Заключительные команды программы будут выглядеть так:

Q8	0 $\neq$	c	0	0	} $s := 5$, если $c = 0$, то (перейти на) выход
	0 П1	(5)	<i>выход</i>	s	
	0 ПБ	0	<i>выход</i>	s	

выход

Впрочем, последнюю команду можно заменить командой

0 ПЧ 0 0 s s:=0

поскольку передача управления по адресу *выход* (т. е. следующей команде) произойдет автоматически.

Но есть и другой путь упрощения той же группы команд. Команды Q8 + 1 и Q8 + 2 (в исходном варианте программы) передают управление команде Q9 при $\omega = 1$ и команде *выход* — в противном случае (т. е. при $\omega = 0$). Но команда Q9 стоит непосредственно за второй из этих команд. Те же переходы можно организовать одной командой условной передачи управления с кодом операции П0 и вторым адресом *выход*. Засылку значения 0 в ячейку s, которую выполняет команда Q8 + 2, можно тоже поручить этой новой команде. То, что эта засылка будет производиться и при $\omega = 1$, не страшно, так как команда Q9 (в старом виде), которая будет работать вслед за этим, тут же исправит положение. Этот вариант сокращения приведет к командам

Q8	0	$\neq$	c	0	0	s:=0, если c $\neq$ 0,
	0	П0	0	выход	s	то выход
Q9	0	ПЧ	<5>	0	s	s:=5.

Есть еще одна возможность немного сократить программу. После вычисления d алгоритм фактически делится не на две ($d \leq 0$ и $d > 0$), а на три ветви ($d < 0$, $d = 0$ и $d > 0$), и только наша неспособность описывать такое тройное разветвление заставляет нас делать две последовательные проверки: на шаге Q3 и в начале шага Q5. Однако последовательность выявления возможных трех случаев безразлична. Заметим теперь, что последняя команда вычисления d — это команда вычитания, которая выработает значение $\omega = 1$, если $d < 0$, и $\omega = 0$, если $d \geq 0$. Поэтому, не производя никаких дополнительных операций, можно выделить случай $d < 0$, а два оставшихся случая разделить позднее.

Наконец, в команде Q2 операцию

0 +АП 0101 a x₁

целесообразно заменить более быстрой операцией

0 + a a x₁

Со всеми усовершенствованиями, о которых шла речь, программа 1 примет вид:

Программа 2

Q1	0	$\neq$	a	0	0	Если $a=0$, то
	0	П1	0	Q7	0	перейти на Q7
Q2	0	+	a	a	x_1	$2a$
	0	/	b	x_1	x_1	$b/2a$
	0	—	0	x_1	x_1	$x_1 := -b/2a$
	0	$\times$	x_1	x_1	d	x_1^2
	0	/	c	a	x_2	c/a
	0	—	d	x_2	d	$d := x_1^2 - c/a$, если $d \geq 0$, то
	0	П0	0	Q3	0	перейти на Q3
Q5	0	—	0	d	d	$-d$
	0	$\sqrt{}$	d	0	x_2	$x_2 := \sqrt{-d}$
	0	ПБ	$\langle 3 \rangle$	выход	s	$s := 3$, выход
Q3	0	—	0	d	0	} $s := 4$, если $d \leq 0$, то
	0	П0	$\langle 4 \rangle$	выход	s	
Q4	0	$\sqrt{}$	d	0	d	$d := \sqrt{d}$
	0	—	x_1	d	x_2	$x_2 := x_1 - d$
	0	+	x_1	d	x_1	$x_1 := x_1 + d$
	0	ПБ	$\langle 2 \rangle$	выход	s	$s := 2$, выход
Q7	0	$\neq$	b	0	0	Если $b=0$, то
	0	П1	0	Q8	0	перейти на Q8
	0	/	c	b	x_1	c/b
	0	—	0	x_1	x_1	$x_1 := -c/b$
	0	ПБ	$\langle 1 \rangle$	выход	s	$s := 1$, выход
Q8	0	$\neq$	c	0	0	} $s := 0$, если $c \neq 0$, то
	0	П0	0	выход	s	
Q9	0	ПЧ	$\langle 5 \rangle$	0	s	$s := 5$
выход						

В связи с изменением последовательности шагов изменилась их логическая часть — вид и содержание проверяемых условий. Отказ от сплошной нумерации шагов упростил переделку программы в ходе ее составления. Так, например, если бы все команды были пронумерованы подряд (от p до $p + 34$, в прежнем варианте и от p до $p + 31$ — в новом, не считая адреса *выход*, который был бы равен $p + 35$ и $p + 32$), то при переделке пришлось бы изменить вторые адреса всех команд перехода (например, в команде $p + 1$ (Q1 + 1) адрес $p + 24$ — на адрес $p + 22$ (в обоих случаях — Q7), адрес $p + 35$ (*выход*) — всюду на адрес $p + 32$ и т. п.).

§ 4. Итерационные циклы (продолжение)

Рассмотрим еще примеры итерационных циклов.

Пример 1. Решение уравнений методом линейной интерполяции.

Пусть на отрезке $[a, b]$ задана непрерывная функция $f(x)$. Для нас это означает, что вычисление $f(x)$ по любому заданному x ,

$a \leq x \leq b$, можно рассматривать как элементарное действие алгоритма, о котором пойдет речь. Пусть в точках a и b функция $f(x)$ принимает значения разных знаков. Тогда отрезок $[a, b]$ должен содержать по крайней мере один корень функции $f(x)$. Требуется найти такой корень (конечно, приближенно). Воспользуемся следующим методом. На графике функции $f(x)$ соединим точки, соответствующие концам отрезка, прямой линией. Пусть эта линия пересекает ось x в точке c . Вычислим $f(c)$. Если окажется, что $f(c) = 0$, то корень найден (на практике будем пользоваться условием $|f(c)| < \varepsilon$, где $\varepsilon > 0$ — малое число). В противном случае сравним знаки $f(c)$ и $f(a)$. Если они совпадают, то отрезок $[c, b]$, а в противном случае — отрезок $[a, c]$ содержит корень функции $f(x)$. К выбранному таким образом отрезку применим тот же прием. Так будем продолжать, пока длина отрезка, содержащего корень функции, остается больше малого числа $\delta > 0$.

Все сказанное — это уже почти готовый алгоритм, осталось придать ему привычную для нас форму. Введем обозначения u , v и w для значений функции $f(x)$ в точках a , b и c соответственно. Алгоритм может быть записан, например, так.

1. Алгоритм отыскания корня c функции $f(x)$ на отрезке $[a, b]$ методом линейной интерполяции.
11. $u := f(a)$, $v := f(b)$.
12. $c := a + \frac{(b-a)u}{u-v}$, $w := f(c)$.
13. Если $|w| < \varepsilon$, то выход.
14. Если $u \cdot w > 0$, то $a := c$, $u := w$,
в противном случае $b := c$, $v := w$.
15. Если $|b - a| > \delta$, то перейти на 12.

Формула для c на шаге 12 вытекает из очевидного геометрического соотношения:

$$\frac{u}{c-a} = \frac{-v}{b-c}. \quad (1)$$

Неравенство $|b - a| > \delta$ вместо $b - a > \delta$ на шаге 15 позволяет применить алгоритм, даже если вначале было $b < a$. Проверка, что знаки $f(a)$ и $f(b)$ различны, не предусмотрена. При желании можно было бы ввести дополнительную переменную q и вставить после шага 11 шаг

- 11.1. Если $u \times v > 0$, то $q := 0$, выход, в противном случае $q := 1$.

Тогда выход из алгоритма с $q = 0$ означал бы неприменимость алгоритма.

Сходимость или, лучше сказать, конечность процесса, описываемого этим алгоритмом, вытекает из следующих соображений. Не-

прерывная функция $f(x)$ на замкнутом отрезке ограничена некоторой константой M , так что в любой момент работы алгоритма $|u| \leq M$, $|v| \leq M$. С другой стороны, новые значения, присваиваемые переменным u и v , — это значения w , которые удовлетворяют неравенству $|w| \geq \varepsilon$. Отсюда и из соотношения (1) следует, что новая длина l_i отрезка $[a, b]$ (т. е. $|c - a|$ или $|b - c|$) связана со старой ($l_{i-1} = |b - a|$) неравенством

$$\frac{l_i}{l_{i-1}} \leq \frac{M}{M + \varepsilon},$$

откуда $l_i \leq l_0 \left(\frac{M}{M + \varepsilon} \right)^i$, и рано или поздно наступит момент, когда окажется, что $l_i \leq \delta$, и алгоритм закончит свою работу.

Напишем программу, соответствующую алгоритму 1. Мы должны предположить, что программу, вычисляющую значение $f(x)$ по заданному x , мы написать умеем. Тогда вся программа может быть такой:

Программа 1

11	программа вычисления $f(a)$ в ячейке u программа вычисления $f(b)$ в ячейке v					
12	0	—	b	a	c	$b - a$
	0	$\times$	c	u	c	$(b - a) u$
	0	—	u	v	w	$u - v$
	0	/	c	w	c	$(b - a) u / (u - v)$
	0	+	a	c	c	$c := a + (b - a) u / (u - v)$
	программа вычисления $f(c)$ в ячейке w					
13	0	—	w	ε	0	Если $ w < \varepsilon $, то
	0	П1	0	выход	0	выход
14	0	$\times$	u	w	r	uw
	0	—	0	r	0	Если $uw \leq 0$, то ($w := 0$)
	0	П0	0	14'	0	перейти на 14'
	0	ПЧ	c	0	a	$a := c$
	0	ПБ	w	15	u	$u := w$, перейти на 15
14'	0	ПЧ	c	0	b	$b := c$
	0	ПЧ	w	0	v	$v := w$
						$b - a$
15	0	—	b	a	r	
	0	—	δ	r	0	Если $ b - a > \delta $, то
	0	П1	0	12	0	перейти на 12

Как и в предыдущем примере, адрес *выход* — это адрес команды, следующей за последней командой программы (т. е. $\text{выход} = 15 + 3$).

Программа вычисления функции $f(x)$ может быть очень длинной. Нужно ли выписывать ее три раза, чтобы завершить составление

программы нашего примера? Конечно, нет. В подобных случаях используется следующий прием. Программа вычисления $f(x)$ пишется один раз. Пусть она берет свой аргумент x из ячейки, адрес которой мы по традиции обозначим буквой x , и оставляет результат (значение $f(x)$) в ячейке y . Обозначим адрес первой команды этой программы буквой f_n . Добавим в конце программы еще одну ячейку, адрес которой обозначим f_k , понимая под этим *выход из f* . Эта ячейка служит для засылки в нее команды возврата по окончании вычисления функции. Например, если $f(x) = \sqrt{x^3 + 1} - 2\sqrt{x + 1}$, то программа вычисления $f(x)$ вместе с дополнительной ячейкой может иметь вид (предполагаем, что значение x можно не сохранять):

Программа 2

f_n	0	×	x	x	y	x^2
	0	×	y	x	y	x^3
	0	+	y	$\langle 1 \rangle$	y	$x^3 + 1$
	0	√	y	0	y	$\sqrt{x^3 + 1}$
	0	+	x	$\langle 1 \rangle$	x	$x + 1$
	0	√	x	0	x	$\sqrt{x + 1}$
	0	+	x	x	x	$2\sqrt{x + 1}$
	0	—	y	x	y	$y := \sqrt{x^3 + 1} - 2\sqrt{x + 1}$
f_k	0	0	0	0	0	выход из f

Теперь участок основной программы, реализующий вычисление $f(x)$ при заданном значении x с засылкой результата в заданную ячейку, например, вычисление $f(c)$ в ячейке w , может иметь вид

	0	ПЧ	c	0	x	$x := c$
	0	ПВ	p_1	f_n	f_k	Перейти на $f(x)$ с возвратом
p_1	0	ПЧ	y	0	w	$w := f(c)$

Эти три команды должны быть вставлены в программу непосредственно перед командой I3. Команда ПВ зайдет в ячейку f_k команду

0 ПВ 0 p_1 0

и передаст управление первой команде (f_n) программы вычисления $f(x)$. Когда эта программа завершит вычисление $f(x)$ в ячейке y , команда, записанная в ячейку f_k , передаст управление команде p_1 и вычисленное значение $f(c)$ перешлется в ячейку w . Аналогично, блок, реализующий шаг II, должен состоять из команд:

II	0	ПЧ	a	0	x	$x := a$
	0	ПВ	p_2	f_n	f_k	Перейти на $f(x)$ с возвратом
p_2	0	ПЧ	y	0	u	$u := f(a)$
	0	ПЧ	b	0	x	$x := b$
	0	ПВ	p_3	f_n	f_k	Перейти на $f(x)$ с возвратом
p_3	0	ПЧ	y	0	v	$v := f(b)$

На этом примере хорошо видны возможности, предоставляемые программисту операций перехода с возвратом (ПВ). Здесь же мы получили первое представление о подпрограммах — в этом примере подпрограммой была программа вычисления $f(x)$ при произвольном x .

§ 5. Приемы вычисления конечных сумм. Чистка циклов

Довольно часто в практике программирования возникают задачи вычисления значений различных конечных сумм: полиномов, отрезков степенных рядов, коэффициенты которых заданы явными выражениями в зависимости от номера члена, и т. п. С основными приемами, используемыми при программировании таких задач, познакомимся на следующем примере.

Пример 1. Вычислить значение многочлена (отрезка степенного ряда для e^x)

$$y = \sum_{i=0}^n \frac{x^i}{i!} = 1 + \frac{x}{1} + \frac{x^2}{1 \cdot 2} + \dots + \frac{x^n}{1 \cdot 2 \cdot \dots \cdot n}. \quad (1)$$

Здесь x и n — заданные числа.

Обозначим общий член ряда через u_i , так что

$$y = \sum_{i=0}^n u_i, \quad u_i = \frac{x^i}{i!}.$$

Общий алгоритм вычисления суммы, конечно, применим и здесь:

A. *Вычисление суммы (1).*

A1. $i := 0, s := 1.$

A2. $i := i + 1, u := \frac{x^i}{i!}, s := s + u.$

A3. *Если $i < n$, то перейти на A2.*

Если каждый член ряда вычислять независимо, это потребует довольно громоздких вычислений. В частности, в выражение общего члена ряда входит факториал $i!$. Целочисленная функция *факториал* сама может быть определена циклическим алгоритмом:

F. *Вычисление $M = m!$ для $m \geq 0$.*

F1. $k := 0, M := 1.$

F2. *Если $k \geq m$, то выход.*

F3. $k := k + 1, M := k \cdot M$, *перейти на F2.*

Никакого другого, существенно отличающегося от алгоритма F, способа определения и способа вычисления этой функции не существует. Часто приводимое определение факториала

$$m! = 1 \cdot 2 \cdot \dots \cdot m$$

фактически является намеком на тот же алгоритм и, кроме того, требует дополнительного уточнения для случая $m = 0$.

Вычисление x^i также представляет собой циклический процесс. Легко видеть, что при независимом вычислении членов суммы (1) мы будем выполнять много одинаковых вычислений. Например, при вычислении значения $i!$, входящего в выражение для u_i , мы вычислим значение $(i-1)!$, которое уже вычислялось для получения u_{i-1} , и умножим его на i . Аналогично, значение x^i , входящее в u_i , получается из x^{i-1} , необходимого для вычисления u_{i-1} , умножением на x .

При разумной организации алгоритма вычисления y мы не должны по нескольку раз повторять одни и те же вычисления. Чтобы избавиться от них, попытаемся выразить очередной член суммы u_i через предыдущий член u_{i-1} . Для этого найдем их отношение

$$\frac{u_i}{u_{i-1}} = \frac{x^i \cdot (i-1)!}{i! \cdot x^{i-1}} = \frac{x}{i},$$

откуда

$$u_i = \frac{x \cdot u_{i-1}}{i}.$$

Таким образом, зная член u_{i-1} , мы можем вычислить следующий член u_i , выполнив всего две операции. Приходим к следующему варианту алгоритма, очищенному от повторяющихся вычислений.

В. Вычисление суммы $y = \sum_{i=0}^n \frac{x^i}{i!}$.

B1. $i := 0, u := 1, s := 1$.

B2. $i := i + 1, u := x \cdot u / i, s := s + u$.

B3. Если $i < n$, то перейти на B2.

Напишем соответствующую программу.

Программа 1

B1	0	ПЧ	0	0	i	$i := 0$
	0	ПЧ	$\langle 1 \rangle$	0	u	$u := 1$
	0	ПЧ	$\langle 1 \rangle$	0	s	$s := 1$
B2	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
	0	$\times$	x	u	u	$x \cdot u$
	0	/	u	i	u	$u := x \cdot u / i$
	0	+	s	u	s	$s := s + u$
B3	0	—	i	n	0	Если $i < n$, то ($\omega := 1$)
	0	П1	0	B2	0	перейти на B2

Пример 2. Вычислить сумму отрезка степенного ряда для $\sin x$:

$$s = \sum_{i=0}^n (-1)^i \frac{x^{2i+1}}{(2i+1)!}. \quad (2)$$

В отличие от предыдущего примера будем считать, что номер n последнего члена суммы не задан явно, а определяется требованием, чтобы последний член u_n не превосходил по абсолютной величине заданное число ε . Здесь

$$u_i = (-1)^i \frac{x^{2i+1}}{(2i+1)!}.$$

Опять полезно рассмотреть отношение

$$\frac{u_i}{u_{i-1}} = \frac{(-1)^i x^{2i+1} \cdot (2i-1)!}{(2i+1)!(-1)^{i-1} x^{2i-1}} = -\frac{x^2}{2i(2i+1)},$$

откуда

$$u_i = u_{i-1} \frac{-x^2}{2i(2i+1)}.$$

Здесь множитель $-x^2$ (обозначим его через v) не зависит от i , а знаменатель $2i(2i+1)$ вычисляется в зависимости от i сравнительно просто — без циклов. Алгоритм вычисления суммы (2), в котором цикл вычисления самой суммы совмещен с циклами, необходимыми для вычисления x^{2i+1} , $(2i+1)!$ и даже $(-1)^i$, оказывается таким:

S. *Вычисление суммы (2).*

S1. $i := 0, u := x, s := x, v := -x^2$.

S2. $i := i + 1, u := \frac{uv}{2i(2i+1)}, s := s + u$.

S3. *Если $|u| > \varepsilon$, то перейти на S2.*

Следует обратить внимание на то, что вычисление $v := -x^2$ вынесено в подготовительную часть цикла (шаг S1), чтобы не повторять его каждый раз заново. Перейти от алгоритма к машинной программе теперь уже не трудно.

Программа 2

S1	0	ПЧ	0	0	i	$i := 0$
	0	ПЧ	x	0	u	$u := x$
	0	ПЧ	x	0	s	$s := x$
	0	×	x	x	v	x^2
	0	—	0	v	v	$v := -x^2$
S2	0	+	i	<1>	i	$i := i + 1$
	0	+	i	i	r	$2i$
	0	+	r	<1>	r_1	$2i + 1$
	0	×	r	r_1	r	$r := 2i(2i + 1)$
	0	×	u	v	u	uv
	0	/	u	r	u	$u := uv/r$
	0	+	s	u	s	$s := s + u$
	0	—	ε	u	0	Если $ u > \varepsilon $, то ($\omega := 1$)
S3	0	П1	0	S2	0	перейти на S2

Обращает на себя внимание то обстоятельство, что из шести команд, затрачиваемых на вычисление u , четыре команды уходят

на вычисление знаменателя выражения для u (в это число мы включили и команду вычисления очередного значения i , поскольку нигде, кроме этого знаменателя, значение переменной i не используется). Нельзя ли упростить вычисление знаменателя? Можно, и по существу тем же самым приемом. Обозначим этот знаменатель через r_i :

$$r_i = 2i(2i + 1) = 4i^2 + 2i \quad (3)$$

и постараемся выразить значение r_i через r_{i-1} . Отношение r_i к r_{i-1} оказывается сложным, поэтому попробуем рассмотреть разность

$$d_i = r_i - r_{i-1} = 4i^2 + 2i - 4(i-1)^2 - 2(i-1) = 8i - 2, \quad (4)$$

откуда

$$r_i = r_{i-1} + d_i,$$

а d_i выражается через d_{i-1} уже совсем просто:

$$d_i = d_{i-1} + 8.$$

Таким образом, очередное значение r_i вычисляется через предыдущее с помощью всего двух операций (одна из них вычисляет d_i через d_{i-1}), а надобность в переменной i отпала. Чтобы начать вычисления, нужно знать начальные значения переменных r и d (при $i = 0$). Из формул (3) и (4) получаем $r_0 = 0$, $d_0 = -2$. Итак, еще один вариант алгоритма может быть записан в следующем виде.

S'. Вычисление суммы (2).

S'1. $d := -2$, $r := 0$, $u := x$, $v := -x^2$.

S'2. $d := d + 8$, $r := r + d$, $u := uv/r$, $s := s + u$.

S'3. Если $|u| > \varepsilon$, то перейти на S'2.

Здесь очень важна правильная последовательность вычислений на шаге S'2. Алгоритму S' соответствует следующая программа.

Программа 3

S'1	0	—	0	<2>	d	$d := -2$
	0	ПЧ	0	0	r	$r := 0$
	0	ПЧ	x	0	u	$u := x$
	0	ПЧ	x	0	s	$s := x$
	0	×	x	x	v	x^2
	0	—	0	v	v	$v := -x^2$
S'2	0	+	d	<8>	d	$d := d + 8$
	0	+	r	d	r	$r := r + d$
	0	×	u	v	u	uv
	0	/	u	r	u	$u := uv/r$
	0	+	s	u	s	$s := s + u$
S'3	0	—	ε	u	0	Если $ u > \varepsilon$, то перейти на S'2
	0	П1	0	S'2	0	

Программа 3 короче программы 2 всего на одну команду. Ради этого не стоило бы затевать переделку. Однако повторяющаяся часть программы (блоки $S'2$ и $S'3$) стала короче на две команды, так что на выполнение всей программы 3 затрачивается $7n + 6$ операций (n — число повторений цикла) вместо $9n + 5$ операций, идущих на выполнение программы 2. Уже при $n = 3$ экономия времени выполнения программы составляет около 15%, а такая экономия оправдывает затраченные усилия.

Сокращение повторяющейся части цикла за счет устранения из нее одинаковых вычислений называется *чисткой* цикла. По существу, весь этот параграф посвящен различным приемам чистки циклов. Она и дальше будет предметом нашего внимания, так как чистка циклов — это один из важнейших способов ускорения работы программы.

У п р а ж н е н и я. 1. Составить алгоритм и программу вычисления отрезка степенного ряда для $\arctg x$:

$$s = \sum_{i=0}^n (-1)^i \frac{x^{2i+1}}{2i+1},$$

обрывая ряд на первом члене, для которого $|u_n| \leq \varepsilon$.

2. Для больших значений $|x|$ формула (1) мало пригодна для вычисления e^x , так как члены ряда сначала возрастают по абсолютной величине и лишь потом начинают убывать. Быстрее и точнее значение e^x при $x = n + \xi$, где n — целое, $|\xi| \leq \frac{1}{2}$, можно вычислить по формуле $e^x = e^n \cdot e^\xi$, где e^n вычисляется последовательными умножениями или делениями на e , а e^ξ — с помощью ряда (1). Составить соответствующий алгоритм и машинную программу.

Вычисление значения полинома

$$P(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n \quad (5)$$

выгодно вести, используя схему Горнера, которая описывается алгоритмом:

Н. *Вычисление полинома по схеме Горнера.*

Н1. $i := 0, p := a_0$.

Н2. $i := i + 1, p := px + a_i$.

Н3. Если $i < n$, то перейти на Н2.

Если бы мы применили обычный алгоритм суммирования, то следовало бы начать с последнего слагаемого, и используя описанный в связи с предыдущим примером прием экономии, мы пришли бы к алгоритму:

Г. *Вычисление полинома (5).*

Г1. $i := n, p := a_n, v := 1$.

Г2. $i := i - 1, v := vx, p := p + a_i v$.

Г3. Если $i > 0$, то перейти на Г2.

При использовании схемы Горнера на втором шаге алгоритма экономится одна операция умножения. Достигается это благодаря тому, что при вычислении суммы первых $m + 1$ членов $P(x)$ выносятся за скобки общий множитель x^{n-m} .

Если коэффициенты a_i заданы их общим выражением в зависимости от i , то обычно можно добиться дополнительной экономии, выражая a_i (или некоторые компоненты a_i) через a_{i-1} (через соответствующие компоненты a_{i-1}).

У п р а ж н е н и е 3. Составить алгоритм и программу вычисления значения полинома

$$q_n(x) = \sum_{i=0}^n (-1)^i \frac{i^2 + 1}{(2i)!} x^{n-i}.$$

Но иногда полезен другой прием, слегка обобщающий схему Горнера. В общем случае он заключается в следующем.

Пусть требуется вычислить сумму

$$s = \sum_{i=0}^n u_i.$$

Обозначим

$$s_m = \sum_{i=0}^m u_i,$$

тогда

$$\begin{aligned} s &= s_n, \quad s_0 = u_0, \\ s_m &= s_{m-1} + u_m, \quad m = 1, 2, \dots, n. \end{aligned}$$

Предположим, что из всех слагаемых суммы s_m вынесен общий множитель c_m , так что

$$s_m = c_m t_m.$$

Вместо последовательности $s_0, s_1, \dots, s_n$ будем вычислять последовательность $t_0, t_1, \dots, t_n$. При этом

$$t_m = \frac{s_m}{c_m} = \frac{s_{m-1} + u_m}{c_m} = \frac{c_{m-1} t_{m-1} + u_m}{c_m},$$

или

$$t_m = a_m t_{m-1} + b_m, \quad m = 1, 2, \dots, n, \quad (6)$$

где

$$a_m = \frac{c_{m-1}}{c_m}, \quad b_m = \frac{u_m}{c_m}. \quad (7)$$

Кроме того,

$$t_0 = \frac{s_0}{c_0} = \frac{u_0}{c_0}, \quad (8)$$

$$s = c_n t_n. \quad (9)$$

Расчетные формулы — это (8), (6) и (9), где коэффициенты a_m и b_m определяются выражениями (7). Множители c_m надо подобрать так, чтобы эти выражения, а также выражения для t_0 и c_n получились возможно проще.

Пример 3. Вычислить значение полинома

$$T_n(x) = \sum_{i=0}^{\left[\frac{n}{2}\right]} (-1)^i \frac{2^{n-2i-1}n}{n-i} C_{n-i}^i x^{n-2i} \quad (10)$$

при фиксированных n и x (n — целое, $[\alpha]$ обозначает целую часть α , т. е. такое целое k , что $k \leq \alpha < k+1$, $C_p^q = \frac{p!}{q!(p-q)!}$).

Здесь

$$u_m = (-1)^m \cdot \frac{2^{n-2m-1}n}{n-m} C_{n-m}^m x^{n-2m} = (-1)^m \cdot \frac{2^{n-2m-1}n(n-m)!x^{n-2m}}{(n-m)m!(n-2m)!}$$

или

$$u_m = (-1)^m \frac{2^{n-2m-1}n(n-m-1)!x^{n-2m}}{m!(n-2m)!}.$$

Учитывая, что $u_0 = 2^{n-1}x^n$, попробуем взять c_m в виде

$$c_m = 2^{n-2m-1}x^{n-2m}.$$

Тогда

$$a_m = \frac{2^{n-2m+1}x^{n-2m+2}}{2^{n-2m-1}x^{n-2m}} = 4x^2,$$

$$b_m = (-1)^m \cdot \frac{n(n-m-1)!}{m!(n-2m)!}.$$

Таким образом, a_m фактически от m не зависит, но выражение для b_m оказалось сравнительно сложным. Здесь можно применить испытанный прием, выразив b_m через b_{m-1} :

$$\frac{b_m}{b_{m-1}} = \frac{(-1)^m n(n-m-1)!(m-1)!(n-2m+2)!}{m!(n-2m)!(-1)^{m-1}n(n-m)!}.$$

Обозначим это отношение d_m . После сокращений получим:

$$d_m = -\frac{(n-2m+1)(n-2m+2)}{m(n-m)}. \quad (11)$$

Пусть далее

$$e_m = (n-2m+1)(n-2m+2),$$

$$f_m = -m(n-m),$$

$$g_m = e_m - e_{m-1} = 8m - 4n - 10,$$

$$h_m = f_m - f_{m-1} = 2m - n - 1.$$

Верхний предел суммирования обозначим через $k = \left\lfloor \frac{n}{2} \right\rfloor$. Выразим $T_n(x)$ через t_k . При четном n :

$$n = 2k,$$

$$T_{2k}(x) = c_k t_k = \frac{t_k}{2}.$$

При нечетном n :

$$n = 2k + 1,$$

$$T_{2k+1}(x) = c_k t_k = x t_k.$$

Различить эти два случая проще всего по конечному значению h_m . Заметим, что при $m = 0$ выражение (11) для d_m не имеет смысла. Подготовим поэтому на начальном шаге алгоритма значения переменных e, f, g и h для $m = 1$, а значения b и t для $m = 0$. Тогда на рабочем шаге последовательность вычислений может быть такой: вычислить d (по e и f), b (по d и предыдущему значению b) и t (по b и предыдущему значению t). Затем следует пересчитать e и f (значения g и h , участвующие в этом пересчете, уже подготовлены), а потом g и h для следующего повторения шага. Если принять такую последовательность вычислений, то при последнем повторении шага должны быть выработаны значения b и t , соответствующие $m = k$, и значения e, f, g и h , соответствующие $m = k + 1$. В частности, конечное значение h будет равно:

$$h_{k+1} = 2(k+1) - 2k - 1 = 1 \text{ при четном } n,$$

$$h_{k+1} = 2(k+1) - (2k+1) - 1 = 0 \text{ при нечетном } n.$$

Все предыдущие значения h отрицательны. Таким образом, алгоритм может иметь вид:

Т. *Вычисление полинома (10).*

Т1. $b := 1, t := 1, e := (n-1)n, f := -n+1,$
 $g := -4n-2, h := -n+1, a := 4x^2.$

Т2. $b := be/f, t := at+b, e := e+g, f := f+h,$
 $g := g+8, h := h+2.$

Т3. *Если $h < 0$, то перейти на Т2.*

Т4. *Если $h > 0$, то $T := t/2$, в противном случае $T := xt$.*

Ему соответствует программа:

Программа 4

T1	0	ПЧ	$\langle 1 \rangle$	0	b	$b := 1$
	0	ПЧ	$\langle 1 \rangle$	0	t	$t := 1$
	0	—	n	$\langle 1 \rangle$	e	$n - 1$
	0	—	0	e	f	$f := -n + 1$
	0	$\times$	e	n	e	$e := (n - 1)n$
	0	$+АП$	102	f	g	$-4n + 4$
	0	—	g	$\langle 6 \rangle$	g	$g := -4n - 2$
	0	ПЧ	f	0	h	$h := -n + 1$
	0	$+$	x	x	a	$2x$
	0	$\times$	a	a	a	$a := 4x^3$
T2	0	$\times$	b	e	b	be
	0	$/$	b	f	b	$b := be/f$
	0	$\times$	a	t	t	at
	0	$+$	t	b	t	$t := at + b$
	0	$+$	e	g	e	$e := e + g$
	0	$+$	f	h	f	$f := f + h$
	0	$+$	g	$\langle 8 \rangle$	g	$g := g + 8$
	0	$+$	h	$\langle 2 \rangle$	h	} $h := h + 2$, если $h < 0$, то перейти на T2
T3	0	П1	0	T2	0	
T4	0	—	0	h	0	} $T := 1/2$, если $h < 0$, то перейти на T5
	0	П1	$\langle 1/2 \rangle$	T5	T	
	0	ПЧ	x	0	T	$T := x$
T5	0	$\times$	T	t	T	$T := Tt$

Последний шаг можно записать еще и так:

$$T4'. T := \left(x - \left(x - \frac{1}{2} \right) h \right) t,$$

но машинная программа не станет при этом короче.

§ 6. Программные массивы

Формально, *программным массивом* называется группа последовательных ячеек памяти вместе с их содержимым. К этому следует добавить неформальное требование, чтобы имелось основание рассматривать весь массив как единое целое. Очень часто в программировании приходится иметь дело с массивами чисел. Например, при решении системы линейных алгебраических уравнений

$$a_{11}x_1 + \dots + a_{1n}x_n = b_1,$$

$$\dots \dots \dots$$

$$a_{n1}x_1 + \dots + a_{nn}x_n = b_n$$

естественно ввести в рассмотрение массив коэффициентов a_{ij} , массив свободных членов b_i , массив неизвестных x_j . Наряду с массивами чисел встречаются массивы данных иной природы, например,

может возникнуть потребность рассматривать некоторый участок программы как массив команд.

Понятие массива тесно связано с понятием структурного цикла. Обычно при выполнении структурного цикла последовательно, в определенном порядке, вовлекаются в обработку (используются или вычисляются) элементы одного или нескольких массивов. На первых порах мы будем иметь дело лишь с такими циклами, в которых адреса элементов массива, участвующих в последовательных повторениях цикла, образуют арифметическую прогрессию. В простейшем случае массив состоит из n элементов, расположенных в ячейках с адресами $a + 1, a + 2, \dots, a + n$, и все эти элементы вовлекаются в вычисления именно в такой последовательности, так что выполнение цикла состоит из n повторений некоторой группы команд и при i -м повторении в вычислениях участвует элемент из ячейки с адресом $a + i$.

Во всех последующих программах будут встречаться команды, содержащие адреса элементов массива, заданные выражениями вида $a + i$ или $c - ki + m$ или еще более сложными. В таких выражениях обязательно присутствует слагаемое, изображающее так называемый *базовый адрес* — адрес, связанный с размещением в памяти некоторого массива. В упомянутом выше простейшем случае таким адресом является адрес, представленный буквой a . Связь базового адреса с адресами элементов массива всегда особо оговаривается. В рассматриваемом случае буква a заменяет адрес ячейки, непосредственно предшествующей ячейкам, образующим массив, так что сама ячейка с адресом a в массив не входит. В других случаях ячейка с базовым адресом может входить в состав массива или, напротив, располагаться в отдалении от массива. Например, если массив состоит из ячеек с адресами

$$c - rn + m, \quad c - rn + m + 1, \quad \dots, \quad c - 1 + m,$$

где c — базовый адрес, а r, n и m — некоторые целые числа ($r > 0, n > 0$), то ячейка с адресом c принадлежит массиву при $1 \leq m \leq rn$ и не принадлежит, если $m < 1$ или $m > rn$.

Кроме базового адреса, в выражения рассматриваемого типа могут входить другие буквы (а также числовые константы) и тогда они обозначают постоянные или переменные числовые значения. В выражении $a + i$ такой буквой является буква i , текущее значение которой в нашем примере совпадает с номером повторения цикла. В выражении $c - ki + m$ буквы k и i могут обозначать переменные, а m — постоянное (в пределах некоторого цикла) числовое значение.

Существует аналогия между такими буквами и буквами, которые в традиционной символике употребляются в роли индексов, например, в обозначениях $a_i, c_{i,m}$. Но если обозначение a_i ничего не гово-

рит о размещении массива с элементами $a_1, \dots, a_n$ в памяти машины, то выражение $a + i$ прямо указывает на связь адреса интересующего нас элемента массива с базовым адресом a и текущим значением переменной i . Обозначения с индексами будут часто использоваться в пояснениях к программе.

С о г л а ш е н и я о с и м в о л и ч е с к и х а д р е с а х (продолжение; см. стр. 78 и 81).

8. Если в алгоритме, по которому составляется программа, и в пояснениях к программе используется буква с переменными индексами (например, a_i), то в качестве базового адреса некоторого массива будет использоваться при записи программы та же буква. При этом для этого массива должна быть оговорена связь между адресом его элемента и соответствующим обозначением в алгоритме (типичный пример: адрес a_i равен $a + i$).

9. Если в команде некоторый адрес заменен выражением, то кроме базового адреса все остальные буквы в этом выражении заменяют не адреса, а числовые значения. Например, в выражении $a + i$ буква i заменяет не адрес ячейки, отведенной для хранения текущего значения переменной i , а само это значение. Таким образом, соглашение 4 (см. гл. 6) распространяется только на буквы, заменяющие целый адрес, и не относится к буквам, входящим в состав выражений.

§ 7. Программирование простых структурных циклов

Разбор методов программирования структурных циклов начнем сразу с примера.

П р и м е р 1. Вычисление суммы $\sum_{i=1}^n x_i$.

Нам уже приходилось писать программы вычисления разных сумм. Но во всех этих программах слагаемые вычислялись по ходу суммирования одно за другим. Теперь рассмотрим случай, когда слагаемые уже вычислены заранее и находятся в последовательных ячейках памяти с адресами $x + 1, x + 2, \dots, x + n$, так что адрес ячейки, в которой размещено слагаемое x_i , равен $x + i$. На алгоритме суммирования это никак не отразится, он имеет привычный нам вид:

S. Вычисление суммы $s = \sum_{i=1}^n x_i$.

S1. $s := 0, i := 0$.

S2. $i := i + 1, s := s + x_i$.

S3. Если $i < n$, то перейти на S2.

Но приемы программирования оказываются существенно иными. Для реализации присваивания $s := s + x_i$ нужно выполнить команду

$$0 \quad + \quad s \quad x + i \quad s \quad s := s + x_i$$

Фактически это не одна команда, а n разных команд, соответствующих разным значениям i от 1 до n . Таким образом, в этой программе придется вычислять не только новые значения переменных, но и новые значения команды. Мы уже знаем, что для этого в машине М-20 существует специальная операция «сложение адресных частей команд» или, короче, «сложение мантисс». Пусть команда требуемого вида помещается в ячейке p , а в ячейке z размещена константа переадресации

$$0 \quad 00 \quad 0000 \quad 0001 \quad 0000,$$

что мы коротко будем записывать в виде

$$z \quad 0 \quad 0 \quad 0 \quad 1 \quad 0$$

Тогда для перехода от одного значения команды к следующему, с увеличенным на 1 значением адреса $x + i$, надо выполнить команду

$$0 \quad +M \quad p \quad z \quad p$$

Какое написать к ней примечание? Можно указать вид полученного результата, но он у нас уже написан в ячейке p . По существу эта команда нужна для реализации присваивания $i := i + 1$, потому что переменной i в программе соответствует не только числовая переменная i , но и команда-переменная, или, как чаще говорят, *переменная команда* p . Так что присваивание $i := i + 1$ реализуется двумя командами

$$\left. \begin{array}{l} 0 \quad + \quad i \quad \langle 1 \rangle \quad i \\ 0 \quad +M \quad p \quad z \quad p \end{array} \right\} i := i + 1$$

Но тогда ясно, что и присваивание $i := 0$ тоже должно быть запрограммировано в виде двух команд:

$$\left. \begin{array}{l} 0 \quad ПЧ \quad 0 \quad 0 \quad i \\ 0 \quad ПЧ \quad z_1 \quad 0 \quad p \end{array} \right\} i := 0$$

где z_1 — адрес ячейки, содержащий значение команды p , соответствующее значению $i = 0$:

$$z_1 \quad 0 \quad + \quad s \quad x \quad s$$

Как запрограммировать остальные операции алгоритма, мы хорошо знаем. В итоге получаем программу

Программа 1

S1	0	ПЧ	0	0	s	} i := 0	s := 0	
	0	ПЧ	0	0	i			
	0	ПЧ	z ₁	0	p			
S2	0	+	i	<1>	i	} i := i + 1		
	0	+M	p	z	p			
p	0	+	s	x + i	s		s := s + x _i	
S3	0	—	i	n	0	Если i < n, то (ω := 1)		
	0	П1	0	S2	0	перейти на S2		

В том, что эта программа работает правильно, нас должны были убедить все предшествующие рассуждения. Тот, кто не вполне в этом убежден, может проследить, какие значения команды p вырабатывает команда $S2 + 1$ при первом и последующих повторениях цикла, как эти значения связаны со значениями переменной i и каково состояние команды p при ее последнем выполнении перед выходом из цикла.

Но от хорошей программы требуется, чтобы она была не только правильной, но и экономной — короткой и быстро работающей. Чтобы сократить программу, сначала переделаем алгоритм S , заменив шаг $S3$ шагом

$S3'$. Если $i \neq n$, то перейти на $S2$.

Поскольку i и n — числа целые и представляются в машине точно, соответствующие команды можно написать в виде:

S3'	0	$\neq$	i	n	0	Если $i \neq n$, то ($\omega := 0$) перейти на S2
	0	П0	0	S2	0	

Но ведь переменной i в алгоритме S соответствует в программе 1 не только число в ячейке i , но и команда в ячейке p . При $i = n$ эта команда принимает вид

$$0 + s \quad x + n \quad s \quad (1)$$

(кстати, именно в таком виде она и должна быть выполнена последний раз). Поэтому команду в ячейке $S3'$ можно заменить командой

S3'	0	$\neq$	p	z ₂	0	Если $i \neq n$, то $\omega := 0$

где z_2 — адрес ячейки, в которую помещен код команды (1). Если произвести такую замену, то числовая переменная i уже не нужна — она нигде в измененной программе не используется. Значит, ячейку i и все связанные с ней операции можно исключить из программы, которая окончательно примет вид:

Программа 2

S1	0	ПЧ	0	0	s	$s := 0$
	0	ПЧ	z_1	0	p	$i := 0$
S2	0	+M	p	z	p	$i := i + 1$
p	0	+	s	$x + i$	s	$s := s + x_i$
S3	0	$\neq$	p	z_2	0	Если $i \neq n$, то ($\omega := 0$)
	0	ПЧ	0	S2	0	перейти на S2

Эта программа использует три константы

z	0	0	0	1	0
z_1	0	+	s	x	s
z_2	0	+	s	$x + n$	s

первую — для переадресации переменной команды p , вторую для ее формирования в исходном состоянии, третью — для сравнения на окончание цикла. Соответственно они так и называются *константой переадресации*, *константой формирования* и *константой сравнения*.

Пример 2. Вычисление суммы c двух векторов a и b n -мерного линейного пространства:

$$c_i = a_i + b_i, \quad i = 1, 2, \dots, n.$$

Пример очень похож на предыдущий и не нуждается в подробных пояснениях. Алгоритм вычисления имеет вид

A. *Вычисление суммы двух векторов.*

A1. $i := 0$.

A2. $i := i + 1$, $c_i := a_i + b_i$.

A3. Если $i \neq n$, то перейти на шаг A2.

При составлении программы будем считать, что для компонент каждого из векторов отведена группа из n рядом стоящих ячеек: $a + 1, \dots, a + n$ — для компонент вектора a и т. д. Тогда в программе должна быть одна переменная команда вида

p 0 + a + i b + i c + i $c_i := a_i + b_i$

Для формирования этой команды при $i = 0$ (шаг A1) надо заслать в ячейку p *заготовку* (так мы будем часто называть константы формирования и сравнения, очень похожие на настоящие команды, но отличающиеся от обычных команд тем, что они никогда не выполняются):

0 + a b c

Для перехода к очередному значению i (действие $i := i + 1$ на шаге A2) надо выполнить операцию

0 +M p z_1 p

где z_1 — константа переадресации

0 0 1 1 1

Наконец, для реализации шага АЗ надо сравнить текущее значение команды p с заготовкой

0 + $a+n$ $b+n$ $c+n$

Итак, программа может иметь вид

Программа 3

A1	0	ПЧ	z	0	p	$i := 0$
A2	0	+M	p	z_1	p	$i := i + 1$
p	0	+	$a+i$	$b+i$	$c+i$	$c_i = a_i + b_i$
A3	0	$\neq$	p	z_2	0	Если $i \neq n$, то
	0	П0	0	A2	0	перейти на A2

со следующими значениями констант

z	0	+	a	b	c
z_1	0	0	1	1	1
z_2	0	+	$a+n$	$b+n$	$c+n$

Пример 3. Программа вычисления значения полинома

$$y = a_0 + a_1x + \dots + a_nx^n$$

с заданными (заранее вычисленными) коэффициентами.

Алгоритм, по-прежнему основанный на схеме Горнера, может иметь вид:

Р. Вычисление полинома с заданными коэффициентами.

Р1. $y := a_n$, $i := n$.

Р2. $i := i - 1$, $y := xy + a_i$.

Р3. Если $i \neq 0$, то перейти на Р2.

Предположим, что коэффициенты полинома хранятся в ячейках a , $a + 1$, ..., $a + n$ и адрес коэффициента a_i равен $a + i$. Для реализации присваивания $y := xy + a_i$ нужны две команды

	0	$\times$	x	y	y	xy
p	0	+	y	$a+i$	y	$y := xy + a_i$

Лишь вторая из этих команд — переменная, только она должна формироваться и переадресовываться в соответствии с значением i . В отличие от предыдущих примеров значение i здесь не возрастает, а убывает при каждом повторении цикла. На формировании переменной команды это не отражается. Для реализации присваивания $i := n$ надо в ячейку p заслать заготовку

0 + y $a+n$ y

Чтобы реализовать переадресацию $i := i - 1$, можно прибегнуть к операции «вычитание мантисс», т. е. выполнить команду

$0 \quad -M \quad p \quad z_1 \quad p$

где z_1 — адрес константы

$0 \quad 0 \quad 0 \quad 1 \quad 0$

Но можно было бы воспользоваться и операцией сложения мантисс

$0 \quad +M \quad p \quad z_1 \quad p$

поместив при этом в ячейку z_1 константу

$0 \quad 0 \quad 0 \quad -1 \quad 0. \quad (2)$

Разумеется, адреса в константах, как и в обычных командах, должны быть неотрицательны, так что адресная часть этой константы в действительности должна быть записана в дополнительном коде — полная восьмеричная запись константы (2) имеет вид

$0 \quad 00 \quad 7777 \quad 7777 \quad 0000$

В остальном реализация алгоритма Р не представляет ничего нового. Окончательная программа имеет вид:

Программа 4

P1	0	ПЧ	$a+n$	0	y	$y := a_n$
	0	ПЧ	z	0	p	$i := n$
P2	0	$-M$	p	z_1	p	$i := i - 1$
	0	$\times$	x	y	y	xy
p	0	$+$	y	$a+i$	y	$y := xy + a_i$
P3	0	$\neq$	p	z_2	0	Если $i \neq 0$, то ($\omega := 0$)
	0	ПО	0	P2	0	перейти на P2

Константы:

z	0	$+$	y	$a+n$	y
z_1	0	0	0	1	0
z_2	0	$+$	y	a	y

Вспомним теперь, что программист должен не только составлять программы в символических обозначениях, как мы привыкли это делать, но и готовить их к перфорации, т. е. переписывать в восьмеричном коде. Только что мы столкнулись с небольшой трудностью — восьмеричной записью константы (2). Не возникнут ли другие трудности при попытке переписать в восьмеричном коде, например, программу 4?

Прежде всего нужно распределить память. Предположим, для примера, что адреса переменных и констант выбраны следующим образом:

x	7000
y	7001
z	6000
z_1	6001
z_2	6002

Пусть далее программа размещается, начиная с ячейки $P1 = 0420$, коэффициенты полинома a_i размещены в ячейках от $a = 3100$ и далее, и, наконец, пусть степень полинома n равна $10_{10} = 12_8$. После этого никакой сложности не представляет переписать все команды и константы программы в восьмеричном виде, кроме переменной команды p (очевидно, $p = 0424$). Но с ней дело обстоит особенно просто именно потому, что эта команда — переменная. Что бы мы ни написали на ее месте, другими словами, что бы мы ни поместили в ячейку p перед началом работы цикла, команда $P1 + 1$ зашлет в эту ячейку заготовку z , сформировав нужное начальное значение переменной команды, и дальше она (эта команда) будет изменяться в соответствии с нашими потребностями. Поэтому при переписывании программы из символических обозначений в восьмеричные можно ячейку p заполнить хотя бы нулями.

Программа примет вид:

0420	0 00	3112	0000	7001
0421	0 00	6000	0000	0424
0422	0 33	0424	6001	0424
0423	0 05	7000	7001	7001
0424	0 00	0000	0000	0000
0425	0 15	0424	6002	0000
0426	0 76	0000	0422	0000

при константах:

6000	0 01	7001	3112	7001
6001	0 00	0000	0001	0000
6002	0 01	7001	3100	7001

У п р а ж н е н и е 1. Чтобы оценить удобство символических обозначений и роль покомандных примечаний к программам, попробуйте разобраться в том, что делает программа

2366	0 00	5012	0000	2371
2367	0 00	0000	0000	0001
2370	0 13	2371	5011	2371
2371	0 05	0000	0000	1344
2372	0 01	0001	1344	0001
2373	0 15	2371	5013	0000
2374	0 76	0000	2370	0000

при константах

5011	0 00	0001	0001	0000
5012	0 05	4000	4040	1344
5013	0 05	4040	4100	1344

Пример 4. В ячейках $x + 1, \dots, x + n$ находятся n чисел $x_1, \dots, x_n$. Требуется в ячейки $y + 1, \dots, y + n$ заслать значения выражений $y_i = 1 - 1/x_i, i = 1, 2, \dots, n$.

Алгоритм для этого примера столь же прост, как все предыдущие:

В. Вычисление $y_i = 1 - 1/x_i$ для $i = 1, 2, \dots, n$.

B1. $i := 0$.

B2. $i := i + 1, y_i := 1 - 1/x_i$.

B3. Если $i \neq n$, то перейти на B2.

Как всегда, программирование следует начать с команд, реализующих присваивание $y_i := 1 - 1/x_i$. Очевидно, они могут иметь вид

p	0	/	$\langle 1 \rangle$	$x+i$	r	$1/x_i$
	0	-	$\langle 1 \rangle$	r	$y+i$	$y_i := 1 - 1/x_i$

где r — адрес рабочей ячейки. Этот пример отличается от предыдущих тем, что в нем переменны обе команды. Поэтому каждое из присваиваний $i := 0$ и $i := i + 1$ должно реализоваться двумя командами. Для сравнения достаточно, конечно, использовать одну (любую) из переменных команд. Вся программа может быть такой:

Программа 5

B1	0	ПЧ	z	0	p	} $i := 0$
	0	ПЧ	z_1	0	$p+1$	
B2	0	+M	p	z_2	p	} $i := i + 1$
	0	+M	$p+1$	z_3	$p+1$	
p	0	/	$\langle 1 \rangle$	$x+i$	r	$1/x_i$
	0	-	$\langle 1 \rangle$	r	$y+i$	$y_i := 1 - 1/x_i$
B3	0	$\neq$	p	z_4	0	Если $i \neq n$, то перейти на B2
	0	ПО	0	B2	0	

со следующими константами:

z	0	/	$\langle 1 \rangle$	x	r
z_1	0	-	$\langle 1 \rangle$	r	y
z_2	0	0	0	1	0
z_3	0	0	0	0	1
z_4	0	/	$\langle 1 \rangle$	$x+n$	r

Одну из констант можно сэкономить, если частное $1/x_i$ вычислять в ячейке $y + i$. Хотя число переменных адресов при этом увеличивается, но число переменных команд не возрастает и они становятся более похожими одна на другую. Программа становится такой:

Программа 6

B1	0	ПЧ	z	0	p	} $i := 0$
	0	ПЧ	z_1	0	$p+1$	
B2	0	+M	p	z_2	p	} $i := i+1$
	0	+M	$p+1$	z_2	$p+1$	
p	0	/	$\langle 1 \rangle$	$x+i$	$y+i$	$1/x_i$
	0	—	$\langle 1 \rangle$	$y+i$	$y+i$	$y_i := 1 - 1/x_i$
B3	0	$\neq$	p	z_3	0	Если $i \neq n$, то
	0	П0	0	B2	0	перейти на B2

где константы имеют вид:

z	0	/	$\langle 1 \rangle$	x	y
z_1	0	—	$\langle 1 \rangle$	y	y
z_2	0	0	0	1	1
z_3	0	/	$\langle 1 \rangle$	$x+n$	$y+n$

Представим себе, что зависимость y_i от x_i была бы более сложной и в соответствующей программе адрес переменной x_i (т. е. $x+i$) пришлось бы употребить много раз. Тогда потребовалось бы написать очень много команд формирования и переадресации переменных команд и использовать много заготовок. Оказывается, без этого можно обойтись, если алгоритм такой задачи написать в виде

C. Вычисление $y_i = f(x_i)$ для $i = 1, 2, \dots, n$.

C1. $i := 0$.

C2. $i := i+1$, $u := x_i$, $v := f(u)$, $y_i := v$.

C3. Если $i \neq n$, то перейти на C2.

Здесь понадобятся лишь две переменные команды, реализующие присваивания $u := x_i$ и $y_i := v$, и соответственно по две команды и константы формирования и переадресации, т. е. столько, сколько их было в программе 5.

У п р а ж н е н и е 2. Написать программу, реализующую алгоритм C, предполагая, что для вычисления $f(u)$ составлена отдельная подпрограмма, обратиться к которой можно с помощью команды ПВ (см. пример 7.4.1).

Подведем некоторые итоги. Переменные, с которыми приходится иметь дело в простом структурном цикле, распадаются на две совокупности X и Y. В совокупность X входят переменные, которые ведут себя подобно переменным, фигурирующим в итерационных циклах, — они последовательно принимают различные значения, причем при каждом повторении цикла каждой такой переменной присваивается, как правило, новое значение. Среди этих переменных можно выделить одну, последовательные значения которой образуют целочисленную арифметическую прогрессию. Эта переменная называется *параметром цикла*. Чаще всего последовательность значений, принимаемых параметром цикла, — это прогрессия

1, 2, ..., n , где n — число повторений цикла. Если же это не так, то линейной заменой параметра

$$k = a + (i - 1)q$$

можно свести случай параметра k , пробегающего прогрессию $a, a + q, \dots, a + (n - 1)q$, к параметру i , принимающему последовательно значения 1, 2, ..., n . Поэтому, не теряя в общности, в дальнейшем будем рассматривать лишь последний случай.

Переменные совокупности Y образуют одну или несколько групп (массивов), в каждую из которых входит n переменных (иногда $n + c$, где c — число, не зависящее от n). Пусть $y_1, \dots, y_n$ — одна из таких групп. Тогда для хранения значений этих переменных должны быть выделены n ячеек, адреса которых изменяются закономерно в зависимости от индекса j переменной y_j . Во всех рассмотренных примерах эта зависимость — линейная: адрес y_j равен $c_y + d_y j$, где c_y и d_y — целочисленные константы, зависящие от рассматриваемой группы переменных $y_1, y_2, \dots, y_n$, но не зависящие от j .

При каждом повторении цикла над переменными, входящими в совокупности X и Y , выполняются некоторые вычисления, в частности, переход к очередному значению параметра, описываемый присваиванием $i := i + 1$. Совокупность всех этих вычислений будем обозначать $P(i)$. Для каждой переменной x из совокупности X при вычислениях $P(i)$ используется прежнее (оставшееся от вычислений $P(i - 1)$) значение этой переменной и вырабатывается новое значение, которое, после того как оно получено, может участвовать при продолжении вычислений $P(i)$. Что касается группы переменных $y_1, \dots, y_n$ из совокупности Y , то в вычислениях $P(i)$ участвует, как правило, переменная y_i и, возможно, еще несколько переменных $y_{i+\alpha}$ с близкими к i значениями индекса $i + \alpha$ (α не зависит от i).

В этих условиях вычисления $P(i)$ описываются командами вида

$$0 \text{ КОП } a_1 + b_1 i \quad a_2 + b_2 i \quad a_3 + b_3 i, \quad (3)$$

адреса которых линейно зависят от текущего значения параметра цикла i (разумеется, при $b_s = 0$ адрес A_s фактически не зависит от i , такой адрес соответствует константе или переменной из X).

Если все эти условия соблюдаются, то схема программы (алгоритм) простого структурного цикла имеет следующий вид.

- L. Алгоритм выполнения простого структурного цикла с параметром i , пробегающим значения 1, 2, ..., n .
- L1. а) Присвоить начальные значения переменным из совокупности X (в частности, $i := 0$).
- б) Сформировать переменные команды для значения параметра $i = 0$.

L2. а) *Переадресовать переменные команды для перехода к очередному значению i , на 1 большему предыдущего.*

б) *Выполнить вычисления $P(i)$ (в том числе присваивание $i := i + 1$).*

L3. *Если $i \neq n$, то перейти на L2.*

Шаг L1 образует *подготовительную часть* цикла, его подшаг L1б принято называть *формированием* переменных команд. Подшаг L2а называется *переадресацией*, а подшаг L2б (иначе говоря, совокупность вычислений $P(i)$) — *рабочей частью* цикла. Наконец, шаг L3 можно назвать *проверкой на окончание работы цикла или на выход* из цикла.

Составление программы структурного цикла должно начинаться с программирования его рабочей части $P(i)$, так как написать команды формирования и переадресации переменных команд можно только после того, как число и вид этих команд станут известны. В ходе программирования шагов L1б, L2а и L3 выявляется число и вид констант (заготовок), необходимых для формирования, переадресации и проверки на выход из цикла. Эти шаги могут быть запрограммированы по следующим формальным правилам.

Переадресация. Для каждой переменной команды (3) подшаг L2а должен содержать команду переадресации вида

$$0 +M \ p \ z_n \ p,$$

где p — адрес ячейки, в которой размещается переменная команда (3), а z_n — адрес ячейки, где хранится константа переадресации

$$0 \ 0 \ b_1 \ b_2 \ b_3.$$

Адресная часть этой константы записывается в дополнительном коде по модулю 2^{36} , т. е. она равна $B = b_1 \cdot 2^{24} + b_2 \cdot 2^{12} + b_3$, если $B > 0$, и $2^{36} + B$, если $B < 0$.

Формирование для $i = 0$. Для каждой переменной команды (3) подшаг L1б должен содержать команду формирования вида

$$0 \ ПЧ \ z_\Phi \ 0 \ p,$$

где p — по-прежнему адрес переменной команды (3), а z_Φ — адрес ячейки, в которой помещается константа формирования

$$0 \ КОП \ a_1 \ a_2 \ a_3.$$

Проверка на выход при $i = n$. Выбирается одна из переменных команд (3) и блок L3 реализуется в виде двух команд

$$\begin{aligned} 0 &\neq p \ z_c \ 0 \\ 0 &П0 \ 0 \ L2 \ 0, \end{aligned}$$

где p — адрес выбранной переменной команды, z_c — адрес ячейки, в которой помещена константа сравнения

$$0 \ КОП \ a_1 + b_1 n \ a_2 + b_2 n \ a_3 + b_3 n,$$

L2 — адрес первой команды участка программы, реализующего шаг L2.

Алгоритм L может быть изменен, если мы хотим, чтобы его этапы (шаги и подшаги) выполнялись в другом порядке. Разумеется, при изменении порядка этапов должно изменяться и их содержание. Например, если поменять местами рабочую часть и переадресацию, то алгоритм должен принять вид:

L'. Алгоритм простого структурного цикла.

L'1. Присвоить начальные значения переменным из X, сформировать переменные команды для $i = 1$.

L'2. P (i), переадресовать переменные команды для $i := i + 1$.

L'3. Если $i \neq n + 1$, то перейти на L'2.

У п р а ж н е н и е 3. Каков должен быть вид констант формирования, переадресации и сравнения для переменной команды (3) в алгоритме L'?

Проверка на выход из цикла может предшествовать рабочей части и (или) переадресации. Более того, если цикл составляется в расчете на то, что значение n может быть равно нулю, то проверка на выход перед первым выполнением рабочей части обязательна. Алгоритм при этом может иметь вид:

L''. Алгоритм простого структурного цикла.

L''1. Сформировать переменные команды для $i = 0$, присвоить начальные значения переменным из X.

L''2. Если $i = n$, то выход.

L''3. Переадресовать переменные команды для $i := i + 1$, P (i), перейти на L''2.

Осуществленная здесь перестановка подшагов подготовительной части не требует изменений в других частях алгоритма.

У п р а ж н е н и я 4. Написать вид констант формирования, переадресации и сравнения для алгоритма L''.

5. Составить алгоритм простого структурного цикла при следующем порядке этапов: подготовительная часть, рабочая часть, проверка на выход, переадресация.

6. Переделать программу 2 так, чтобы порядок этапов был следующим: подготовительная часть, проверка на выход, рабочая часть, переадресация.

П р и м е р 5. Линейная интерполяция по таблице.

Пусть для функции $y = f(x)$ составлена таблица ее значений для значений аргумента $x_0, x_1, \dots, x_n$, расположенных в порядке возрастания: $x_0 < x_1 < x_2 < \dots < x_n$. Обозначим: $y_i = f(x_i)$. Пусть табличные значения функции и аргумента расположены в последовательных ячейках памяти, начиная с ячейки с адресом $a + 1$ в порядке: $y_0, x_0, y_1, x_1, \dots$, так что адрес y_i равен $a + 2i + 1$, адрес x_i равен $a + 2i + 2$. Для заданного значения x , $x_0 \leq x \leq x_n$, требуется выбрать содержащий его отрезок $[x_{i-1}, x_i]$ и найти соответ-

ствующее значение y , пользуясь линейной интерполяцией в этом отрезке

$$y = y_{i-1} + \frac{y_i - y_{i-1}}{x_i - x_{i-1}} (x - x_{i-1}). \quad (4)$$

Алгоритм решения этой задачи может быть таким.

1. *Линейная интерполяция по таблице.*
11. $i := 0$.
12. $i := i + 1$.
13. *Если $x > x_i$, то перейти на 12.*
14. $y := y_{i-1} + (y_i - y_{i-1})(x - x_{i-1}) / (x_i - x_{i-1})$.

По сравнению с ранее рассмотренными примерами здесь есть две особенности. В цикле как бы отсутствует рабочая часть, ибо основные вычисления с переменными, индексы которых зависят от параметра цикла i , выполняются на шаге 14, не входящем в цикл. Однако следует считать, что эти вычисления входят в совокупность действий $P(i)$, потому что адреса переменных, участвующих в этих вычислениях, зависят от i . Это становится особенно очевидным, если заметить шаги 13 и 14 одним шагом:

- 13'. *Если $x \leq x_i$, то*
 $y := y_{i-1} + (y_i - y_{i-1})(x - x_{i-1}) / (x_i - x_{i-1})$,
в противном случае перейти на 12.

Вторая особенность этого примера состоит в том, что число повторений цикла заранее не определено и проверка на окончание цикла делается не по значению параметра цикла i , а по значению величины x_i , индекс которой зависит от i . Таким образом, для реализации этой проверки также понадобится переменная команда (команда, вырабатывающая признак ω). Заметим, кстати, что эта команда вычисляет разность $x_i - x$, поэтому вместо формулы (4) целесообразно воспользоваться другим вариантом интерполяционной формулы, в котором вместо разности $x - x_{i-1}$ используется разность $x_i - x$:

$$y = y_i + \frac{y_{i-1} - y_i}{x_i - x_{i-1}} (x_i - x).$$

С учетом этих замечаний алгоритм примет вид:

- 1'. *Линейная интерполяция по таблице.*
- 1'1. $i := 0$.
- 1'2. $i := i + 1$, $r := x_i - x$.
- 1'3. *Если $r < 0$, то перейти на 1'2.*
- 1'4. $y := y_i + (y_{i-1} - y_i) r / (x_i - x_{i-1})$.

Программирование этого алгоритма начнем со второй части шага 1'2, шагов 1'3 и 1'4. Среди команд, реализующих эти действия, содержатся все переменные команды цикла.

126	0	—	$a+2i+2$	x	r	$r := x_i - x$, если $r < 0$,
13	0	П1	0	12	0	то перейти на 12
14	0	—	$a+2i-1$	$a+2i+1$	y	$y_{i-1} - y_i$
	0	×	y	r	y	$(y_{i-1} - y_i) r$
	0	—	$a+2i+2$	$a+2i$	r	$x_i - x_{i-1}$
	0	/	y	r	y	$(y_{i-1} - y_i) r / (x_i - x_{i-1})$
	0	+	$a+2i+1$	y	y	$y := y_i + \dots$

Всего набралось 4 переменные команды. Зная их общий вид, завершить составление программы уже не трудно. Готовая программа примет вид:

Программа 7

11	0	ПЧ	z_1	0	14-2	} $i := 0$
	0	ПЧ	z_2	0	14	
	0	ПЧ	z_3	0	14+2	
	0	ПЧ	z_4	0	14+4	
12	0	+M	14-2	z_5	14-2	} $i := i + 1$
	0	+M	14	z_6	14	
	0	+M	14+2	z_6	14+2	
	0	+M	14+4	z_5	14+4	
	0	—	$a+2i+2$	x	r	} $r := x_i - x$, если $r < 0$, то перейти на 12
13	0	П1	0	12	0	
14	0	—	$a+2i-1$	$a+2i+1$	y	$y_{i-1} - y_i$
	0	×	y	r	y	$(y_{i-1} - y_i) r$
	0	—	$a+2i+2$	$a+2i$	r	$x_i - x_{i-1}$
	0	/	y	r	y	$(y_{i-1} - y_i) r / (x_i - x_{i-1})$
	0	+	$a+2i+1$	y	y	$y := y_i +$ $+(y_{i-1} - y_i) \cdot r / (x_i - x_{i-1})$

Эта программа использует константы:

z_1	0	—	$a+2$	x	r
z_2	0	—	$a-1$	$a+1$	y
z_3	0	—	$a+2$	a	r
z_4	0	+	$a+1$	y	y
z_5	0	0	2	0	0
z_6	0	0	2	2	0

Казалось бы, в этой программе уже ничего сэкономить нельзя. Следующее упражнение показывает, что это не так. Сходство команд 14 — 2 и 14 + 4, а также команд 14 и 14 + 2 между собой позволяет использовать особые приемы формирования и переадресации (правда, для увеличения этого сходства приходится слегка изменить схему вычислений и помещать промежуточные результаты в другие рабочие ячейки). В результате программа сокращается на две команды и экономится одна константа. Если же при распределе-

нии памяти сделать так, чтобы адрес x был на 1 больше адреса y , то и константа z_4 будет не нужна (она совпадет с z_5).

У п р а ж н е н и е 7. Показать, что следующая программа реализует тот же алгоритм, что программа 7.

Программа 8

I1	0	ПЧ	z_1	0	$14+4$	} Формирование для $i := 0$
	0	ПЧ	z_2	0	$14+2$	
I2	0	+M	$14+4$	z_3	$14+4$	Переадресация для $i := i+1$
	0	+M	$14+4$	z_4	$14-2$	Формирование $14-2$ для текущего значения i
	0	+M	$14+2$	z_5	14	Формирование 14 по предыдущему значению $i+2$
	0	+M	14	z_5	$14+2$	Формирование $14+2$ по текущему значению 14
	0	—	$a+2i+2$	x	y	$y := x_i - x$, если $y < 0$, то перейти на I2
	0	П1	0	I2	0	
I4	0	—	$a+2i+1$	$a+2i-1$	r	$y_i - y_{i-1}$
	0	×	r	y	y	$(y_i - y_{i-1}) y$
	0	—	$a+2i+2$	$a+2i$	r	$x_i - x_{i-1}$
	0	/	y	r	y	$(y_i - y_{i-1}) y / (x_i - x_{i-1})$
	0	—	$a+2i+1$	y	y	$y := y_i -$ $(y_i - y_{i-1}) y / (x_i - x_{i-1})$

Константы:

z_1	0	—	$a+1$	y	y
z_2	0	—	$a+2$	a	r
z_3	0	0	2	0	0
z_4	0	0	1	$x-y$	0
z_5	0	0	1	1	0

Глава 8. ЭЛЕМЕНТЫ ЯЗЫКА АЛГОЛ 60

В этой главе будет кратко описан международный алгоритмический язык АЛГОЛ 60. Этот язык определен документом [2], мало-пригодным для первого ознакомления с языком. Пособиями для изучения АЛГОЛа могут служить работы [5, 6, 7, 8, 18, 23].

§ 1. Значения и типы

Все операции в АЛГОЛе выполняются над *числовыми* или *логическими* значениями и вырабатывают результаты, относящиеся к одному из этих классов значений. Числовые значения, в свою очередь, делятся на *целые* и *вещественные* (приближенные). Целые числа в АЛГОЛе понимаются так же, как в математике. В частности, результат операции над целыми числами, если он сам является целым числом, определен точно и однозначно. Вещественные же числа в АЛГОЛе, напротив, имеют очень мало общего с соответствующим строгим математическим понятием и рассматриваются с позиций приближенных вычислений. Например, $1/3$, $\sqrt{2}$ и π не считаются в АЛГОЛе вещественными числами и равенство

$$(1/2) \times 2 = 1$$

не обязано быть справедливым. Принимается, что любая имеющая смысл операция над вещественными числами, а также любая операция над целыми числами с нецелым результатом выполняются приближенно, причем степень приближения неизвестна. При выполнении алгоритма, записанного на АЛГОЛе, на конкретной вычислительной машине погрешность приближенных вычислений определяется возможностями этой машины, а также особенностями *транслятора* — программы, переводящей алгоритмы с АЛГОЛа на язык этой машины.

Логические значения — это «истина» и «ложь». Не следует искать слишком много смысла в этих терминах. Они применяются в ситуациях, допускающих лишь два возможных исхода, для обозначения этих исходов. При этом часто бывает совершенно безразлично, какой именно из исходов обозначен словом «истина». Например, некоторый прибор может быть исправен или поврежден. Человеку, пользующемуся этим прибором, может показаться более естествен-

ным связать слово «истина» с исправным состоянием прибора, тогда как механик, ответственный за ремонт, может склониться к тому, чтобы обозначить словом «истина» неисправное состояние, так как именно оно требует его внимания.

Многие объекты в АЛГОЛе способны принимать значения одного из названных *типов*: целые, приближенные или логические. Тип возможного значения объекта считается в таких случаях типом самого объекта.

§ 2. Обозначения

В АЛГОЛе существуют два рода обозначений. Одни обозначения применяются к объектам, понятиям и операциям, определенным в самом языке, другие — к объектам, вводимым при записи конкретного алгоритма. Обозначения первого рода называются *основными символами*, второго рода — *идентификаторами*.

Перечислим некоторые из основных символов. Символы

0 1 2 3 4 5 6 7 8 9

(десятичные *цифры*) обозначают соответствующие целые числовые значения. Символы

true false

изображают соответственно логические значения «истина» и «ложь». В АЛГОЛе используется свыше 20 таких *служебных слов*, выделяемых при наборе полужирным шрифтом, а при письме — подчеркиванием. Они рассматриваются как неделимые символы, имеющие тот смысл, который им предписан правилами языка. Поэтому мы и не делаем попытки заменить их аналогичными русскими словами.

Далее, символы

integer real Boolean

служат для обозначения типов: целый, вещественный и логический. Символы

+ − × / ÷ ↑

— это *знаки арифметических операций*, применяемых (за исключением знака ÷) как к целым, так и к вещественным числовым значениям. Смысл первых трех знаков понятен. Тип результата операции для них совпадает с типом операндов, если же операнды — разных типов (один целый, другой вещественный), то операция выполняется так, как будто оба операнда вещественные. Операция деления со знаком / всегда выполняется так, как если бы операнды были вещественными, и всегда дает вещественный (т. е. приближенный) результат. Именно поэтому значение выражения

$$(1/2) \times 2$$

может оказаться не точно равным числу 1.

Знаком $\div$ обозначена операция деления, которая определена лишь для целых чисел. Пусть m и n — значения делимого и делителя (делитель, как и в случае деления вещественных чисел, не должен быть равен нулю). Найдем такие числа q и r , что

$$|m| = q \cdot |n| + r, \quad 0 \leq r < |n|.$$

Тогда результат операции $\div$ равен q , если операнды m и n — одного знака, и $-q$, если их знаки различны.

Знак $\uparrow$ обозначает операцию возведения в степень. Она дает целый результат только тогда, когда целое число возводится в целую неотрицательную степень. Операция не имеет смысла, если число нуль возводится не в положительную степень, а также если при отрицательном основании показатель степени — вещественное число.

Символы

$$< \leq = \geq > \neq$$

называются *знаками операций отношения*. Эти операции выполняются над числовыми значениями и вырабатывают логические значения. Так, например, значение отношения

$$0 < 1$$

есть **true**, а значение

$$2 \times 2 = 5$$

false. Еще раз подчеркнем, что значения выражения

$$(1/2) \times 2 = 1$$

определено неоднозначно. Скорее всего, это **true**, но правила АЛГОЛа допускают здесь и значение **false**.

Символы

$$\neg \wedge \vee \supset \equiv$$

— это *знаки логических операций*: отрицания, конъюнкции (логического умножения), дизъюнкции (логического сложения), импликации и эквивалентности. Отрицание — операция одноместная, остальные операции двуместные. Правила употребления этих знаков и выработки результата операции сведены в табл. 1.

Таблица 1

Значения операндов		Значения результатов операций				
x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \supset y$	$x \equiv y$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

Среди прочих основных символов отметим три сорта *скобок*:
обыкновенные

$$(\quad)$$

индексные

$$[\quad]$$

и операторные

$$\text{begin} \quad \text{end}$$

а также знакомый нам по § 3 гл. 4 знак присваивания

$$:=$$

Эти символы наряду с рядом других, с которыми мы познакомимся позже, служат для объединения отдельных частей текста алгоритма на АЛГОЛе в более крупные.

К обозначениям первого рода примыкают обозначения числовых значений. *Целые числа* — это последовательности из одной или нескольких цифр, возможно, со знаком $+$ или $-$ впереди:

$$\begin{array}{cc} 0 & 1 \\ -1 & +1972 \end{array}$$

При записи *десятичных дробей* вместо привычной нам десятичной запятой используется точка. Если целая часть дроби равна нулю, ее можно опускать, так что

$$0.001 \quad 2.0 \quad -25.1 \quad 3.14159 \quad .1$$

— все это примеры записи десятичных дробей в АЛГОЛе. Нулевую дробную часть в записи чисел можно отбрасывать только вместе с точкой. Числа вида 10^n , где n — целое, записываются по следующему образцу:

$$_{10}1 \quad _{10}-5 \quad _{10}+3$$

где $_{10}$ — специальный основной символ. Такие обозначения чисел могут употребляться самостоятельно, а также могут быть приписаны к десятичной дроби или к записи целого числа в качестве множителя:

$$.314159_{10}1 \quad 5_{10}-10 \quad -1.972_{10}+3$$

Обозначения чисел, содержащие либо точку, либо символ $_{10}$, либо оба эти символа вместе, изображают приближенные числовые значения, так что равенство

$$.1 = _{10}-1$$

не обязано быть истинным (после перевода его левой и правой части в машинное представление могут возникнуть слегка отличающиеся друг от друга машинные числа).

Перейдем к обозначениям второго рода. В алгоритмах, записанных на АЛГОЛе, необходимо использовать переменные, группы переменных (массивы), функции и другие объекты, с которыми мы познакомимся позднее. Они обозначаются так называемыми *идентификаторами* — произвольными последовательностями букв и цифр, начинающимися обязательно с буквы, например

$$\begin{array}{ll} x & i \\ a1 & \log \\ \text{epsilon} & \text{jan1972} \end{array}$$

Официальный АЛГОЛ разрешает пользоваться при записи идентификаторов заглавными и строчными латинскими буквами и даже допускает введение букв других алфавитов. Однако не следует злоупотреблять разнообразием алфавитов. Практически всегда можно обойтись одним регистром (только строчными или только заглавными буквами) латинского алфавита. Так и следует поступать, потому что при этом отпадают некоторые технические трудности по подготовке программ к вводу в машину. В отличие от обозначений первого рода идентификаторы не имеют определенного назначения и смысла. В разных алгоритмах и, как мы увидим, даже в разных частях одного алгоритма идентификатор может обозначать разные объекты. Конкретное назначение каждого идентификатора в алгоритме должно быть специально определено. Для этого служат так называемые *описания* и *метки* при операторах. Однако идентификаторы

$$\begin{array}{lllll} \sin & \cos & \arctan & \ln & \exp \\ \text{sqrt} & \text{abs} & \text{sign} & \text{entier} & \end{array}$$

в исключение из общего правила могут использоваться без специальных описаний. Они обозначают широко применяемые в математике функции, которые в АЛГОЛе носят название *стандартных*. В частности, $\ln$ и $\exp$ обозначают натуральный логарифм и обратную ему показательную функцию при основании $e = 2,71828\dots$ Идентификатор sqrt обозначает квадратный корень, abs — абсолютную величину, entier — целую часть (наибольшее целое, не превосходящее значение аргумента), sign — функцию, определяемую правилом

$$\text{sign}(x) = \begin{cases} 1 & \text{при } x > 0, \\ 0 & \text{при } x = 0, \\ -1 & \text{при } x < 0. \end{cases}$$

Наряду с этими идентификаторами трансляторы с АЛГОЛа часто допускают использование других закрепленных идентификаторов tg (или tan), arctg (вместо или наряду с arctan), arcsin , ввод (или read), вывод (или print) и т. п.

§ 3. Переменные

В АЛГОЛе *переменные* используются так же, как мы их использовали в предыдущих главах, т. е. в качестве наименований, которые можно на какой-то период времени связывать с отдельными числовыми или логическими значениями. Каждая переменная имеет право принимать значения только одного из трех типов: целые, вещественные или логические. Возможный тип значений называется *типом* этой переменной. Он задается соответствующим описанием.

Переменные различаются, кроме того, на *простые переменные* и *переменные с индексами*. Простые переменные — это самостоятельные объекты, а переменные с индексами — это элементы так называемых *массивов*.

Простая переменная — это идентификатор, которому предписана эта роль так называемым *описанием типа*. Описание типа состоит из обозначения типа и списка идентификаторов переменных, разделенных запятыми, если список состоит больше чем из одного идентификатора. Например, описания типа

integer *n, i*
real *x, x1, x2, y*
Boolean *alpha*

говорят о том, что идентификаторы *n* и *i* могут обозначать любые целые числовые значения, идентификаторы *x, x1, x2* и *y* — любые вещественные значения, а идентификатор *alpha* — одно из двух логических значений.

Переменная с индексами — это идентификатор массива, за которым в квадратных скобках помещен набор из одного или нескольких выражений для индексов, разделенных запятыми, если их больше одного. Идентификатор называется *идентификатором массива*, если он включен в некоторое *описание массива*. Это описание задает для данного массива его *размерность* — число индексов, которыми должна быть снабжена каждая переменная с индексами, относящаяся к этому массиву, и границы, в которых должны лежать значения индексов. Структуру описания массива в простейшем случае лучше всего объяснить на примере. Описание

real array *a* [1:2, 1:3]

означает, что идентификатор *a* является идентификатором массива (по-английски — *аггау*) с вещественными элементами. Каждая переменная, относящаяся к этому массиву, должна быть снабжена двумя индексами, причем значение первого индекса должно быть заключено между 1 и 2, в второго — между 1 и 3. Таким образом, у массива *a* всего 6 элементов:

a[1,1] *a*[1,2] *a*[1,3] *a*[2,1] *a*[2,2] *a*[2,3]

В общем случае структура описания массивов такова. Оно начинается с обозначения типа, задающего тип значений элементов массива, причем тип **real** разрешается опускать. Затем обязательно пишется символ **array**, а за ним один или несколько *сегментов массивов*, разделенных запятыми. Каждый сегмент состоит из списка идентификаторов массивов, отделенных друг от друга запятыми, и заключенного в квадратные скобки списка *граничных пар*. Этот список относится ко всем массивам данного сегмента. Каждая граничная пара состоит из двух выражений для границ, разделенных двоеточием. Если в список входит более одной граничной пары, то эти пары разделяются запятыми.

В качестве выражений для границ в описаниях массивов и выражений для индексов у переменных с индексами могут применяться любые *арифметические выражения*. Арифметическое выражение в АЛГОЛе — это запись правила для вычисления одного числового значения. Пока мы не объяснили, как в общем случае составляются такие выражения, будем пользоваться простейшими частными случаями арифметических выражений — обозначениями чисел и переменными. Таким образом, возможны такие описания массивов:

integer array *m1, m2, m3* [*1 : n*], *c* [*0 : 2, -1 : 1*]
Boolean array *beta, gamma* [*0 : n*], *p, q* [*11 : 15*]

Каждое из этих описаний содержит по два сегмента массивов. В силу первого описания идентификаторы *m1, m2, m3* и *c* обозначают массивы с целочисленными элементами, идентификаторы *beta, gamma, p* и *q* — с логическими. У массива *c* — девять элементов, так как каждый из двух индексов может принимать по три значения. У массивов *p* и *q* — по пять элементов, индексы которых заключены в пределах от 11 до 15. Что касается массивов *m1, m2, m3, beta* и *gamma*, то у них верхняя граница значений индекса определяется в тот момент, когда эти описания массивов обрабатываются в ходе выполнения алгоритма, содержащего эти описания. Когда именно наступает такой момент — будет сказано в § 6. Значение верхней границы индексов этих массивов равно текущему значению переменной *n* в этот момент. Если при этом окажется, что значение переменной *n* равно нулю, то массивы *m1, m2* и *m3* не будут содержать ни одного элемента, так как не существует значений индекса, которые одновременно удовлетворяли бы ограничениям, накладываемым нижней и верхней границей. Массивы *beta* и *gamma* будут содержать по одному элементу, а именно *beta* [0] и *gamma* [0].

Переменная с индексами *c* [*i, k*] в условиях, о которых идет речь, может обозначать разные элементы массива *c* в зависимости от текущих значений переменных *i* и *k*. Однако эта переменная имеет смысл, если только соблюдены ограничения $0 \leq i \leq 2, -1 \leq k \leq 1$. При нарушении хотя бы одного из этих условий переменная *c* [*i, k*] ничего не обозначает.

§ 4. Выражения

Выражения в АЛГОЛе — это тексты, задающие правило вычисления одного числового или логического значения. В этом отношении они аналогичны тому, что называется выражением в обычной математике. Выражения строятся из переменных и обозначений числовых и логических значений с помощью знаков операций и скобок. В простейшем случае сами переменные и обозначения значений образуют выражения, например,

$$\begin{array}{c} 0 \\ \text{true} \\ 2.39_{10} - 8 \\ x1 \\ \text{beta } [0] \end{array}$$

Вот примеры выражений, содержащих знаки операций:

$$\begin{array}{c} n + 1 \\ (x + x1) \times (x1 - x2) \\ (a[2, 2] - i) \uparrow 3 \\ x > 0 \\ (0 \leq m1[1]) \wedge (m1[1] \leq n) \end{array}$$

Смысл этих выражений ясен — они предписывают произвести указанные операции над текущими значениями переменных и заданными числовыми значениями. Порядок операций определяется скобками. Если присутствующих в выражении скобок недостаточно для установления порядка выполнения операций, то следует руководствоваться правилами *старшинства* операций. По старшинству операции располагаются в следующем порядке:

$$\begin{array}{c} \uparrow \\ \times \quad / \quad \div \\ + \quad - \\ < \quad \leq \quad = \quad \geq \quad > \quad \neq \\ \lceil \quad \rfloor \\ \wedge \quad \vee \\ \supset \quad \equiv \end{array}$$

Операции с большим старшинством (начиная с $\uparrow$) выполняются раньше, чем операции с меньшим старшинством. Например, выражение

$$x \times a[i, 3] - x2 \uparrow n/m1[1] \uparrow 2$$

эквивалентно выражению

$$(x \times a[i, 3]) - ((x2 \uparrow n)/(m1[1] \uparrow 2))$$

Одинаковые операции, а также операции с одинаковым старшинством (знаки которых выписаны в одной строчке) выполняются в последовательности слева направо. Таким образом, выражения

$$a[1, 1] \times x/m3[n] \uparrow i \uparrow 2 \times c[0, -1] - 5 \times x2 + .15$$

и

$$(((a[1, 1] \times x) / ((m3[n] \uparrow i) \uparrow 2)) \times c[0, -1]) - (5 \times x2)) + .15$$

эквивалентны.

Разумеется, выражение должно быть построено так, чтобы оно предписывало выполнять арифметические операции и операции отношения только над числовыми значениями, а логические операции — только над логическими значениями. Если вычисляемое значение — числовое, то выражение называется *арифметическим*. Если же выражение служит для вычисления логического значения, то оно называется *логическим*.

Примеры логических выражений нам уже встречались, а именно

$$\begin{array}{c} \text{true} \\ \text{beta}[0] \\ x > 0 \\ 0 \leq m1[1] \wedge m1[1] \leq n \end{array}$$

(ранее последний пример был записан с избыточными скобками). Вот еще один пример:

$$p[12] \equiv \neg \text{alpha} \wedge \text{gamma}[i] \wedge \neg q[15] \vee x \neq x1 \supset \neg \text{beta}[0]$$

и эквивалентная ему запись со скобками:

$$p[12] \equiv ((((\neg \text{alpha}) \wedge \text{gamma}[i]) \wedge (\neg q[15])) \vee (x \neq x1)) \supset (\neg \text{beta}[0]))$$

Арифметические выражения от логических можно различить по их виду (если выражение содержит хотя бы одну логическую переменную или значение, или хотя бы один знак операции отношения или логической операции, то оно — логическое). Напротив, опеределить тип значения арифметического выражения по его виду нельзя, даже если из описаний известны типы всех входящих в него переменных. Например, выражение

$$2 \uparrow i$$

где i — переменная типа «целый» может вырабатывать как целый, так и вещественный результат в зависимости от знака текущего значения переменной i . Поэтому в АЛГОЛе арифметические выражения не различаются по типам.

Иногда от арифметического выражения требуется, чтобы оно вырабатывало целочисленный результат, например, если оно занимает позицию индекса переменной или границы индекса в описании мас-

символ. В таких случаях, если вычисленное значение выражения оказывается не целым, оно округляется до ближайшего целого.

До сих пор мы рассматривали только выражения, называемые *простыми* (или *безусловными*). В математике часто прибегают к записям вида

$$|x| = \begin{cases} x & \text{при } x \geq 0, \\ -x & \text{в противном случае (т. е. при } x < 0). \end{cases}$$

Аналогичная конструкция, называемая *условным выражением*, есть и в АЛГОЛе. Только что приведенному примеру соответствует условное выражение

if $x \geq 0$ then x else $-x$

(английские слова if, then и else означают «если», «то» и «иначе»). В общем случае условное выражение имеет вид

if B then E_1 else E

где B заменяет логическое выражение, E_1 — простое (безусловное) выражение, E — произвольное выражение с тем лишь ограничением, что выражения E_1 и E должны быть оба арифметическими или оба логическими. Соответственно и все условное выражение считается либо арифметическим, либо логическим. Выражение E может быть снова условным. Это позволяет строить выражения вида

if B_1 then E_1 else if B_2 then E_2 else ...
... else if B_{n-1} then E_{n-1} else E_n

где $E_1, E_2, \dots, E_{n-1}, E_n$ — простые выражения.

Значение условного выражения

if B then E_1 else E

совпадает либо со значением выражения E_1 , либо со значением выражения E. Чтобы выбрать одну из этих двух возможностей, сначала вычисляется значение логического выражения B. Если значение B — «истина», то выбирается выражение E_1 , если — «ложь», то E. Затем вычисляется значение выбранного выражения и принимается за значение всего условного выражения.

Если применить этот принцип к условному выражению

if B_1 then E_1 else ... else if B_{n-1} then E_{n-1} else E_n

то придем к следующему правилу. Поочередно вычисляются значения логических выражений $B_1, B_2, \dots$ до тех пор, пока не встретится выражение B_i ($1 \leq i \leq n-1$) со значением «истина» или пока весь набор $B_1, \dots, B_{n-1}$ не будет исчерпан (если все B_i имеют значение «ложь»). В первом случае выбирается выражение E_i , в последнем — E_n . Значение выбранного выражения является значением данного условного выражения.

Из условного выражения можно получить эквивалентное ему безусловное, заключив его в скобки. Полученное безусловное выражение можно наравне с любыми другими выражениями в скобках, а также переменными и обозначениями значений использовать в качестве операндов операций и строить таким образом сколь угодно сложные выражения. Вот несколько примеров на применение логических и условных выражений.

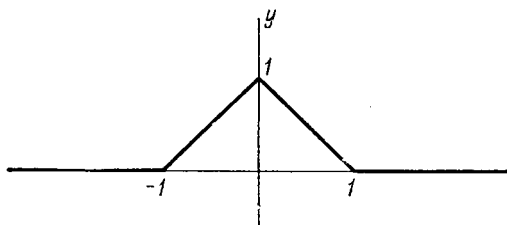


Рис. 8.4.1. Пример функции.

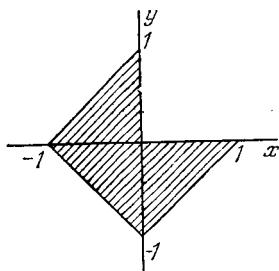


Рис. 8.4.2. Пример области.

Большее из значений переменных x и $x1$ определяется выражением

if $x > x1$ then x else $x1$

Заметно сложнее выглядит выражение для наибольшего из значений трех переменных x , $x1$ и $x2$:

if $x > x1$ then (if $x > x2$ then x else $x2$)
else if $x1 > x2$ then $x1$ else $x2$

или

if $x \geq x1 \wedge x \geq x2$ then x
else if $x1 \geq x2$ then $x1$
else $x2$

Выражение, эквивалентное значению стандартной функции *sign* от аргумента x , таково:

if $x > 0$ then 1 else if $x < 0$ then -1 else 0

Зависимость, график которой изображен на рис. 8.4.1, может быть задана выражением

if $x \leq -1$ then 0
else if $x \leq 0$ then $x + 1$
else if $x \leq 1$ then $1 - x$
else 0

Функция, принимающая значение «истина» внутри области, изображенной на рис. 8.4.2, и значение «ложь» вне этой области,

определяется выражением

$$\begin{aligned} x \leq 0 \wedge x + y + 1 \geq 0 \wedge y - x \leq 1 \\ \vee x > 0 \wedge y \geq x - 1 \wedge y \leq 0 \end{aligned}$$

или

$$\begin{aligned} \text{if } x \leq 0 \text{ then } y \geq -x - 1 \wedge y \leq x + 1 \\ \text{else } y \geq x - 1 \wedge y \leq 0 \end{aligned}$$

Типичная ошибка начинающих программистов состоит в том, что они пишут

if B then true else false

там, где достаточно написать просто B. Оба эти выражения совершенно эквивалентны.

Число элементов, имеющих значение «ложь», логического массива p , описание которого было дано на стр. 130, можно вычислить с помощью выражения

$$\begin{aligned} &(\text{if } p[11] \text{ then } 0 \text{ else } 1) + (\text{if } p[12] \text{ then } 0 \text{ else } 1) \\ &+ (\text{if } p[13] \text{ then } 0 \text{ else } 1) + (\text{if } p[14] \text{ then } 0 \text{ else } 1) \\ &+ (\text{if } p[15] \text{ then } 0 \text{ else } 1) \end{aligned}$$

§ 5. Операторы присваивания

Описания типов и описания массивов позволяют вводить в рассмотрение новые обозначения (переменные), но не связывают эти обозначения ни с какими конкретными значениями. Выражения позволяют образовывать новые значения, но не снабжают эти значения никакими наименованиями. Связь между переменной и вновь вычисленным значением выражения устанавливается с помощью *оператора присваивания*. Структура этого оператора такова. Выписывается переменная, затем символ присваивания $:=$ и выражение, значение которого требуется вычислить и присвоить этой переменной, т. е. связать с ней. Примеры:

$$n := 1$$

$$a[1, 1] := \text{if } x \leq x1 \text{ then } x \text{ else } x1$$

$$beta[i] := m2[i] \neq m3[i]$$

Естественно, что тип переменной должен быть согласован с видом значения — если в левой части оператора присваивания стоит логическая переменная, то тогда (и только тогда) выражение в правой части оператора должно быть логическим. Если переменная в левой части — вещественная или целая, то справа может стоять любое арифметическое выражение. Если при этом вещественное значение требуется присвоить целой переменной, то оно округляется до ближайшего целого числа.

Допускаются операторы, присваивающие значение выражения сразу нескольким переменным, например,

$$x := y := a[i, 3] := 0$$

Символ присваивания ставится после каждой из переменных левой части. Здесь требуется, чтобы все эти переменные были в точности одного типа.

Выражения для индексов у переменных в левой части вычисляются до того, как начнется вычисление выражения в правой части. Если, например, переменные i и k перед выполнением оператора

$$i := k := c[i + k, i] := 2$$

имели значения 0 и 1, то в результате выполнения этого оператора значение 2 получит переменная $c[1, 0]$, а также переменные i, k .

§ 6. Блоки

Основным средством объединения составных частей АЛГОЛ-программы (так обычно называются алгоритмы, написанные на АЛГОЛе) в одно целое является *блок*. Блок открывается символом **begin** (по русски — начало), за которым следует последовательность из одного или нескольких описаний, далее следует один или несколько операторов и все это завершается символом **end** (конец). После каждого описания, а также между соседними операторами должен стоять разделительный символ ; (точка с запятой). Пример блока:

```
begin real pi; integer n; n := 1; pi := 3.14159 end
```

Каждый блок сам является оператором и поэтому может содержаться в роли оператора в другом блоке. Вся АЛГОЛ-программа также, как правило, представляет собой блок. Кроме блоков и операторов присваивания имеются другие виды операторов, с которыми мы познакомимся позже. Каждый оператор при необходимости может быть помечен одной или несколькими *метками*. Метка состоит из *идентификатора метки* и двоеточия и ставится перед оператором, который она метит. Идентификатор метки используется в операторах перехода в качестве наименования оператора, помеченного этой меткой. Примеры помеченных операторов:

```
l: x := 0.1
```

```
B0: start: w: begin Boolean omega; w1: omega := false end
```

Как мы вскоре увидим, оба приведенных выше примера блоков носят формальный характер и не могут с пользой быть включены ни в одну АЛГОЛ-программу.

Идентификатор величины, описанной в одном из описаний в начале блока, или метки, поставленной перед одним из операторов

этого блока, называется *локальным* в этом блоке. Все прочие идентификаторы, встречающиеся в этом блоке, считаются в нем *глобальными*. Каждый глобальный идентификатор должен быть локальным в одном из блоков, охватывающих данный блок. Таким образом, в самом внешнем блоке АЛГОЛ-программы не может быть глобальных идентификаторов. Напомним, однако, что для идентификаторов, закрепленных за стандартными функциями и за некоторыми другими объектами, допускается исключение из этого правила. Локальные идентификаторы блока становятся глобальными в его подблоках, если в последних не локализованы такие же идентификаторы. Локализация одного и того же идентификатора в разных блоках допускается как в случае непересекающихся блоков, так и в случае, когда один из блоков целиком вложен в другой. При этом в каждом из таких блоков идентификатор представляет свой объект, недоступный другому блоку. Напротив, в одном блоке любой идентификатор может быть локализован только одним способом: либо описан (только в одном из описаний), либо включен в состав метки (лишь одной из меток операторов данного блока).

Для любого *вхождения* идентификатора в программу (так называется появление идентификатора в каком-либо месте программы) его роль в этом месте может быть установлена следующим образом. Выбирается самый внутренний из всех блоков, охватывающих это вхождение, в которых локализован данный идентификатор. Если таких блоков нет, то этот идентификатор может быть только идентификатором, закрепленным за некоторым стандартным объектом (см. стр. 128), и тогда его роль определена независимо от программы. В противном случае идентификатор в рассматриваемом вхождении представляет тот же объект, что в выбранном блоке.

Выполнение блока состоит в том, что создаются *локальные объекты*, представленные всеми описанными в начале блока идентификаторами (в этот момент и определяются границы индексов массивов), а затем начинают выполняться внутренние операторы этого блока в том порядке, в каком они написаны. Если эти операторы сами не заставят изменить этот порядок (см. § 7), то выполнение блока завершится, когда будет выполнен последний из внутренних операторов. В этот момент все локальные объекты блока перестают существовать.

Рассмотрим в качестве примера следующую АЛГОЛ-программу:

```

B0: begin real a; integer b; Boolean c;
      a := 1; b := 2; c := true; ...;
B1: begin real b; Boolean c;
      b := 3; c := false; ...
      end;
      a := if c then b else 4; ...;

```

```

B2: begin Boolean c; real d;
      a := 5.1; ...
      end;
      b := a; ...
end

```

Предположим, что других блоков, описаний и операторов присваивания с идентификаторами a , b и c в левой части в программе нет. Тогда идентификатор a локален в блоке $B0$, глобален в блоках $B1$ и $B2$ и всюду представляет одну и ту же вещественную переменную. Идентификатор b локален в блоках $B0$ и $B1$ и представляет целую переменную в блоке $B0$ и вещественную (совсем другую) переменную в блоке $B1$. В блоке $B2$ этот идентификатор глобален и представляет ту же целую переменную, что в блоке $B0$. Идентификатор c локален во всех трех блоках и всюду представляет логические переменные, но эти переменные — разные. Идентификатор d локален в блоке $B2$ и не имеет смысла (не может быть использован) в блоках $B0$ и $B1$.

Рассмотрим оператор

```
a := if c then b else 4
```

Он находится в блоке $B0$, поэтому идентификатор c обозначает в нем ту переменную, которая в этом же блоке получила значение **true**. Присваивание

```
c := false
```

было выполнено в блоке $B1$, и следовательно, значение **false** получила другая переменная, уже прекратившая свое существование к моменту выполнения рассматриваемого оператора. Таким образом этот оператор присваивает переменной a текущее значение переменной b . Это значение из тех же соображений равно 2 (a не 3).

Хотя присваивание

```
a := 5.1
```

выполняется во внутреннем блоке $B2$, значение 5.1 получает переменная a , глобальная в этом блоке. После завершения выполнения блока $B2$ выполняется оператор присваивания

```
b := a
```

Поскольку переменная b в блоке $B0$ — целая, присваиваемое ей значение 5.1 округлится до ближайшего целого, т. е. до 5.

Переменные, встречающиеся в граничных парах в описаниях массивов, должны обладать определенными значениями и, следовательно, обязаны быть глобальными.

§ 7. Операторы перехода

Операторы перехода прерывают ту естественную последовательность выполнения операторов, о которой говорилось в предыдущем параграфе, и явно указывают, какой оператор должен выполняться следующим. Оператор перехода состоит из символа *go to* (перейти на) и так называемого *именующего выражения*. Именуемое выражение может быть простым или условным. Простое именуемое выражение — это либо идентификатор метки, либо взятое в скобки условное именуемое выражение. Условное именуемое выражение имеет такую же структуру, как условные арифметические и логические выражения, а именно

if B then D_1 else D

где B — логическое выражение, D_1 — простое именуемое выражение, D — произвольное именуемое выражение. Значением именуемого выражения является некоторая метка. Это та метка, которая представлена идентификатором метки, либо заданным непосредственно, либо выбираемым в результате проверки одного или нескольких условий вида

if B then

Примеры операторов перехода:

go to l
go to if $x < 0$ then (if p [15] then w else w1)
else if $i \neq n$ then continue else exit

где l, w, w1, continue, exit — идентификаторы меток.

Выполнение оператора перехода состоит в том, что определяется метка, являющаяся значением именуемого выражения в этом операторе, после чего начинает выполняться оператор, помеченный найденной меткой.

Пусть требуется вычислить сумму

$$s = 1 + 1/2 + \dots + 1/n$$

Для этого можно воспользоваться следующей группой операторов:

$i := 0; s := 0;$
continue: $i := i + 1; s := s + 1/i;$
go to if $i = n$ then exit else continue;
exit:

где метка exit: метит оператор, следующий за оператором перехода.

Метка, представленная идентификатором метки, найденным в результате вычисления именуемого выражения, определяется в соответствии с правилами локализации идентификаторов в блоках, сформулированными в предыдущем разделе. Может случиться,

что эта метка находится за пределами блока B_1 , одним из операторов которого является выполняемый оператор перехода. Пусть идентификатор метки локализован в блоке B_2 , охватывающем блок B_1 . Тогда выполнение блока B_1 , а также всех блоков, содержащих блок B_1 и содержащихся в блоке B_2 , прекращается. При этом перестают существовать все объекты, представленные всеми идентификаторами, локальными в этих блоках.

Поясним сказанное на примере. Пусть программа имеет следующую структуру (показаны только границы блоков, один из операторов перехода к метке l : и все метки с этим же идентификатором, расположенные в разных блоках):

```

B0: begin ...
    l: ...
    B1: begin ...
        B2: begin ...
            go to l; ...
        B3: begin ...
            l: ...
            end; ...
        end; ...
    B4: begin ...
        l: ...
        end; ...
    l: ...
    end; ...
end

```

Оператор перехода `go to l` не охватывается блоками $B3$ и $B4$. Поэтому во время выполнения этого оператора идентификатор l не может представлять метки, содержащиеся в этих блоках. В блоке $B2$ идентификатор l не локализован. Поэтому обращаемся к блоку $B1$, непосредственно охватывающему блок $B2$, и обнаруживаем в нем метку (самую нижнюю на схеме) с этим идентификатором. К этой метке и ведет оператор перехода. Выполнение блока $B2$ при этом заканчивается.

Из этих разъяснений следует, в частности, что оператор перехода, расположенный вне некоторого блока, не может вести к метке, метящей один из операторов этого блока (в нашем примере — блока $B4$ или $B3$).

§ 8. Операторы ввода и вывода

Операторов, о которых пойдет речь в этом параграфе, в официальном АЛГОЛе нет. Однако без них невозможно завершить составление ни одной программы. Действительно, после выхода из самого внешнего блока программы перестанут существовать все

объекты, представленные идентификаторами программы, и следовательно, станут недоступными все вычисленные значения. Нет также средств сообщить АЛГОЛ-программе какие-либо исходные данные для ее работы, кроме тех, которые явно включены в текст самой программы.

Поэтому наряду с внутренними объектами, представленными идентификаторами программы, введем понятие о *внешней среде*, в которой можно запастись последовательности исходных данных — числовых и логических значений, и в которую программа может последовательно передавать выработанные в ней значения. Для передачи данных из внешней среды (или, как еще говорят, с *внешнего носителя* данных) переменным программы будем пользоваться *операторами ввода*. Оператор ввода состоит из закрепленного идентификатора *ввод* и следующего за ним в круглых скобках списка *параметров ввода*. Параметром ввода может быть либо переменная, либо идентификатор массива. Число параметров ввода произвольно. Отделяются они друг от друга запятыми. Пример оператора ввода:

ввод (*n*, *x*, *p*, *a*, *c* [*0*, *1*])

При выполнении оператора ввода очередная группа значений, запасенных во внешней среде, присваивается параметрам ввода в указанной в нем последовательности. Присваивание значений массиву означает присваивание всем его элементам. В нашем примере первое из очередных значений, подготовленных во внешней среде, присваивается переменной *n*, следующее — переменной *x*, следующие значения — элементам логического массива *p* и т. д. Предполагается, что соответствие числа и типов значений, запасенных во внешней среде, характеристикам параметров ввода обеспечивают те, кто готовит эти значения. В рассматриваемом примере первое вводимое значение должно быть целым, следующее — вещественным, следующие пять — логическими и т. д.

Оператор вывода состоит из закрепленного идентификатора *вывод* и заключенного в скобки списка *параметров вывода*. Параметром вывода может быть арифметическое или логическое выражение (в частности, число, логическое значение или переменная) или идентификатор массива. Пример оператора вывода:

вывод (*6*, *a* [*i*, *1*], *a* [*i*, *2*], *beta*, $x_1 + x_2 \uparrow 2$, *alpha*)

Будут выведены число 6, значения двух указанных элементов массива *a*, значения всех элементов массива *beta*, значение выражения $x_1 + x_2 \uparrow 2$, значение переменной *alpha*.

Используя операторы ввода и вывода, мы можем завершить составление программы для вычисления суммы $1 + 1/2 + \dots + 1/n$, начатое в предыдущем разделе. Если мы хотим, чтобы значение *n*

могло быть произвольным, программа должна быть такой:

```
begin integer n, i; real s; ввод (n); i := 0; s := 0;
continue: i := i + 1; s := s + 1/i;
go to if i = n then exit else continue;
exit: вывод (n, s) end
```

Перед началом работы программы во внешней среде надо запасти одно целочисленное значение, которое во время выполнения оператора ввода будет присвоено переменной n . Начальные значения переменным i и s программа задает сама. В результате своей работы программа выведет значение n , с которым она проводила расчет, и вычисленное значение s . Вывод значения n полезен для контроля.

§ 9. Условные и составные операторы

Подобно условным выражениям, *условные операторы* служат для того, чтобы в зависимости от результата проверки некоторого условия выбрать тот или иной способ действий, конкретнее — выполнить тот или иной оператор. Структура условного оператора:

if B then S_1 else S

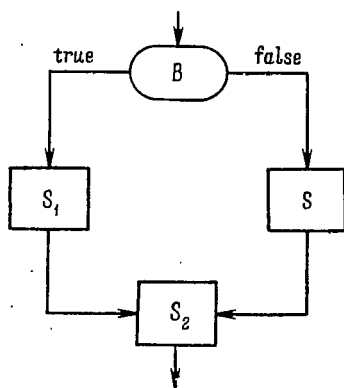


Рис. 8.9.1. Блок-схема выполнения условного оператора.

где B — логическое выражение, S_1 — безусловный (т. е. не условный) оператор, S — произвольный оператор. Выполнение условного оператора начинается с вычисления логического выражения B . Если это вычисление дает значение «истина», то выбирается оператор S_1 , в противном случае — оператор S . Выбранный оператор выполняется и на этом выполнение условного оператора заканчивается.

Пусть S_2 — оператор, который должен выполняться после рассматриваемого условного оператора. Тогда работу условного оператора можно пояснить блок-схемой, изображенной на рис. 8.9.1. Однако, если внутренний оператор S_1 или S (или оба) является оператором перехода, то стрелку, ведущую от соответствующего блока схемы к блоку S_2 , надо выбросить.

Пример условного оператора:

if $i < n$ then $i := i + 1$ else go to l

Часто с помощью условных операторов можно сделать то же самое, что делают безусловные операторы, содержащие условные

выражения. Например, оператор присваивания

$$y := \text{if } x \geq x1 \text{ then } x \text{ else } x1$$

можно заменить в точности эквивалентным ему условным оператором

$$\text{if } x \geq x1 \text{ then } y := x \text{ else } y := x1$$

Оператор

$$\text{go to if } i = n \text{ then } \textit{exit} \text{ else } \textit{continue}$$

можно заменить оператором

$$\begin{aligned} &\text{if } i = n \text{ then go to } \textit{exit} \\ &\text{else go to } \textit{continue} \end{aligned}$$

или

$$\begin{aligned} &\text{if } i \neq n \text{ then go to } \textit{continue} \\ &\text{else go to } \textit{exit} \end{aligned}$$

Варианты, в которых используются условные операторы вместо условных выражений, чуть более громоздки, но многие программисты пользуются ими охотнее. Если конечная цель работы программиста — составить машинную программу, то так и следует поступать, потому что структура этих вариантов ближе к структуре соответствующих участков машинной программы.

Очень часто при построении алгоритмов возникают ситуации, которые хорошо отражает блок-схема на рис. 8.9.1, но функции блоков S_1 и (или) S не могут быть описаны каким-либо одним из известных нам операторов АЛГОЛа. В таких случаях на помощь приходит понятие *составного оператора*. Внешне составной оператор очень напоминает блок и отличается от него лишь тем, что не содержит описаний, а только операторы. Иначе говоря, составной оператор — это последовательность из одного или нескольких операторов, разделенных точками с запятой, заключенная в операторные скобки **begin end**. Выполнение составного оператора сводится к последовательному выполнению операторов, из которых он составлен. Как и в случае блока, операторы перехода могут изменять эту естественную последовательность выполнения внутренних операторов и даже выводить за пределы составного оператора.

Идентификаторы в составных операторах не локализуются. Поэтому возможен переход от оператора, расположенного вне составного оператора, к метке, метящей один из его внутренних операторов.

Пользуясь введенными в этом разделе понятиями, можно так переписать программу вычисления суммы $1 + 1/2 + \dots + 1/n$:

```
begin integer i, n; real s; read (n); i := 0; s := 0;
  l1 if i < n then
    begin i := i + 1; s := s + 1/i; go to l end
    else write (s)
end
```

На случай, если при составлении алгоритма окажется, что оператор S не должен содержать никаких действий, АЛГОЛ допускает сокращенную конструкцию условного оператора, называемую также *оператором «если»*:

if B then S_1

Выполнение такого оператора, за которым выполняется оператор S_2 , показано на рис. 8.9.2. Если прибегнуть к этой конструкции, то предыдущая программа принимает более естественный вид:

```
begin integer i, n; real s; ввод (n);
      i := 0; s := 0;
  l:  if i < n then
      begin i := i + 1; s := s + 1/i; go to l end;
      вывод (s)
end
```

В качестве еще одного примера составим программу решения уравнения не выше второй степени (см. пример 7.3.2):

```
begin real a, b, c, d, x1, x2; integer s;
      ввод (a, b, c);
      if a ≠ 0 then
      begin x1 := -b/(2×a); d := x1↑2 - c/a;
            if d > 0 then
            begin s := 2; d := sqrt(d);
                  x2 := x1 - d; x1 := x1 + d end
            else if d < 0 then
            begin s := 3; x2 := sqrt(-d) end
            else s := 4
            end
      else if b ≠ 0 then
      begin s := 1; x1 := -c/b end
      else if c ≠ 0 then s := 0 else s := 5
      вывод (s, x1, x2)
end
```

Следует обратить внимание на то, как в этой программе осуществляется выход из разветвленного условного оператора. Пусть, например, оказалось, что при выполнении этой программы $a \neq 0$ и $d < 0$. Тогда выполнение оператора $x2 := \text{sqrt}(-d)$ завершает выполнение составного оператора, в котором этот оператор присваивания является последним из внутренних операторов. Поскольку этот составной оператор является одним из внутренних операторов условного оператора с условием $\text{if } d < 0 \text{ then}$, то завершается выполнение и этого условного оператора. Но он в свою очередь является одним из внутренних операторов условного оператора с ус-

ловием $\text{if } d > 0 \text{ then}$, так что выполнение этого условного оператора также завершается. Вместе с тем завершается выполнение составного оператора, в котором этот условный оператор занимает место непосредственно перед **end**. Но этот составной оператор сам является внутренним оператором условного оператора с условием $\text{if } a \neq 0 \text{ then}$. Таким образом, завершается выполнение и этого условного оператора и начинается выполнение следующего за ним оператора *вывод* ($s, x1, x2$). Никаких явных указаний на то, что вслед за оператором $x2 := \text{sqrt}(-d)$ (а также за операторами $x1 := x1 + d, s := 4, x1 := -c/b, s := 0$ и $s := 5$) должен выполняться оператор вывода, не требуется — это вытекает из правил выполнения условных и составных операторов.

Написанную программу можно сократить, избавляясь от лишних присваиваний и вывода значений, не представляющих интереса в тех или иных частных случаях, и используя вместо переменных d и $x1$ исходные переменные a и b .

Улучшенная программа такова:

```
begin real a, b, c; ввод (a, b, c); вывод (a, b, c);
  if  $a \neq 0$  then
    begin  $b := -b/(2 \times a); a := b^2 - c/a;$ 
      if  $a > 0$  then
        begin  $a := \text{sqrt}(a); \text{вывод}(2, b + a, b - a)$  end
      else if  $a < 0$  then  $\text{вывод}(3, b, \text{sqrt}(-a))$ 
      else  $\text{вывод}(4, b)$ 
    end
  else if  $b \neq 0$  then  $\text{вывод}(1, -c/b)$ 
  else if  $c \neq 0$  then  $\text{вывод}(0)$ 
  else  $\text{вывод}(5)$ 
end
```

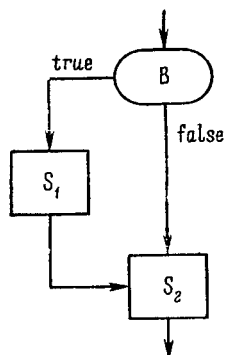


Рис. 8.9.2. Блок-схема выполнения оператора «если».

вспомогательных

§ 10. Операторы цикла и пустые операторы

Хотя циклические процессы можно, как мы видели, описывать с помощью операторов перехода и условных операторов, в АЛГОЛе существует и специальный аппарат для этой цели — так называемые *операторы цикла*. Ими и рекомендуется пользоваться в первую очередь, так как при этом программы становятся более обзримыми и легче контролируемые.

Оператор цикла состоит из *заголовка цикла* и внутреннего оператора, называемого также *телом цикла*. Телом цикла может быть

любой оператор, который требуется повторить несколько раз (может быть, лишь один раз или даже ни разу). Заголовок цикла содержит некоторую целую или вещественную переменную, называемую *параметром цикла*, и *список цикла*, с помощью которого задается последовательность значений, присваиваемых параметру цикла. С каждым из этих значений параметра выполняется тело цикла.

Заголовок цикла имеет следующую структуру:

for $V := L$ do

где **for** (для) и **do** (выполнить) — основные символы, вместо V должен стоять параметр цикла, а вместо L — список цикла. Список цикла — это последовательность *элементов списка цикла*, разделенных запятыми. Элемент списка цикла может быть одного из трех видов:

A
A step A_1 until A_2
A while B

где обозначения A , A_1 и A_2 заменяют арифметические выражения, B — логическое выражение; **step** (шаг), **until** (до) и **while** (пока) — основные символы АЛГОЛа.

Элемент списка цикла, состоящий из одного арифметического выражения A , предписывает параметру цикла одно значение — значение выражения A .

Элемент списка цикла вида

A step A_1 until A_2

задает параметру цикла последовательность значений, вырабатываемую следующим образом. Начальное значение в этой последовательности равно значению выражения A . Каждое следующее получается из текущего значения параметра цикла добавлением к нему значения выражения A_1 (шага). Если очередное значение параметра не превосходит значения выражения A_2 при неотрицательном шаге или не меньше значения A_2 при отрицательном шаге, то с этим значением параметра выполняется тело цикла. В противном случае последовательность значений, предписанных параметру цикла данным элементом списка цикла, считается исчерпанной. Все это можно следующим образом изобразить средствами АЛГОЛа:

V := A;
l: if $(V - (A_2)) \times \text{sign}(A_1) \leq 0$ then
begin S; V := V + A_1 ; go to l end

где S заменяет тело цикла.

На практике в подавляющем большинстве случаев выражения A_1 и A_2 не зависят от параметра цикла V , а выполнение тела цикла S не влияет ни на значение параметра цикла V , ни на значения выра-

жений A_1 и A_2 при их повторном вычислении. При этих условиях элемент рассматриваемого вида задает параметру цикла значения, образующие арифметическую прогрессию с начальным элементом A и шагом A_1 , обрывающуюся, когда ее очередной член (не входящий в последовательность) «перескакивает» через A_2 . Поэтому такой элемент и называется элементом вида арифметической прогрессии.

Перепишем программу вычисления суммы $1 + 1/2 + \dots + 1/n$, используя оператор цикла со списком цикла, состоящим из одного элемента вида арифметической прогрессии

```
begin real s; integer i, n; ввод (n); s := 0;
  for i := 1 step 1 until n do s := s + 1/i;
  вывод (s)
end
```

Вот еще несколько примеров применения оператора цикла со списком цикла, состоящим из одного элемента вида арифметической прогрессии.

Пусть задан массив

array a [0 : n]

коэффициентов многочлена $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ и требуется вычислить значение y этого многочлена в точке x . Это можно сделать с помощью операторов:

```
y := a[0];
for i := 1 step 1 until n do
  y := y * x + a[i]
```

Чтобы подсчитать число m неотрицательных элементов массива

array u [1 : n]

можно воспользоваться операторами

```
m := 0;
for i := 1 step 1 until n do
  if u[i] ≥ 0 then m := m + 1
```

Здесь тело цикла образовано условным оператором.

Чтобы среди элементов того же массива u найти наибольший и определить его номер k , следует написать операторы

```
k := 1; y := u[1];
for i := 2 step 1 until n do
  if u[i] > y then
    begin k := i; y := u[i] end
```

Пусть требуется вычислить сумму

$$s = \sqrt{0.01} + \sqrt{0.02} + \dots + \sqrt{1.00}$$

Для этого можно написать операторы

```
s := 0;
for i := 1 step 1 until 100 do s := s + sqrt(i/100)
```

но лучше прибегнуть к операторам

```
s := 0;
for x := 0.01 step 0.01 until 1.005 do
  s := s + sqrt(x)
```

позволяющим сэкономить 100 операций деления во время выполнения цикла. Во втором варианте параметр цикла x — вещественная переменная, значения которой вычисляются приближенно, поэтому верхний предел значений параметра задан с запасом в половину шага.

Элемент списка цикла вида

A while B

называемый *элементом итерационного типа*, задает последовательность значений параметра, вырабатываемую в результате многократного вычисления выражения A. Если после присваивания параметру цикла очередного значения из этой последовательности выражение B принимает значение «истина», то выполняется тело цикла, в противном случае последовательность, предписанная данным элементом, обрывается. Иначе это можно описать с помощью операторов:

```
l: V := A;
   if B then
     begin S; go to l end
```

Любой из приведенных выше примеров можно переписать, используя вместо элемента вида арифметической прогрессии элемент итерационного вида. Например, программу вычисления суммы $1 + 1/2 + \dots + 1/n$ можно составить так:

```
begin real s; integer i, n; ввод (n);
  s := 0; i := 0;
  for i := i + 1 while i ≤ n do s := s + 1/i;
  вывод(s)
end
```

Для поиска максимального элемента массива u и определения номера этого элемента можно прибегнуть к операторам

```
i := k := 1; y := u[1];
for i := i + 1 while i ≤ n do
  if u[i] > y then
    begin k := i; y := u[i] end
```


Однако чаще к элементу итерационного вида прибегают, когда ни одна из переменных не изменяется в цикле по арифметической прогрессии, а число повторений цикла заранее не известно. Например, пусть даны числа x и y , такие, что $x > y > 0$ и требуется найти их арифметико-геометрическое среднее z — общий предел двух последовательностей $x_1, x_2, \dots$ и $y_1, y_2, \dots$, где

$$x_1 = x, y_1 = y, x_{i+1} = \frac{x_i + y_i}{2}, y_{i+1} = \sqrt{x_i y_i}, \quad i = 1, 2, \dots \quad (1)$$

Пусть заданное число $\epsilon > 0$ определяет требуемую точность вычисления z . Тогда вычисления по формулам (1) можно прекратить, как только разность $x_i - y_i$ станет меньше или равна ϵ . На АЛГОЛе этот процесс можно описать программой

```
begin real x, y, z, eps; ввод (x, y, eps);
  for z := x - y while z > eps do
    begin z := (x + y)/2; y := sqrt (x * y); x := z end;
  вывод (y)
end
```

В общем случае, когда список цикла состоит из нескольких элементов, параметр цикла должен принимать все значения, предписываемые каждым из элементов. Так, например, программа

```
begin real s; integer i, n; ввод (n); s := 0;
  for i := 1, i + 1 while i ≤ n do
    s := s + 1/i;
  вывод (s)
end
```

также может служить для вычисления суммы $1 + 1/2 + \dots + 1/n$

Заголовок

```
for i := 1, 2, 3 do
```

предписывает выполнить тело цикла для явно указанных в этом заголовке значений параметра i . Он записывается заметно проще эквивалентного ему заголовка

```
for i := 1 step 1 until 3 do
```

Для заголовка

```
for x := 1, 2, 5, 10 do
```

невозможно найти никакой разумный эквивалент с элементами вида арифметической прогрессии или итерационного вида.

Не будем приводить примеров более сложных заголовков, так как пользоваться ими настоятельно не рекомендуется.

Входить в тело цикла разрешается только через заголовок цикла. Иначе говоря, вне тела цикла нельзя написать оператор перехода

к метке, метящей это тело или один из его внутренних операторов (если тело — составной или условный оператор). Выйти из цикла можно, включив в его тело оператор перехода к метке, расположенной вне цикла. Пусть, например, требуется написать оператор, разрешающий переход к следующему оператору программы, если все элементы массива

array a [$1:n$]

положительны, и отсылающий к метке l в противном случае. Этот оператор может быть таким:

**for $i := 1$ step 1 until n do
if $a[i] \leq 0$ then go to l**

Здесь выполнение цикла прекращается, как только обнаруживается первый неположительный элемент массива a . В подобных случаях считается, что параметр цикла сохраняет то значение, которое он имел в момент перехода. Если же цикл завершился в результате исчерпания всех значений, предписанных заголовком, то значение параметра не определено.

Иногда бывает необходимо выйти из составного оператора или блока, образующего тело цикла, не выходя из самого цикла. Для этого можно воспользоваться так называемым *пустым оператором*. Пустой оператор в записи программы никак не обозначается и его выполнение не влечет никаких действий, но он может быть помечен. Метка, метящая пустой оператор в конце составного оператора или блока, располагается непосредственно перед символом **end**, ограничивающим этот оператор. Переход к такой метке влечет за собой завершение выполнения оператора.

Применение пустого оператора можно продемонстрировать на следующем примере. Пусть требуется вычислить сумму положительных элементов массива

array a [$1:n$]

Для этого следует просмотреть все элементы массива и, если окажется, что очередной элемент неположителен, то ничего с ним не делать, в противном случае прибавить его к ранее вычисленной сумме. Описание этого процесса на АЛГОЛе может быть таким:

**$s := 0;$
for $i := 1$ step 1 until n do
begin if $a[i] \leq 0$ then go to $l;$
 $s := s + a[i];$**

$l: \text{end}$

Здесь меткой l помечен пустой оператор. Но можно было бы (и следовало бы) воспользоваться противоположным условием

```

s := 0;
for i := 1 step 1 until n do
    if a[i] > 0 then s := s + a[i]

```

```

for  $i := 1$  step 1 until  $n$  do
  for  $j := 1$  step 1 until  $n$  do
     $e[i, j] :=$  if  $j = i$  then 1 else 0

```

if B then S_1 else S

§ 11. Пример составления программы

Задача заключается в решении системы линейных алгебраических уравнений методом Гаусса. Напомним, в чем состоит этот метод. Пусть дана система

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

Процесс ее решения состоит из двух последовательных этапов, называемых прямым и обратным ходом. На прямом ходе система

приводится к виду

$$\begin{array}{rcl} x_1 + a_{12}'x_2 + \dots + a_{1n}'x_n & = & b_1', \\ x_2 + \dots + a_{2n}'x_n & = & b_2', \\ \cdot & & \cdot \\ \cdot & & \cdot \\ x_n & = & b_n'. \end{array} \quad (2)$$

На обратном ходе вычисляются значения неизвестных, начиная с x_n и кончая x_1 .

Не вдаваясь пока в детали выполнения прямого и обратного ходов, мы можем записать общую схему программы следующим образом:

begin описание данных;
ввод исходных данных;
прямой ход;
обратный ход;
вывод результатов

end

Здесь использованы псевдооператоры и псевдоописания, которых нет в АЛГОЛе. Однако роль и взаимосвязь этих частей программы вполне понятны программисту (с учетом сделанных выше пояснений о методе Гаусса). Псевдооператоры *ввод исходных данных* и *вывод результатов*, а также псевдоописание *описание данных* составляют более или менее стандартную часть любой программы, и их расшифровкой мы займемся в последнюю очередь. Сейчас же нам предстоит детализировать псевдооператоры *прямой ход* и *обратный ход*. Начнем с последнего как с более простого.

Заметим, что значение x_n , равное b_n , определяется уже в результате прямого хода. Поэтому структура псевдооператора *обратный ход* должна быть такой:

обратный ход:
for $i := n - 1$ step -1 until 1 do
 вычисление i -го неизвестного

Здесь идентификатор *обратный ход* стоит уже в качестве метки, используемой не по своему прямому назначению — быть адресом перехода, а как пояснение к оператору, который она метит. Этот оператор имеет форму оператора цикла, тело которого образует псевдооператор *вычисление i-го неизвестного*. К моменту выполнения этого псевдооператора для текущего значения i неизвестные $x_{i+1}, \dots, x_n$ уже вычислены, так что окончательная расшифровка псевдооператора в терминах «чистого» АЛГОЛа делается без труда:

```

вычисление  $i$ -го неизвестного:
begin  $x[i] := b1[i];$ 
  for  $k := i+1$  step 1 until  $n$  do
     $x[i] := x[i] - a1[i, k] \times x[k];$ 
end

```


При выполнении псевдооператора *исключение i -го неизвестного i -е уравнение* должно быть образовано не так, как уравнения с $(i+1)$ -го по n -е, тогда как последние уравнения образуются однотипно. Поэтому мы запишем этот псевдооператор в виде

```

    исключение  $i$ -го неизвестного:
    begin формирование  $i$ -го уравнения;
        for  $k := i+1$  step 1 until  $n$  do
            исключение  $i$ -го неизвестного из  $k$ -го уравнения
        end
    end
  
```

Здесь возникли два псевдооператора, требующие дальнейший детализации.

Проще всего было бы сформировать i -е уравнение из i -го же уравнения предыдущей системы, разделив все его коэффициенты на $a_{ii}^{(i-1)}$. Но этот коэффициент может оказаться равным нулю. Даже если он не равен, а близок к нулю, то при делении на него частное может оказаться настолько большим, что его нельзя будет представить в машине. Чтобы уменьшить эту опасность, будем формировать i -е уравнение с помощью того из уравнений системы (3), у которого коэффициент при x_i наибольший. Если номер s этого уравнения не совпадает с i , то, формируя i -е уравнение новой системы (4), мы должны переставить i -е уравнение системы (3) на место s -го уравнения. Это приводит нас к схеме:

```

    формирование  $i$ -го уравнения:
    begin поиск номера  $s$  наибольшего коэффициента в  $i$ -м столбце;
        if  $s = i$  then
            формирование  $i$ -го уравнения на своем месте
        else
            формирование  $i$ -го уравнения из  $s$ -го
        end
    end
  
```

Входящие сюда псевдооператоры уже легко выражаются в терминах операторов АЛГОЛа:

```

    поиск номера  $s$  наибольшего коэффициента в  $i$ -м столбце:
    begin  $s := i$ ;  $r := a[i, i]$ ;
        for  $k := i+1$  step 1 until  $n$  do
            if  $a[k, i] > abs(r)$  then
                begin  $s := k$ ;  $r := a[k, i]$  end
            end
        end
    end
  
```

```

    формирование  $i$ -го уравнения на своем месте:
    begin for  $j := i+1$  step 1 until  $n$  do
         $a[i, j] := a[i, j]/r$ ;
         $b[i] := b[i]/r$ 
    end
  
```

формирование i -го уравнения из s -го:

```
begin a[s, i] := a[i, i];
  for j := i + 1 step 1 until n do
    begin u := a[s, j]/r; a[s, j] := a[i, j]; a[i, j] := u end;
    u := b[s]/r; b[s] := b[i]; b[i] := u
  end
```

Столь же просто детализируется оставшийся псевдооператор прямого хода:

исключение i -го неизвестного из k -го уравнения:

```
begin for j := i + 1 step 1 until n do
  a[k, j] := a[k, j] - a[k, i] × a[i, j];
  b[k] := b[k] - a[k, i] × b[i]
end
```

Прежде чем начать детализацию оставшихся псевдооператоров и псевдоописания, полезно критически взглянуть на результаты проделанной работы. Сейчас в программе фигурируют три массива: массив коэффициентов a , свободных членов b и неизвестных x . Массивы эти появились не в результате сознательно принятого решения, а просто потому, что в записи исходной системы (1) присутствовали соответствующие обозначения. Мы уже видели, что нет смысла различать обозначения a и a' , а также b и b' , так как в результате выполнения прямого хода коэффициенты и свободные члены системы (2) вычисляются на том же месте, где располагались коэффициенты и свободные члены исходной системы. Нельзя ли упростить программу, совмещая и другие массивы? Легко видеть, что массив неизвестных x , вычисляемый на обратном ходе, целесообразно совместить с массивом свободных членов b . Это позволит обойтись без присваивания $x[i] := b[i]$ в псевдооператоре *вычисление i -го неизвестного*. Бросается в глаза также, что над столбцом свободных членов во многих псевдооператорах выполняются в точности те же операции, что и над столбцами матрицы коэффициентов. Ясно, что присоединение массива b свободных членов к массиву a в качестве $(n + 1)$ -го столбца позволит упростить запись этих псевдооператоров. Нетрудно также догадаться, что массив a можно нарастить не одним, а сразу несколькими столбцами свободных членов. Это позволит одновременно решать несколько систем линейных уравнений с одной и той же матрицей коэффициентов, но с разными свободными членами. Пусть расширенная матрица коэффициентов имеет размеры $n \times m$ (т. е. одновременно решается $m - n$ систем). Тогда, например, псевдооператор *исключение i -го неизвестного из k -го уравнения* примет вид

```
for j := i + 1 step 1 until m do
  a[k, j] := a[k, j] - a[k, i] × a[i, j]
```

Аналогично изменятся и другие псевдооператоры.

Зададимся теперь вопросом, оправдано ли выделение псевдооператора *формирование i -го уравнения на своем месте* при $s = i$ в особый случай? Этот псевдооператор, конечно, выполняется быстрее, чем псевдооператор *формирование i -го уравнения из s -го*, но выполнять его приходится лишь в одном случае из $n - i + 1$ (так как s может принимать любое из значений $i, i + 1, \dots, n$), так что программа ускоряется незначительно, а удлиняется заметно. Результат же работы этих двух псевдооператоров при $s = i$ один и тот же. Поэтому заменим условный оператор с условием **if $s = i$ then** одним псевдооператором *формирование i -го уравнения из s -го*.

Взглянем теперь, во что вылился псевдооператор

исключение i -го неизвестного:

begin *формирование i -го уравнения*:

begin *поиск номера s наибольшего коэффициента в i -м столбце*;

формирование i -го уравнения из s -го:

$a[s, i] := a[i, i]$;

for $j := i + 1$ **step** 1 **until** m **do**

begin $u := a[s, j]/r$; $a[s, j] := a[i, j]$;

$a[i, j] := u$ **end**

end;

for $k := i + 1$ **step** 1 **until** n **do**

исключение i -го неизвестного из k -го уравнения:

for $j := i + 1$ **step** 1 **until** m **do**

$a[k, j] := a[k, j] - a[k, i] \times a[i, j]$

end

Обращает на себя внимание то, что в этом псевдооператоре содержатся два цикла с параметром j с одним и тем же заголовком. Нельзя ли их объединить и тем самым укоротить программу и ускорить ее выполнение? Если это и возможно, то придется существенно изменить схему исключения i -го неизвестного, так как циклы, которые мы хотим объединить, находятся в разных псевдооператорах. Поскольку у нас уже есть опыт, сделаем это быстрее. Преобразованный псевдооператор примет следующий вид

исключение i -го неизвестного:

begin *поиск номера s наибольшего коэффициента r в i -м столбце*;

$a[s, i] := a[i, i]$;

for $j := i + 1$ **step** 1 **until** m **do**

формирование j -го столбца расширенной матрицы:

begin *формирование элемента i -й строки*:

$u := a[s, j]/r$; $a[s, j] := a[i, j]$; $a[i, j] := u$;

for $k := i + 1$ **step** 1 **until** n **do**

пересчет элемента k -й строки:

$a[k, j] := a[k, j] - a[k, i] \times u$

end

end

Детализация остальных частей программы настолько проста, что мы совместим ее со сборкой программы в одно целое. Готовая программа имеет следующий вид:

```

begin integer n, m; ввод (n, m);
  begin integer i, j, k, s; real r, u; array a [1:n, 1:m];
    ввод начальных данных: ввод (a);
    прямой ход:
    for i := 1 step 1 until n do
      исключение i-го неизвестного:
      begin поиск номера s наибольшего коэффициента r
        в i-м столбце:
        s := i; r := a[i, i];
        for k := i+1 step 1 until n do
          if abs(a[k, i]) > abs(r) then
            begin s := k; r := a[k, i] end;
        a[s, i] := a[i, i];
        for j := i+1 step 1 until m do
          формирование j-го столбца расширенной матрицы:
          begin формирование элемента i-й строки:
            u := a[s, j]/r; a[s, j] := a[i, j];
            a[i, j] := u;
            for k := i+1 step 1 until n do
              пересчет элемента k-й строки:
              a[k, j] := a[k, j] - a[k, i] × u
            end
          end;
        обратный ход:
        for i := n-1 step -1 until 1 do
          вычисление i-х неизвестных:
          for j := n+1 step 1 until m do
            for k := i+1 step 1 until n do
              a[i, j] := a[i, j] - a[i, k] × a[k, j];
          вывод результатов:
          for j := n+1 step 1 until m do
            for i := 1 step 1 until n do
              вывод (a[i, j])
            end
          end
        end
      end
    end
  end
end

```

§ 12. Процедуры

Имеющийся в АЛГОЛе аппарат процедур предназначен главным образом для того, чтобы отдельные части программ можно было описывать более или менее независимо от программы в целом.

Впоследствии такая часть программы может быть многократно использована в одной или в разных программах.

Процедура задается *описанием процедуры*. Его главной составной частью является оператор, называемый *телом процедуры*. Некоторые идентификаторы, встречающиеся в теле процедуры, могут быть объявлены *формальными параметрами*. Это значит, что при выполнении процедуры (точнее, ее тела) эти идентификаторы заменяются другими идентификаторами или выражениями, или же им присваиваются некоторые начальные значения, не указанные в самой процедуре. Чтобы заставить процедуру выполниться, в программе должно содержаться *обращение* к ней, представляющее собой либо *оператор процедуры*, либо *указатель функции*. В обращении к процедуре задаются *фактические параметры* этого обращения — те идентификаторы и выражения, которые либо подставляются вместо формальных параметров, либо задают их начальные значения.

Прежде чем описать все это более подробно, покажем на примере, как самостоятельная программа может быть превращена в описание процедуры и тем самым подготовлена к включению в состав другой программы. Возьмем программу, уже неоднократно использованную во многих примерах, — программу вычисления суммы $1 + 1/2 + \dots + 1/n$. Один из вариантов этой программы был таким

```
begin real s; integer i, n; ввод (n); s := 0;
  for i := 1 step 1 until n do
    s := s + 1/i;
  вывод (s)
end
```

Чтобы превратить эту программу в описание процедуры, выбрасываем из нее операторы ввода и вывода, так как связь процедуры с программой, в которую она включена, осуществляется через параметры, а не через аппарат ввода-вывода (хотя возможны процедуры, в которых этот аппарат используется, как и в самостоятельных программах, для связи с внешней средой). В программе остаются три идентификатора s , i и n , и нужно решить, какие из них должны быть включены в число формальных параметров процедуры. Очевидно, что переменная i будет нужна только самой процедуре для ее работы. Ее начальное значение вырабатывается самой процедурой. Ее конечное значение не представляет интереса после выполнения процедуры, к тому же оно не будет определено после выхода из цикла, в котором эта переменная выступает в роли параметра цикла. В ином положении находятся переменные n и s . Значение переменной n должно быть задано перед началом работы процедуры, так как оператор ввода этого значения мы решили выбросить из ее тела. Значение переменной s , выработанное в результате выполнения процедуры, является единственным результатом этого

выполнения и, разумеется, должно быть доступно программе, обратившейся к процедуре. В силу этих соображений обе эти переменные следует рассматривать как параметры процедуры. Описания этих переменных исключаются из программы в ходе ее преобразования в тело процедуры. Описание переменной i остается в теле процедуры. Таким образом, тело процедуры должно иметь следующий вид:

```
begin integer i; s := 0;
  for i := 1 step 1 until n do
    s := s + 1/i
  end
```

Чтобы задать перечень формальных параметров процедуры и указать их свойства, процедура снабжается *заголовком*. В заголовке процедуры ей дается также некоторое наименование — идентификатор. В рассматриваемом случае описание процедуры вместе с заголовком может быть таким:

```
procedure h (n, s); value n;
  integer n; real s;
begin integer i; s := 0;
  for i := 1 step 1 until n do
    s := s + 1/i
  end
```

Здесь h — идентификатор процедуры, n и s — ее формальные параметры. Заголовок содержит так называемые *спецификации* формальных параметров

integer n; real s;

В случае параметров, выступающих в роли простых переменных, их спецификации внешне не отличаются от описаний типа. Часть заголовка процедуры

value n;

говорит о различии между параметрами n и s . Параметр n , упомянутый в этой части (заметим, что *value* по-русски — значение), получает при обращении к процедуре значение соответствующего фактического параметра. Большого и не требуется, чтобы можно было начать выполнение тела процедуры. Параметр s , который не был упомянут в этой части заголовка, во время выполнения тела процедуры заменяется соответствующим фактическим параметром. В качестве этого фактического параметра должна быть задана некоторая переменная — та, которой мы хотим присвоить значение суммы $1 + 1/2 + \dots + 1/n$, вычисляемой в процессе выполнения процедуры. Как мы увидим из дальнейшего, единственный способ сделать это значение доступным программе, обратившейся

к процедуре, в том и состоит, чтобы указать, какой переменной мы хотим его присвоить, и заменить этой переменной параметр *s* на время выполнения тела процедуры при данном обращении.

В общем случае *описание процедуры* состоит из *заголовка процедуры* и *тела процедуры*, выписанных одно за другим. Тело процедуры — это произвольный оператор, чаще всего — блок. Заголовок состоит из выписанных подряд символа **procedure** (процедура), идентификатора процедуры, совокупности формальных параметров, списка значений и совокупности спецификаций. Лишь первые два из названных пяти элементов заголовка процедуры обязательны, а совокупность формальных параметров, список значений и спецификации могут отсутствовать. Идентификатор процедуры (в нашем примере — *h*) может быть любым. Но поскольку описание процедуры локализует этот идентификатор в блоке, в начале которого оно помещено, он должен отличаться от всех других локальных идентификаторов этого блока.

Совокупность формальных параметров состоит из списка формальных параметров, заключенного в круглые скобки. После совокупности формальных параметров (а при ее отсутствии — после идентификатора процедуры) ставится точка с запятой. *Формальные параметры* — это попарно различные идентификаторы. В списке они разделяются запятыми. Глобальный идентификатор, встречающийся в теле процедуры, считается формальным параметром лишь в том случае, когда он включен в список формальных параметров.

Список значений состоит из символа **value**, списка идентификаторов из числа формальных параметров и точки с запятой. Этот список разделяет все формальные параметры на два класса. К первому классу принадлежат формальные параметры, включенные в этот список. Про них говорят, что они *вызываются значением*. Это означает, что при обращении к процедуре этим формальным параметрам присваиваются значения соответствующих фактических параметров. Значениями в АЛГОЛе могут обладать только переменные, идентификаторы массивов (в этом случае значением считается совокупность значений элементов массива) и идентификаторы меток. Поэтому формальный параметр может быть включен в список значений лишь в том случае, если в теле процедуры он употребляется в качестве идентификатора переменной, массива или метки. Величина, представленная таким формальным параметром, ведет себя после присваивания ей начального значения как величина, введенная обычным способом. В нашем примере такой величиной является простая переменная, представленная в теле процедуры идентификатором *n*.

Формальные параметры, не включенные в список значений, попадают в класс параметров, *вызываемых по наименованию*. Это значит, что при обращении к процедуре такие параметры всюду, где они встречаются в теле процедуры, заменяются соответствующими

им фактическими параметрами. По наименованию в принципе может вызываться любой формальный параметр, какова бы ни была его роль в процедуре. Однако чтобы замена формального параметра фактическим не приводила к бессмыслице, фактический параметр должен удовлетворять некоторым требованиям. Так, в нашем примере по наименованию вызывается формальный параметр *s*. Этот параметр в теле процедуры дважды встречается в левой части оператора присваивания. Ясно, что соответствующий ему фактический параметр может быть только переменной (простой или с индексами), причем арифметической переменной, так как ей при выполнении процедуры будут присваиваться арифметические значения. В АЛГОЛе действует требование, чтобы любое обращение к процедуре приводило к осмысленному результату — чтобы формальным параметрам, вызываемым значением, присваивались при обращении значения, которые могут быть им присвоены, а формальные параметры, вызываемые по наименованию, заменялись такими фактическими параметрами, чтобы тело процедуры превращалось после этой замены в полноценный оператор, удовлетворяющий всем правилам АЛГОЛа.

Спецификации, включаемые в заголовок процедуры, отчасти способствуют формулировке этих ограничений на фактические параметры. В спецификациях указывается класс величин *), представленных формальными параметрами в теле процедуры, и тип их значений, если эти величины могут обладать значениями. *Совокупность спецификаций* состоит из одной или нескольких спецификаций, выписанных подряд. Каждая *спецификация* состоит из спецификатора, списка идентификаторов из числа формальных параметров данной процедуры и точки с запятой.

Возможны следующие *спецификаторы*:

integer
real
Boolean
integer array
real array (или просто **array**)
Boolean array
label (**label** — метка)
procedure
integer procedure
real procedure
Boolean procedure
switch (**switch** — переключатель)
string (**string** — строка)

*) Величины в АЛГОЛе делятся на следующие *классы*: переменные, массивы, метки, процедуры и переключатели (по поводу последних см. § 16).

Однако из этих спецификаторов только первые три, да и то лишь для формальных параметров, включенных в список значений, полностью определяют ограничения на соответствующие фактические параметры. В остальных случаях спецификации дают гораздо меньше информации о формальных параметрах, чем, скажем, описания массивов или процедур. Поэтому программист, включая обращение к некоторой процедуре (а следовательно, и описание этой процедуры) в свою программу, должен хорошо разобрат~~ься~~ в работе этой процедуры и в требованиях, которые она предъявляет к фактическим параметрам. При этом условии в программе можно использовать процедуры, составленные другими программистами для других задач. Эта возможность — одна из наиболее привлекательных черт аппарата процедур.

Оператор процедуры, с помощью которого производится обращение к процедуре, состоит из идентификатора этой процедуры и списка фактических параметров, заключенного в круглые скобки. Однако, если в описании процедуры нет формальных параметров, то оператор процедуры состоит из одного лишь идентификатора процедуры.

Фактические параметры — это выражения или идентификаторы. Их число должно совпадать с числом формальных параметров в описании процедуры. Это дает возможность установить взаимно однозначное соответствие между фактическими и формальными параметрами по их номерам в списках.

Выполняется оператор процедуры следующим образом. Мысленно создается блок, охватывающий тело процедуры. В этом *воображаемом блоке* локализуются идентификаторы всех формальных параметров, включенных в список значений. Затем осуществляется вызов параметров в соответствии с их классом, как это было описано выше.

Подробнее, каждому из формальных параметров, вызываемых значением, присваивается значение соответствующего фактического параметра. Тем самым для каждого формального параметра — массива (т. е. параметра, специфицированного как массив и используемого в теле процедуры в качестве идентификатора массива) определяются границы его индексов — они заимствуются у соответствующего фактического параметра. Каждый формальный параметр, не включенный в список значений, заменяется в теле процедуры соответствующим фактическим параметром, причем если этот фактический параметр — выражение, содержащее знаки операций вне скобок, то он ~~зак~~ключается в скобки. Построенный таким образом блок выполняется, и если в ходе выполнения не произошел переход к какому-либо оператору вне тела процедуры, то после этого начинает выполняться тот оператор, который должен выполняться после оператора процедуры.

§ 13. Примеры процедур

Пусть для $m = 1, 2, 4, \dots, 1024$ требуется отпечатать таблицу значений суммы $1 + 1/2 + \dots + 1/m$. Для этого можно воспользоваться следующей программой, в которую включена процедура h предыдущего примера:

```
begin integer m; real x;
  procedure h (n, s); value n;
    integer n; real s;
    begin integer i; s := 0;
      for i := 1 step 1 until n do
        s := s + 1/i
      end;
      for m := 1, 2 × m, while m ≤ 1024 do
        begin h (m, x); вывод (m, x) end
      end
    end
```

Блок, который следует мысленно построить во время выполнения оператора процедуры $h(m, x)$, таков:

```
begin integer n; n := m;
  begin integer i; x := 0;
    for i := 1 step 1 until n do
      x := x + 1/i
    end
  end
end
```

Он содержит в себе описание величины, представленной формальным параметром n — единственным параметром процедуры h , включенным в список значений. За этим описанием следует оператор присваивания этой величине значения соответствующего фактического параметра m и тело процедуры, в котором формальный параметр s , не включенный в список значений, заменен фактическим параметром x . В результате выполнения такого блока переменная x получит требуемое значение, которое будет отпечатано вместе со значением m при выполнении оператора *вывод* (m, x).

Можно ли было в основной программе (так обычно называется программа, содержащая описание процедуры и обращение к ней) вместо идентификаторов m и x воспользоваться идентификаторами h, n, s или i , занятыми в описании процедуры? В отношении идентификатора процедуры h ответ ясен — он должен отличаться от всех других идентификаторов, локальных в основном блоке, и следовательно, второй раз использован быть не может. Что касается идентификаторов формальных параметров n и s и идентификатора i , локального в теле процедуры, то ими в основной программе можно пользоваться свободно — можно, например, заменить m на n ,

а x на i , т. е. записать программу в виде

```
begin integer n; real i;
  procedure h (n, s); ...;
  for n := 1, 2 × n while n ≤ 1024 do
    begin h (n, i); вывод (n, i) end
  end
```

где описание процедуры h остается прежним. Прямолинейное применение правил построения воображаемого блока привело бы к блоку, содержащему бессмысленные конструкции:

```
begin integer n; n := n;
  . . . . .
  i := i + 1/i...
end
```

Однако следует руководствоваться принципом локализации идентификаторов в блоках, из которого вытекает, что идентификаторы n и i , локальные, соответственно, в воображаемом блоке и в преобразованном теле процедуры, обозначают там другие величины, не те, которые обозначены теми же идентификаторами в основном блоке. Поэтому при построении воображаемого блока следует эти локальные идентификаторы заменить (также мысленно) другими идентификаторами, например, $n1$ и $i1$, т. е. считать, что этот воображаемый блок имеет вид

```
begin integer n1; n1 := n;
  begin integer i1; i1 := 0;
    for i1 := 1 step 1 until n1 do
      i := i + 1/i1
    end
  end
end
```

Этот блок уже вполне осмыслен и его выполнение приводит к желаемым результатам.

В качестве следующего примера попробуем составить процедуру, заменяющую одно из самых употребительных математических обозначений — символ суммирования

$$s = \sum_{i=m}^n u_i$$

Глядя на эту запись, естественно предположить, что у процедуры должно быть пять параметров; переменная s , которой присваивается вычисленное значение суммы, переменная i , по которой ведется суммирование, значения m и n , определяющие границы изменения переменной i , и, наконец, выражение u_i общего члена суммы. Мы не зря, говоря о параметрах m и n , употребили

термин «значения», а применительно к параметру ui — термин «выражение», хотя каждый из этих формальных параметров может быть только идентификатором. Этим мы хотели подчеркнуть, что для параметров m и n важны только заданные им значения, а для параметра ui , напротив, важен характер зависимости соответствующего фактического параметра от переменной суммирования. Поэтому параметры m и n следует вызывать значением, а параметр ui — по наименованию.

Самый ответственный момент в этом примере — это решение, к какому классу отнести переменную i . Кроме уже сделанного предположения о том, что она должна быть параметром процедуры, вызываемым либо значением, либо по наименованию, следует рассмотреть еще две возможности — сделать ее локальной в теле процедуры или глобальной в теле, но локальной в основной программе.

Первый вариант — сделать переменную i параметром, вызываемым значением, — отвергается довольно легко. Действительно, переменная i станет при этом локальной величиной в воображаемом блоке, который создается при обращении к процедуре. В то же время фактический параметр, соответствующий параметру ui , должен, как мы уже установили, быть выражением, зависящим от переменной i . Но это выражение входит в состав оператора процедуры, который располагается в основном блоке — там, где локальная величина воображаемого блока недоступна. Поэтому у нас нет никакой возможности сделать выражение общего члена суммы зависящим от формального параметра i , если этот параметр вызывать значением. Чтобы сделать это еще более ясным, совершим все же попытку описать процедуру суммирования следующим образом:

```

procedure signal (s, i, m, n, ui); value i, m, n;
integer i, m, n; real s, ui;
begin s := 0;
      for i := m step 1 until n do
        s := s + ui
end

```

Пусть решается та же задача: отпечатать таблицу значений суммы $\sum_{i=1}^n \frac{1}{i}$ для $n = 1, 2, 4, \dots, 1024$. Если мы попробуем написать для решения этой задачи программу:

```

begin integer i, n; real s;
      procedure signal (s, i, m, n, ui); ...;
      for n := 1, 2 × n while n ≤ 1024 do
        begin signal (s, i, 1, n, 1/i); вывод (n, s) end
end

```

где описание процедуры *sigma1* имеет приведенный выше вид, то при выполнении оператора *sigma1* (*s*, *i*, *l*, *n*, *l/i*) будет построен воображаемый блок, в котором идентификаторы формальных параметров *i*, *m*, *n*, включенных в список значений, придется заменить какими-то другими, новыми идентификаторами:

```
begin integer i1, m1, n1; i1 := i; m1 := l; n1 := n;
  begin s := 0;
    for i1 := m1 step 1 until n1 do
      s := s + (l/i1)
    end
  end
end
```

Выполнение этого блока приведет к бессмысленному результату, так как локальная переменная внешнего блока *i* не получает в этой программе никакого конкретного значения.

К сходным результатам мы придем, если попробуем исключить переменную *i* из числа параметров и локализовать ее в теле процедуры:

```
procedure sigma2 (s, m, n, ui); value m, n;
integer m, n; real s, ui;
begin integer i; s := 0;
  for i := m step 1 until n do
    s := s + ui
  end
end
```

Здесь в воображаемом блоке описание переменной *i1* переместится еще глубже — в преобразованное тело процедуры, где по-прежнему в цикле с параметром *i1* будет выполняться оператор

$$s := s + (l/i)$$

Глобальная переменная *i* не имеет ничего общего с локальной переменной *i1*.

Положение выправится, если отказаться от локализации переменной *i* как в теле процедуры, так и в воображаемом блоке. Составим программу, содержащую процедуру *sigma3*, следующим образом:

```
begin integer i, n; real s;
  procedure sigma3 (s, m, n, ui); value m, n;
    begin s := 0;
      for i := m step 1 until n do s := s + ui
    end;
    for n := 1, 2 × n while n ≤ 1024 do
      begin sigma3 (s, 1, n, 1/i); вывод (n, s) end
    end
end
```

Теперь воображаемый блок будет таким:

```
begin integer m1, n1; m1 := 1; n1 := n;
  begin s := 0;
    for i := m1 step 1 until n1 do
      s := s + (1/i)
    end
  end
end
```

Цель достигнута, но довольно дорогой ценой. В программе появилась переменная i , внешне никак не связанная с процедурой σ , но по существу введенная только для того, чтобы обслуживать работу этой процедуры. Выражение для общего члена суммы должно зависеть именно от этой переменной, и никак иначе обозначить переменную суммирования нельзя. Не говоря о том, что это очень стеснительно для программистов, которые захотят воспользоваться так описанной процедурой σ , область применения этой процедуры становится довольно ограниченной, как мы вскоре увидим.

Наибольшую гибкость в применении процедуры суммирования можно обеспечить, если прибегнуть к оставшейся возможности — сделать переменную i параметром, вызываемым по наименованию. Во время выполнения тела процедуры эта переменная вообще исчезнет из него и заменится соответствующим фактическим параметром. От него и должно зависеть выражение для общего члена суммы — фактический параметр, соответствующий формальному параметру ui . Описание процедуры примет вид

```
procedure sigma (s, i, m, n, ui); value m, n;
integer i, m, n; real s, ui;
begin s := 0;
  for i := m step 1 until n do
    s := s + ui
  end
end
```

Программа решения рассматриваемой задачи может быть такой:

```
begin integer i, n; real s;
  procedure sigma (s, i, m, n, ui); ...;
  for n := 1, 2 × n while n ≤ 1024 do
    begin sigma (s, i, 1, n, 1/i); вывод (n, s) end
  end
```

При выполнении оператора процедуры $\sigma(s, i, 1, n, 1/i)$ создастся такой воображаемый блок:

```
begin integer m1, n1; m1 := 1; n1 := n;
  begin s := 0;
    for i := m1 step 1 until n1 do
      s := s + (1/i)
    end
  end
end
```

В этом блоке идентификаторы s и i глобальны. Следовательно, они обозначают те же величины, что и в основном блоке. Поэтому переменная s в результате выполнения воображаемого блока получит требуемое значение. Полезно заметить, что глобальная переменная i во время выполнения воображаемого блока будет пробегать последовательность значений $1, \dots, n$, а после выхода из цикла и из блока получит неопределенное значение. В рассматриваемом примере это не страшно, так как значение этой переменной вне процедуры не используется. В общем случае надо считаться с тем, что любая глобальная величина воображаемого блока может изменить или утратить свое значение в результате обращения к процедуре. Это явление называют *побочным эффектом* процедур.

Составим теперь процедуру решения систем линейных алгебраических уравнений методом Гаусса, взяв за основу программу, полученную в § 11. Мы уже видели, что в описаниях процедур формальными параметрами рекомендуется считать идентификаторы тех величин, которые в самостоятельных программах были бы объектами ввода и вывода. Этот принцип, хотя и не универсальный, в данном случае применим — в список формальных параметров процедуры следует включить идентификатор массива a — расширенной матрицы системы и идентификаторы переменных n и m , определяющих размер этой матрицы и границы изменения параметров ряда циклов в теле процедуры. По уже известным мотивам параметры m и n следует вызывать значением. Параметр a должен вызываться по наименованию, так как часть элементов этого массива представляет результат работы процедуры — решение системы, которое должно быть доступно после выхода из процедуры.

В результате преобразования программы и выбрасывания поясняющих меток получаем следующее описание процедуры:

```

procedure gauss (a, n, m); value n, m;
  array a; integer n, m;
begin integer i, j, k, s; real r, u;
  for i:=1 step 1 until n do
    begin s:=i; r:=a[i, i];
      for k:=i+1 step 1 until n do
        if abs(a[k, i]) > abs(r) then
          begin s:=k; r:=a[k, i] end;
      a[s, i]:=a[i, i];
      for j:=i+1 step 1 until m do
        begin u:=a[s, j]/r; a[s, j]:=a[i, j]; a[i, j]:=u;
          for k:=i+1 step 1 until n do
            a[k, j]:=a[k, j]-a[k, i]×u
          end
        end
      end
    end
  end;
end;
```

```

for  $i := n - 1$  step  $-1$  until  $1$  do
  for  $k := i + 1$  step  $1$  until  $n$  do
    begin  $u := a[i, k]$ ;
      for  $j := n + 1$  step  $1$  until  $m$  do
         $a[i, j] := a[i, j] - u \times a[k, j]$ 
      end
    end
  end
end

```

end

По сравнению с программой-прототипом в процедуре в операторе обратного хода изменен порядок выполнения циклов с параметрами k и j . Предоставляем читателю самому решить, в чем выгода этого изменения.

Пусть требуется решить две линейные алгебраические системы четвертого порядка, различающиеся лишь свободными членами, и одну систему седьмого порядка. Расширенные матрицы этих систем имеют соответственно размеры 4×6 и 7×8 . Предположим, что значения элементов этих матриц заготовлены на внешнем носителе. Тогда для решения задачи можно составить такую программу:

```

begin procedure gauss ( $a, n, m$ ); ...;
  array  $a[1:4, 1:6], b[1:7, 1:8]$ ; integer  $i$ ;
  ввод ( $a, b$ );
  gauss ( $a, 4, 6$ );
  вывод ( $a[1, 5], a[2, 5], a[3, 5], a[4, 5],$ 
     $a[1, 6], a[2, 6], a[3, 6], a[4, 6]$ );
  gauss ( $b, 7, 8$ );
  for  $i := 1$  step  $1$  until  $7$  do вывод ( $b[i, 8]$ )
end

```

end

В этой программе массивы a и b невелики, и можно не опасаться, что для них не хватит места в памяти. Но если бы для таких опасений имелись основания, то следующий вариант программы был бы предпочтительнее:

```

begin procedure gauss ( $a, n, m$ ); ...;
  integer  $n$ ;
  for  $n := 4, 7$  do
    begin integer  $m, i, j$ ;  $m :=$  if  $n = 4$  then  $6$  else  $8$ ;
      begin array  $a[1:n, 1:m]$ ;
        ввод ( $a$ ); gauss ( $a, n, m$ );
        for  $j := n + 1$  step  $1$  until  $m$  do
          for  $i := 1$  step  $1$  until  $n$  do
            вывод ( $a[i, j]$ )
          end
        end
      end
    end
  end
end

```

end

end

end

Здесь массив a описан в самом внутреннем блоке программы. Этот блок выполняется дважды — один раз с $n = 4$, $m = 6$, второй раз — с $n = 7$, $m = 8$. После первого выполнения блока массив, представленный локальным в этом блоке идентификатором a , перестает существовать. При повторном входе в этот блок создается новый массив с тем же идентификатором. Эти два массива никогда не используются одновременно. Это значит, что при переводе АЛГОЛ-программы на машинный язык можно размещать оба массива на одном и том же участке памяти — на участке, остающемся свободным после размещения машинной программы и локальных величин внешних блоков — переменных n , m , i и j . В первом варианте программы массивы a и b были одновременно описаны в одном и том же блоке, поэтому занятые ими участки памяти не имели права налегать один на другой.

Для последнего примера возьмем очень простую процедуру без параметров. Оператор такой процедуры, состоящий из одного лишь идентификатора процедуры, выполняется совсем просто — вместо него выполняется тело процедуры точно в таком виде, в каком оно содержится в описании процедуры.

Пусть в программе описаны переменные x , y , z и r , причем требуется, чтобы значение последней переменной было связано со значением трех других переменных соотношением

$$r = \sqrt{x^2 + y^2 + z^2}.$$

Можно было бы каждый раз вслед за операторами, присваивающими новые значения переменным x , y и z , помещать оператор

$$r := \text{sqrt}(x \uparrow 2 + y \uparrow 2 + z \uparrow 2)$$

Но проще (по крайней мере для программиста) сделать этот оператор телом процедуры без параметров:

$$\text{procedure norm; } r := \text{sqrt}(x \uparrow 2 + y \uparrow 2 + z \uparrow 2)$$

а в тех местах программы, где требуется вычислять новое значение r , помещать оператор этой процедуры

$$\text{norm}$$

Выполнение этого оператора приведет к тому, что переменная r получит требуемое значение.

§ 14. Вызов параметров

Вызовом параметров называются операции по подготовке к выполнению тела процедуры в соответствии с информацией о фактических параметрах, содержащейся в операторе процедуры. Как было сказано выше, вместо формальных параметров, включенных в список значений, создаются величины (простые переменные или

массивы), локальные в воображаемом блоке, непосредственно охватывающем тело процедуры во время его выполнения, и этим величинам присваиваются значения фактических параметров. На этом связь таких формальных параметров с фактическими прекращается. В частности, если формальным параметрам — переменным или элементам формальных параметров — массивов при выполнении тела процедуры присваиваются новые значения, то эти значения недоступны вне тела процедуры, так как они перестают существовать в момент, когда выполнение воображаемого блока завершается.

При вызове формальных параметров по наименованию эти формальные параметры исчезают из тела процедуры, вместо них появляются соответствующие фактические параметры. Если эти фактические параметры — переменные или массивы, то они могут в результате выполнения процедуры получить новые значения.

Отсюда вытекает важное следствие — если некоторый формальный параметр представляет результат (или один из результатов) работы процедуры, то такой параметр следует вызывать только по наименованию. Таковы параметр s процедуры h , параметр s процедуры σ и параметр a процедуры $gauss$.

Другой важный случай, требующий вызова параметров по наименованию, был продемонстрирован выше на примере процедуры σ . Это случай, когда в качестве фактического параметра требуется задать выражение, зависящее от некоторой переменной, заданной в качестве другого фактического параметра. При изменении значения переменной должно меняться и значение выражения, поэтому оно должно вычисляться многократно. Этого можно достичь, только если и выражение и используемое в нем обозначение переменной войдут в состав тела процедуры при его выполнении. Для этого формальные параметры, соответствующие обоим фактическим параметрам (в случае процедуры σ — параметры iii и i), должны вызываться по наименованию.

В прочих случаях для процедуры и для обращающейся к ней программы безразлично, вызывается ли формальный параметр, вступающий в тело процедуры в роли простой переменной или идентификатора массива, значением или наименованием. Здесь следует руководствоваться соображениями экономии. При переводе описания процедуры на машинный язык вызов значением формальных параметров — переменных реализуется, как правило, экономнее, чем вызов по наименованию. Для такого формального параметра выделяется всего одна ячейка, в которую помещается значение соответствующего фактического параметра, вычисляемое один раз — в момент обращения к процедуре. Для формальных параметров — массивов, наоборот, проще и экономнее организуется вызов наименованием. Для этого достаточно передать процедуре (точнее — реализующему ее участку машинной программы) информацию о месте расположения в памяти массива — фактического параметра

и о его размерах. При вызове массивов значением необходимо выделить участок памяти и разместить на нем копию массива — фактического параметра. Поэтому не рекомендуется без особой нужды вызывать параметры-массивы значением, а параметры-переменные — по наименованию.

§ 15. Функции

Функция — это процедура, основным результатом работы которой является вычисление одного числового или логического значения. Описание процедуры-функции отличается от описания обычной процедуры двумя особенностями. В заголовке процедуры перед символом **procedure** (т. е. в самом начале заголовка) помещается описатель типа значения функции **real**, **integer** или **Boolean**. В теле процедуры-функции должен содержаться оператор присваивания, в левой части которого стоит идентификатор этой процедуры, а в правой части — выражение, определяющее значение функции. Таких операторов может быть даже несколько, но выполняться должен только один из них. Заметим, что такой оператор выполняет лишь символическое присваивание. Идентификатор процедуры-функции не обладает свойствами простой переменной, и присвоенное ему значение не может быть использовано в самом теле процедуры.

Обращение к процедуре-функции производится с помощью *указателя функции*. Он состоит, как и оператор процедуры, из идентификатора процедуры и списка фактических параметров, заключенного в скобки, или только из идентификатора процедуры, если процедура описана без формальных параметров. Однако, в отличие от оператора процедуры, указатель функции является выражением, а не оператором, и может стоять в программе только там, где допускается помещать выражения, например, в качестве операнда операции, правой части оператора присваивания, параметра оператора вывода и т. п.

Некоторые из приведенных выше примеров процедур легко могут быть переделаны в функции. Например, описание процедуры *h* (§ 12) следующим образом преобразуется в описание функции:

```
real procedure h1 (n); value n; integer n;
begin integer i; real s; s:=0;
    for i:=1 step 1 until n do
        s:=s+1/i;
    h1:=s
end
```

Формальный параметр *s* процедуры *h* здесь пришлось превратить в локальную переменную тела процедуры *h1* и добавить оператор

$h1 := s$, так как идентификатор $h1$ нельзя использовать в теле процедуры в качестве простой переменной.

Программа, которая печатает таблицу значений суммы $1 + 1/2 + \dots + 1/m$ для $m = 1, 2, 4, \dots, 1024$, может теперь быть составлена следующим образом:

```
begin integer m;
  real procedure h1 (n); value n; integer n;
  begin integer i; real s; s := 0;
    for i := 1 step 1 until n do s := s + 1/i;
    h1 := s end;
  for m := 1, 2 × m while m ≤ 1024 do
    вывод (m, h1 (m))
  end
```

Та же задача может быть решена и с помощью другой программы, в которой используется процедура-функция *fsigma*, описывающая процесс вычисления суммы в общем виде (для произвольного общего члена):

```
begin integer i, m;
  real procedure fsigma (i, m, n, ui); value m, n;
  integer i, m, n; real ui;
  begin real s; s := 0;
    for i := m step 1 until n do s := s + ui;
    fsigma := s
  end;
  for m := 1, 2 × m while m ≤ 1024 do
    вывод (m, fsigma (i, 1, m, 1/i))
  end
```

Заслуживает внимания следующий пример использования функции *fsigma*. Для вычисления значения квадратичной формы

$$y = \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j$$

с заданными коэффициентами $a[i, j]$ при известных значениях переменных $x[1], \dots, x[n]$ можно обратиться к функции *fsigma* с помощью указателя функции, составляющего правую часть следующего оператора присваивания:

$$y := fsigma(i, 1, n, fsigma(j, 1, n, a[i, j] \times x[i] \times x[j]))$$

Параметр *ui* процедуры *fsigma* вызывается по наименованию. Поэтому при вычислении указателя функции в преобразованном теле процедуры *fsigma* возникнет обращение к ней самой (в операторе $s := s + fsigma(j, 1, n, a[i, j] \times x[i] \times x[j])$). Такие обра-

щения называются *рекурсивными*. Как видно уже из этого простого примера, рекурсивные обращения позволяют сравнительно компактно описывать довольно сложные вычисления.

Естественно, что процедура *gauss*, результатом которой является не одно числовое значение, а целый массив, не может быть описана в виде процедуры-функции. А в условиях, для которых была составлена процедура *norm*, гораздо естественнее заменить описание переменной *r* описанием функции:

real procedure *r*; *r* := *sqrt* ($x \uparrow 2 + y \uparrow 2 + z \uparrow 2$)

Идентификатор *r* за пределами этого описания становится указателем функции. Значение указателя функции *r* совпадает со значением выражения

sqrt ($x \uparrow 2 + y \uparrow 2 + z \uparrow 2$)

Заметим, кстати, что последнее выражение также представляет собой указатель функции (стандартной) с идентификатором *sqrt*. Стандартные функции (см. § 8.2) можно использовать в программах, не давая для них никаких описаний процедур.

§ 16. Переключатели

Рассмотрим следующее описание процедуры:

```

procedure s (i); value i; integer i;
    if i = 1 then go to D1
    else if i = 2 then go to D2
    . . . . .
    else if i = N then go to DN
  
```

где *N* — некоторое натуральное число, а *D*₁, *D*₂, ..., *D*_{*N*} — именуемые выражения. Пусть выполняется оператор процедуры *s* (*A*), где *A* — арифметическое выражение. Если значение выражения *A*, округленное до ближайшего целого числа *i*, заключено в пределах $1 \leq i \leq N$, то выполнение оператора *s* (*A*) приведет к переходу на метку, являющуюся значением именуемого выражения *D*_{*i*}. Если же $i < 1$ или $i > N$, то оператор процедуры сработает вхолостую (подобно пустому оператору).

В АЛГОЛе есть конструкция, называемая *описанием переключателя*, служащая в точности для тех же целей, что и приведенное выше описание процедуры *s*. Она имеет вид:

switch *s* := *D*₁, *D*₂, ..., *D*_{*N*}

Идентификатор *s*, описанный таким образом, становится в пределах

действия этого описания *идентификатором переключателя*. Список $D_1, \dots, D_N$ называется *переключательным списком*. Для обращения к переключателю вместо использованного в предыдущем абзаце оператора процедуры $s(A)$ служит оператор перехода

go to $s[A]$

выполнение которого приводит к описанным выше результатам. Выражение $s[A]$ называется *указателем переключателя*. Оно является одной из разновидностей именуемого выражения и может быть использовано не только непосредственно в операторе перехода, но и в качестве компоненты условного именуемого выражения, в переключательном списке, в качестве фактического параметра при обращении к процедуре. Во всех этих случаях значение указателя переключателя вычисляется одинаково.

Глава 9. КРАТНЫЕ ЦИКЛЫ

§ 1. Размещение в памяти многомерных массивов

При знакомстве с АЛГОЛОм мы встретились с понятием массива произвольной размерности. На практике редко приходится иметь дело с массивами размерности больше двух, но двумерные массивы — матрицы — встречаются сплошь и рядом. Кратные циклы, рассматриваемые в этом разделе, часто возникают при работе с матрицами.

Сам термин «многомерный массив» указывает, что человек не склонен приписывать элементам такого массива простую линейную упорядоченность, при которой каждый элемент получает свой отличительный порядковый номер. Однако, если мы хотим разместить массив в памяти, то его элементы неизбежно выстраиваются в такую линейную последовательность в соответствии с адресами, которые они получают. В принципе допустимо размещение элементов массива в памяти в любом порядке, лишь бы для каждого элемента была выделена своя ячейка. Но на практике многомерные массивы, как и одномерные, размещаются в памяти так, чтобы адрес элемента линейно зависел от его индексов.

Рассмотрим в качестве примера матрицу

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

Для ее размещения в памяти необходимо выделить mn ячеек. Ограничимся лишь такими вариантами размещения, когда эти ячейки занимают один сплошной участок памяти, т. е. составляют один программный массив. Линейная зависимость адреса от индексов получится, если на этом участке элементы матрицы будут расположены одним из двух способов: *по строкам*

$$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, \dots, a_{2n}, \dots, a_{m1}, \dots, a_{mn}$$

или *по столбцам*

$$a_{11}, a_{21}, \dots, a_{m1}, a_{12}, \dots, a_{m2}, \dots, a_{1n}, \dots, a_{mn}.$$

Размещение матрицы по строкам — самое употребительное, но, вероятно, лишь потому, что в таком порядке мы читаем печатный текст. Между тем, многие программы, например, программы решения систем линейных алгебраических уравнений несколько упрощаются, если фигурирующие в них матрицы расположить по столбцам.

Обозначим базовый адрес массива, в котором размещаются элементы матрицы, буквой a . Пусть адрес элемента a_{i1} равен $a + 1$. Подсчитаем адрес ячейки, в которой должен быть размещен элемент a_{ij} — j -й элемент i -й строки матрицы. Первые $i - 1$ строка занимают $(i - 1)n$ ячеек, поэтому элементы i -й строки начинают размещаться с ячейки с адресом $a + (i - 1)n + 1$, и следовательно, адрес элемента a_{ij} равен $a + (i - 1)n + j$. Аналогично, при размещении матрицы по столбцам элемент a_{ij} (теперь мы смотрим на него как на i -й элемент j -го столбца) попадает в ячейку с адресом $a + (j - 1)m + i$. Эти выражения для адреса элемента a_{ij} будут очень часто использоваться в дальнейшем, особенно первое, соответствующее размещению матрицы по строкам.

У п р а ж н е н и е 1. Для обоих способов размещения матрицы возможны еще по 3 варианта, возникающие, если изменить на обратный порядок строк (т. е. брать их снизу вверх), либо столбцов, либо тех и других. Написать выражения для адреса элемента a_{ij} для этих 6 вариантов размещения матрицы (для каждого варианта принять, что наименьший из адресов элементов равен $a + 1$).

Перейдем к более общему случаю. Пусть k -мерный массив описан в программе на АЛГОЛе следующим образом:

`array a [ $l_1 : u_1, \dots, l_k : u_k$ ],`

где l_j и u_j ($j = 1, \dots, k$) — некоторые целые числа. Размещению матрицы по строкам аналогично такое размещение этого массива, при котором вслед за элементом $a[i_1, \dots, i_k]$ размещается (при $i_k < u_k$) элемент $a[i_1, \dots, i_{k-1}, i_k + 1]$, а вслед за элементом

$a[i_1, \dots, i_j, u_{j+1}, \dots, u_k]$

при $i_j < u_j$ — элемент $a[i_1, \dots, i_{j-1}, i_j + 1, l_{j+1}, \dots, l_k]$. Таким образом, при движении вдоль программного массива с наибольшей частотой меняется самый правый индекс i_k , с наименьшей — самый левый i_1 (который вообще меняется монотонно от l_1 до u_1). Обозначим: $w_j = u_j - l_j + 1$ — число возможных значений индекса i_j или *размер* массива по j -му измерению для $j = 1, \dots, k$; $d_k = 1$; $d_j = w_{j+1}d_{j+1}$ для $j = k - 1, \dots, 1$. При изменении младшего индекса i_k на единицу адрес элемента также меняется на единицу. Когда младший индекс пробежит через все w_k своих возможных значений и снова примет значение l_k , увеличится на единицу индекс i_{k-1} , так что этому изменению отвечает приращение адреса $w_k = d_{k-1}$. Когда и индекс i_{k-1} пройдет один раз через весь набор своих возможных значений и вернется к значению l_{k-1} , адрес элемента возрастет на

$w_{k-1}d_{k-1} = d_{k-2}$. Это приращение соответствует увеличению на единицу индекса i_{k-2} . Так мы убеждаемся, что для любого j величина d_j равна приращению адреса элемента массива при увеличении на единицу индекса i_j (и неизменных значениях остальных индексов). Отсюда следует, что адрес элемента $a[i_1, \dots, i_k]$ при таком размещении массива равен

$$a + d_1 i_1 + \dots + d_k i_k + c,$$

где a — базовый адрес, c — константа, зависящая от выбора базового адреса. Чаще всего базовый адрес выбирают одним из следующих способов. Принимают, что a — это наименьший из адресов элементов массива, т. е. адрес элемента $a[l_1, \dots, l_k]$. Тогда $c = -d_1 l_1 - \dots - d_k l_k$ и адрес элемента $a[i_1, \dots, i_k]$ равен

$$a + d_1 (i_1 - l_1) + \dots + d_k (i_k - l_k).$$

Можно принять, что элемент $a[l_1, \dots, l_k]$ размещается в ячейке с адресом $a + 1$. Тогда адрес элемента $a[i_1, \dots, i_k]$ оказывается равным

$$a + d_1 (i_1 - l_1) + \dots + d_k (i_k - l_k) + 1.$$

Простейшее выражение для адреса мы получим, если примем, что a — это адрес элемента $a[0, \dots, 0]$, возможно, не существующего в рассматриваемом массиве. Тогда $c = 0$ и адрес элемента $a[i_1, \dots, i_k]$ равен

$$a + d_1 i_1 + \dots + d_k i_k.$$

У п р а ж н е н и е 2. Написать выражение для адреса элемента $a[i_1, \dots, i_k]$, если массив размещается в памяти «по столбцам» и a — это адрес элемента $a[l_1, \dots, l_k]$.

В АЛГОЛе все массивы — прямоугольные, то есть границы изменения любого индекса не зависят от значений других индексов. Если пользоваться менее ограничительными средствами записи алгоритмов, то в рассмотрение можно вводить массивы более сложной формы — треугольные, трапециевидные и т. п. Пусть, например, в вычислениях участвует симметричная матрица (a_{ij}) порядка n . Для экономии места достаточно хранить в памяти, например, только ее левый нижний угол, размещая его по строкам, т. е. в следующей последовательности:

$$a_{11}, a_{21}, a_{22}, a_{31}, a_{32}, a_{33}, a_{41}, \dots$$

Элементы строк с 1-й по $(i-1)$ -ю занимают в памяти $1 + 2 + \dots + (i-1) = i(i-1)/2$ ячеек. Если элемент a_{11} поместить в ячейку $a + 1$, то адрес элемента a_{ij} будет равен $a + i(i-1)/2 + j$ (при $j \leq i$).

У п р а ж н е н и я 3. Для трехдиагональной матрицы (т. е. такой, что $a_{ij} = 0$ при $|j - i| > 1$) незачем хранить в памяти заведомо нулевые элементы. Вычислить адрес элемента a_{ij} ($|j - i| \leq 1$), если место в памяти отводится лишь

для элементов главной диагонали и двух соседних с ней диагоналей, адрес элемента a_{11} равен $a + 3$ и элементы располагаются по строкам.

4. Прямоугольная матрица размером $n \times m$ ($n < m$) обладает свойством: $a_{ij} = a_{ji}$ при $i, j \leq n$. В памяти хранятся лишь элементы a_{ij} с $j \geq i$, расположенные по строкам. Найти адрес a_{ij} , если адрес a_{11} равен a .

5. Из кубического массива $a[0:n, 0:n, 0:n]$ вырезан «тетраэдрический» массив, индексы элементов $a[i, j, k]$ которого удовлетворяют неравенствам

$$j+k-i \geq 0, \quad i+k-j \geq 0, \quad i+j-k \geq 0, \quad i+j+k \leq 2n.$$

Эти элементы размещены в памяти «по строкам» и адрес элемента $a[0, 0, 0]$ равен a . Найти выражение для адреса элемента $a[i, j, k]$, принадлежащего вырезке.

§ 2. Программирование кратных циклов

Методы программирования кратных циклов в сущности ничем не отличаются от уже знакомых нам методов, используемых при программировании простых циклов. Как правило, в кратном цикле каждый внутренний цикл можно рассматривать как рабочую часть следующего, более обширного цикла. Кроме рабочей части, этот цикл должен содержать команды переадресации, проверки логического условия и формирования.

Рассмотрим примеры.

Пример 1. Составим программу вычисления произведения матрицы на вектор:

$$y_i = \sum_{j=1}^n a_{ij} x_j \quad (i = 1, 2, \dots, m).$$

Алгоритм вычисления можно записать на АЛГОЛе в следующем виде (опуская описания, а также операторы ввода и вывода):

```
for i := 1 step 1 until m do
  begin y[i] := 0;
        for j := 1 step 1 until n do
          y[i] := y[i] + a[i, j] × x[j]
        end
  end
```

Будем считать, что матрица (a_{ij}) расположена в памяти по строкам, так что адрес элемента a_{ij} равен $a + (i - 1)n + j$, составляющая x_j вектора x имеет адрес $x + j$, $j = 1, \dots, n$, а составляющая y_i вектора y должна быть заслана в ячейку с адресом $y + i$, $i = 1, \dots, m$.

Программирование начнем с рабочей части внутреннего цикла (в алгоритме ей соответствует оператор присваивания $y[i] := y[i] + a[i, j] \times x[j]$)

$$\begin{array}{ccccccc} p & 0 & \times & a + (i-1)n + j & x + j & r & a_{ij}x_j \\ p+1 & 0 & + & y + i & r & y + i & y_i := y_i + a_{ij}x_j \end{array}$$

Обе команды имеют переменные адреса: в команде p адреса зависят от параметров i и j , в команде $p + 1$ — только от i .

Завершим программирование внутреннего цикла (по параметру j). Для этого нужно добавить команду формирования для $j = 0$ и команду переадресации перед рабочей частью, соответствующую переходу от j к $j + 1$. Сравнение на окончание цикла будем проводить по значению команды p .

Получим следующую программу:

$p-2$	0	ПЧ	z_1	0	p	$j := 0$
$p-1$	0	$+M$	p	z_2	p	$j := j + 1$
p	0	$\times$	$a + (i-1)n + j$	$x + j$	r	$a_{ij}x_j$
$p+1$	0	$+$	$y + i$	r	$y + i$	$y_i := y_i + a_{ij}x_j$
$p+2$	0	$\neq$	p	z_3	0	Если $j \neq n$, то ($\omega := 0$)
$p+3$	0	П0	0	$p-1$	0	перейти на $p-1$

Значение использованных констант:

z_1	0	$\times$	$a + (i-1)n$	x	r
z_2	0	0	1	1	0
z_3	0	$\times$	$a + in$	$x + n$	r

В отличие от программ, реализующих простые циклы, значения заготовок z_1 и z_3 зависят от параметра i и поэтому являются переменными во внешнем цикле (по i). Естественно выделить для них две рабочие ячейки. В соответствии с этим было бы желательно изменить обозначения их адресов. В следующих примерах мы так и будем поступать, здесь же сохраним уже введенные обозначения.

Дальнейшая задача состоит в том, чтобы построить внешний цикл, обеспечивающий выполнение внутреннего цикла m раз для всех целых значений i от 1 до m . В этом внешнем цикле команды $p-2 \div p+3$ будут являться рабочей частью. Очевидно, к ним нужно присоединить команду, засылающую нуль в ячейку $y + i$:

$p-3$	0	ПЧ	0	0	$y + i$	$y_i := 0$

При изменении индекса i на 1 должна быть произведена переадресация всех команд, зависящих от i . Это команды с адресами $p-3$, $p+1$ и, на первый взгляд, p . Но при более внимательном изучении программы видно, что значение последней команды определяется значением заготовки z_1 , которая засылается командой $p-2$ в ячейку p . Значит, если мы хотим изменить вид команды p , нам нужно менять заготовку z_1 . Кроме того, от i зависит первый адрес заготовки z_3 . Следовательно, нам нужно осуществить переадресацию для $i := i + 1$ содержимого четырех ячеек ($p-3$, $p+1$, z_1 , z_3).

При записи константы, с помощью которой осуществляется переадресация заготовок z_1 и z_3 , мы столкнемся с небольшой проблемой. Содержимое первого адреса этих заготовок должно изменяться

на n единиц при изменении i на единицу. Однако записать соответствующую константу переадресации в виде

0 0 n 0 0

нельзя, так как в силу соглашения 4 (см. гл. 6) буква n , занимающая позицию адреса в команде или константе, заменяет адрес ячейки, где хранится значение величины n , а не само это значение. Можно воспользоваться соглашением 9 (§ 6 гл. 7), по которому буква n обозначает значение величины n , если она входит в состав выражения, занимающего позицию адреса, и записать первый адрес заготовки в виде $1 \times n$, $n + 0$ или просто $+n$. На последнем варианте мы и остановимся. Так как подобные обозначения адресов будут в дальнейшем встречаться очень часто, то мы закрепим их смысл в виде отдельного и последнего

Соглашения о записи символических адресов (продолжение; см. стр. 78, 81, 109)

10. Если в записи команды некоторый адрес заменен буквой с предшествующим ей знаком $+$, то во время выполнения программы этот адрес должен быть численно равен текущему значению переменной или постоянной величины, обозначенной в соответствующем алгоритме той же буквой.

Команды переадресации поместим после рабочей части цикла по i :

$$\left. \begin{array}{l} 0 \ +M \ p-3 \ z_1 \ p-3 \\ 0 \ +M \ z_1 \ z_5 \ z_1 \\ 0 \ +M \ z_3 \ z_5 \ z_3 \\ 0 \ +M \ p+1 \ z_6 \ p+1 \end{array} \right\} i := i + 1$$

Использованные константы: —

z_1	0	0	0	0	1
z_5	0	0	$+n$	0	0
z_6	0	0	1	0	1

Любой из переменных кодов в ячейках $p-3$, $p+1$, z_1 , z_3 может служить для сравнения на окончание цикла. Остановим свой выбор на содержимом ячейки $p-3$.

Тогда завершающие команды цикла будут иметь вид

$$\begin{array}{l} 0 \neq \ p-3 \ z_7 \ 0 \\ 0 \ p0 \ 0 \ p-3 \ 0 \end{array}$$

Так как переадресация в цикле по i происходит после выполнения соответствующих операций, то выход из цикла осуществляется при $i = m + 1$ и соответствующее значение должна иметь заготовка z_7 :

$$z_7 \quad 0 \ p4 \ 0 \ 0 \ y+m+1$$

Программа будет закончена, если ее дополнить командами формирования переменных кодов для $i = 1$:

$$\left. \begin{array}{l} 0 \text{ ПЧ } z_8 \quad 0 \quad p-3 \\ 0 \text{ ПЧ } z_9 \quad 0 \quad p+1 \\ 0 \text{ ПЧ } z_{10} \quad 0 \quad z_1 \\ 0 \text{ ПЧ } z_{11} \quad 0 \quad z_3 \end{array} \right\} i := 1$$

где заготовки имеют следующие значения:

$$\begin{array}{lclclcl} z_8 & 0 & \text{ПЧ} & 0 & 0 & y+1 \\ z_9 & 0 & + & y+1 & r & y+1 \\ z_{10} & 0 & \times & a & x & r \\ z_{11} & 0 & \times & a+n & x+n & r \end{array}$$

(формирование производится для $i = 1$, так как переадресация по i происходит в конце цикла по этому параметру).

Итак, мы получили следующую программу:

Программа 1

$$\begin{array}{lclclcl} 0 & \text{ПЧ} & z_8 & 0 & p-3 & \\ 0 & \text{ПЧ} & z_9 & 0 & p+1 & \\ 0 & \text{ПЧ} & z_{10} & 0 & z_1 & \\ 0 & \text{ПЧ} & z_{11} & 0 & z_3 & \\ p-3 & 0 & \text{ПЧ} & 0 & 0 & y+i \quad y_i := 0 \\ & 0 & \text{ПЧ} & z_1 & 0 & p \quad j := 0 \\ p-1 & 0 & +M & p & z_2 & p \quad j := j+1 \\ p & 0 & \times & a+(i-1) \times & x+j & r \quad a_{ij} x_j \\ & & & \times n+j & & \\ p+1 & 0 & + & y+i & r & y+i \quad y_i := y_i + a_{ij} x_j \\ & 0 & \neq & p & z_3 & 0 \quad \text{Если } j \neq n, \text{ то } (\omega := 0) \\ & 0 & \text{ПО} & 0 & p-1 & 0 \quad \text{перейти на } p-1 \\ & 0 & +M & p-3 & z_4 & p-3 \\ & 0 & +M & z_1 & z_5 & z_1 \\ & 0 & +M & z_3 & z_5 & z_3 \\ & 0 & +M & p+1 & z_6 & p+1 \\ & 0 & \neq & p-3 & z_7 & 0 \quad \left. \begin{array}{l} \text{Если } i \neq m+1, \\ \text{то } (\omega := 0) \end{array} \right\} i := i+1 \\ & 0 & \text{ПО} & 0 & p-3 & 0 \quad \text{перейти на } p-3 \end{array}$$

Константы

$$\begin{array}{lclcl} z_2 & 0 & 0 & 1 & 1 & 0 \\ z_4 & 0 & 0 & 0 & 0 & 1 \\ z_5 & 0 & 0 & +n & 0 & 0 \\ z_6 & 0 & 0 & 1 & 0 & 1 \\ z_7 & 0 & \text{ПЧ} & 0 & 0 & y+m+1 \\ z_8 & 0 & \text{ПЧ} & 0 & 0 & y+1 \\ z_9 & 0 & + & y+1 & r & y+1 \\ z_{10} & 0 & \times & a & x & r \\ z_{11} & 0 & \times & a+n & x+n & r \end{array}$$

Общий вид содержимого рабочих ячеек z_1 и z_3 :

$$\begin{array}{lcl} z_1 & 0 & \times \quad a+(i-1)n \quad x \quad r \\ z_3 & 0 & \times \quad a+in \quad x+n \quad r \end{array}$$

Эта программа составлена по простым рецептам и, разумеется, решает поставленную задачу. Но в ней далеко не использованы все резервы для экономии времени работы машины.

Попробуем улучшить эту программу. Для этого познакомимся с новым приемом программирования — *восстановлением* команд.

Предположим, что в процессе выполнения программы встречается простой цикл

for $i := 1$ step 1 until n do S

где S — некоторый оператор, содержащий переменную команду p вида:

$$p \quad 0 \text{ КОП } a_1 + b_1 i \quad a_2 + b_2 i \quad a_3 + b_3 i$$

Если команда переадресации стоит перед командой p , то во время работы цикла содержимое ячейки p меняется от

$$0 \text{ КОП } a_1 \quad a_2 \quad a_3 \tag{1}$$

до

$$0 \text{ КОП } a_1 + b_1 n \quad a_2 + b_2 n \quad a_3 + b_3 n \tag{2}$$

(от $i = 0$ до $i = n$). Если же переменная команда выполняется до переадресации, то ее вид меняется от значения

$$0 \text{ КОП } a_1 + b_1 \quad a_2 + b_2 \quad a_3 + b_3 \tag{3}$$

соответствующего $i = 1$, до значения

$$0 \text{ КОП } a_1 + b_1(n+1) \quad a_2 + b_2(n+1) \quad a_3 + b_3(n+1) \tag{4}$$

соответствующего $i = n + 1$.

В обоих случаях «приращение», которое получит переменная команда, равно:

$$0 \quad 0 \quad nb_1 \quad nb_2 \quad nb_3$$

Чтобы получить начальный вид команды — восстановить ее — достаточно вычесть из значения ее адресной части, с которым завершился цикл, это суммарное приращение. Сделать это можно, например, операцией вычитания мантисс ($-M$).

При использовании восстановления команд в ячейку, предназначенную для переменной команды, при составлении программы обязательно должно быть записано исходное значение этой команды. Возможны два варианта, в зависимости от того, что выполняется первым при работе программы — сам цикл или восстановление переменной команды. Операция восстановления преобразует переменную команду от вида (2) к виду (1) или от вида (4) к виду (3).

Поэтому, если по ходу основной программы первым выполняется цикл, а потом следует восстановление переменной команды, исходное значение последней нужно взять соответственно в виде (1) или (3). Если же восстановление предшествует первому выполнению цикла, то за исходное значение переменной команды нужно принять значение (2) или (4).

Поясним сказанное на простом примере.

Пусть нужно переслать содержимое группы n ячеек из одного места памяти в другое, т. е. реализовать оператор цикла:

for $i := 1$ step 1 until n do $b[i] := a[i]$

С использованием восстановления команд можно написать следующие два варианта программы:

1)	0	—M	p	z_3	p	$i := i - n$ (восстановление вместо формирования)
	p	0	ПЧ	$a + n + 1$	0	$b + n + 1$ $b_i := a_i$ (для $i = n + 1$)
		0	+M	p	z_1	p $i := i + 1$
		0	$\neq$	p	z_2	0 Если $i \neq n + 1$, то ($\omega := 0$)
		0	ПО	0	p	0 перейти на p
2)	p	0	ПЧ	$a + 1$	0	$b + 1$ $b_i := a_i$ (для $i = 1$)
		0	+M	p	z_1	p $i := i + 1$
		0	$\neq$	p	z_2	0 Если $i \neq n + 1$, то ($\omega := 0$)
		0	ПО	0	p	0 перейти на p
		0	—M	p	z_3	p Восстановление p

Значения констант в обоих вариантах совпадают:

z_1	0	0	1	0	1
z_2	0	ПЧ	$a + n + 1$	0	$b + n + 1$
z_3	0	0	+n	0	+n

Как видно, при программировании простых циклов прием восстановления никакого выигрыша не дает, а переменную команду мы должны записывать в менее наглядном виде. Поэтому в программах простых циклов лучше пользоваться формированием, а не восстановлением команд.

Посмотрим, что происходит при программировании кратных циклов.

Пусть в программе выполняется цикл

for $i := 1$ step 1 until m do
for $j := 1$ step 1 until n do S

где S — некоторый оператор, который содержит переменную команду p такого вида:

p 0 КОП $a_1 + b_1 i + c_1 j$ $a_2 + b_2 i + c_2 j$ $a_3 + b_3 i + c_3 j$,

Для восстановления этой команды по значению параметра j во внешнем цикле (по i) используется команда

$$p_1 \quad 0 \quad -M \quad p \quad z_B \quad p$$

где константа z_B должна иметь следующее значение:

$$z_B \quad 0 \quad 0 \quad c_1 n \quad c_2 n \quad c_3 n$$

При этом в команде p мы возвращаемся от значения $j = n$ к значению $j = 0$ (или от $j = n + 1$ к $j = 1$) при неизменном значении i . Но, кроме того, мы должны осуществить переадресацию по i , например, с помощью следующей команды:

$$p_2 \quad 0 \quad +M \quad p \quad z_{II} \quad p,$$

где

$$z_{II} \quad 0 \quad 0 \quad +b_1 \quad +b_2 \quad +b_3$$

Очевидно, эти две операции можно совместить, введя вместо констант z_B и z_{II} новую константу переадресации

$$z_{II+B} \quad 0 \quad 0 \quad b_1 - c_1 n \quad b_2 - c_2 n \quad b_3 - c_3 n$$

и заменив команды p_1 и p_2 одной:

$$0 \quad +M \quad p \quad z_{II+B} \quad p$$

То же самое можно сделать, воспользовавшись операцией

$$0 \quad -M \quad p \quad z'_{II+B} \quad p$$

Используемая в этом случае константа должна быть записана так:

$$z'_{II+B} \quad 0 \quad 0 \quad c_1 n - b_1 \quad c_2 n - b_2 \quad c_3 n - b_3$$

Оба варианта равноправны. Если числа, стоящие во всех трех адресах константы, имеют одинаковые знаки, лучше выбрать тот вариант, в котором эти числа положительны, так как такие константы записываются проще.

Таким образом, мы видим, что использование приема восстановления при программировании сложных циклов может сократить число команд и заготовок в программе.

Вернемся к примеру 1 и попробуем для внутреннего цикла воспользоваться восстановлением вместо формирования. Все команды цикла по j сохраняют свой вид:

$p-1$	0	$+M$	p	z_2	p	$j := j + 1$
p	0	$\times$	$a + (i-1)n + j$	$x + j$	r	$a_{ij}x_j$
$p+1$	0	$+$	$y + i$	r	$y + i$	$y_i := y_i + a_{ij}x_j$
	0	$\neq$	p	z_3	0	Если $j \neq n$, то ($\omega := 0$)
	0	PO	0	$p-1$	0	перейти на $p-1$

Значения констант z_2 и z_3 те же, что в программе 1:

$$\begin{array}{l} z_2 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \\ z_3 \quad 0 \quad \times \quad a+in \quad x+n \quad r \end{array}$$

В ячейке p перед началом внутреннего цикла должна находиться команда

$$p \quad 0 \quad \times \quad a+(i-1)n \quad x \quad r$$

В программе 1 эта команда каждый раз формировалась в результате засылки в ячейку p заготовки z_1 . Но то же самое можно осуществить, выполняя после выхода из цикла восстановление ячейки p с помощью операции

$$0 \quad -M \quad p \quad z_{12} \quad p$$

где

$$z_{12} \quad 0 \quad 0 \quad +n \quad +n \quad 0$$

При этом мы вернемся от значения $j = n$ к значению $j = 0$, не изменив значения i .

Во внешнем цикле, как и в программе 1, нужно менять коды, зависящие от i , т. е. команды p , $p+1$, заготовку z_3 и команду засылки 0 в y :

$$p-2 \quad 0 \quad ПЧ \quad 0 \quad 0 \quad y+i$$

Переадресацию выполняют следующие команды:

$$\begin{array}{l} 0 \quad +M \quad p-2 \quad z_4 \quad p-2 \\ 0 \quad +M \quad p \quad z_5 \quad p \\ 0 \quad +M \quad p+1 \quad z_6 \quad p+1 \\ 0 \quad +M \quad z_3 \quad z_5 \quad z_3 \end{array}$$

с помощью констант:

$$\begin{array}{l} z_4 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \\ z_5 \quad 0 \quad 0 \quad +n \quad 0 \quad 0 \\ z_6 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \end{array}$$

Мы видим, что во внешнем цикле команду p меняют две команды — одна (восстановление по j) уменьшает первый и второй адрес на n единиц, вторая (переадресация по i) увеличивает первый адрес на те же n единиц. Это равносильно уменьшению второго адреса на n единиц, что можно осуществить операцией

$$0 \quad -M \quad p \quad z_{13} \quad p$$

где

$$z_{13} \quad 0 \quad 0 \quad 0 \quad +n \quad 0$$

Все остальные переменные коды, как и прежде, будем только переадресовывать по i .

Напишем полученную программу, сохраняя, для наглядности, те же обозначения констант, что в программе 1.

Программа 2

	0	ПЧ	z_8	0	$p-2$	} $i := 1$
	0	ПЧ	z_9	0	$p+1$	
	0	ПЧ	z_{11}	0	z_3	
	0	ПЧ	z_{10}	0	p	$i := 1, j := 0$
$p-2$	0	ПЧ	0	0	$y+i$	$y_i := 0$
$p-1$	0	$+M$	p	z_2	p	$j := j+1$
p	0	$\times$	$a+(i-1) \times$ $\times n+j$	$x+j$	r	$a_{ij} x_j$
$p+1$	0	$+$	$y+i$	r	$y+i$	$y_i := y_i + a_{ij} x_j$
	0	$\neq$	p	z_3	0	Если $j \neq n$, то ($\omega := 0$)
	0	П0	0	$p-1$	0	перейти на $p-1$
	0	$-M$	p	z_{13}	p	$j := j-n, i := i+1$ (восстановление и переадресация команды p)
	0	$+M$	$p-2$	z_4	$p-2$	} $i := i+1$
	0	$+M$	$p+1$	z_6	$p+1$	
	0	$+M$	z_3	z_5	z_3	
	0	$\neq$	$p-2$	z_7	0	Если $i \neq m+1$, то ($\omega := 0$)
	0	П0	0	$p-2$	0	перейти на $p-2$

Использованные константы:

z_2	0	0	1	1	0
z_4	0	0	0	0	1
z_5	0	0	$+n$	0	0
z_6	0	0	1	0	1
z_7	0	ПЧ	0	0	$y+m+1$
z_8	0	ПЧ	0	0	$y+1$
z_9	0	$+$	$y+1$	r	$y+1$
z_{10}	0	$\times$	a	x	r
z_{11}	0	$\times$	$a+n$	$x+n$	r
z_{13}	0	0	0	$+n$	0

Общий вид содержимого рабочей ячейки z_3 :

$$z_3 \quad 0 \quad \times \quad a+in \quad x+n \quad r$$

Полученную программу все еще можно улучшить. Во внутреннем цикле невыгодно получать сумму в ячейке с переменным адресом $y+i$. Этот адрес используется в двух командах ($p-2$ и $p+1$), обе команды приходится формировать и переадресовывать. Если же во внутреннем цикле вместо ячейки $y+i$ использовать рабочую ячейку с постоянным адресом, а после получения суммы произвести пересылку результата из этой ячейки в ячейку $y+i$, то во внеш-

нем цикле можно сэкономить по одной команде пересдресации и формирования. Это — еще один из приемов чистки циклов, знакомой нам по § 5 гл. 7. Такая программа будет соответствовать алгоритму:

```

for  $i := 1$  step 1 until  $m$  do
  begin  $s := 0$ ;
    for  $j := 1$  step 1 until  $n$  do
       $s := s + a[i, j] \times x[j]$ ;
     $y[i] := s$ 
  end

```

Напишем сразу готовую программу, сохраняя прежние обозначения:

Программа 3

	0	ПЧ	z_{10}	0	p	$i := 1, j := 0$
	0	ПЧ	z_{14}	0	$p+4$	} $i := 1$
	0	ПЧ	z_{11}	0	z_3	
$p-2$	0	ПЧ	0	0	s	$s := 0$
$p-1$	0	+M	p	z_2	p	$j := j+1$
p	0	×	$a+(i-1) \times$ $\times n+j$	$x+j$	r	$a_{ij}x_j$
	0	+	s	r	s	$s := s + a_{ij}x_j$
	0	$\neq$	p	z_3	0	Если $j \neq n$, то перейти на $p-1$
$p+3$	0	П0	0	$p-1$	0	
$p+4$	0	ПЧ	s	0	$y+i$	$y_i := s$
	0	-M	p	z_{13}	p	$j := j-n, i := i+1$
	0	+M	$p+4$	z_4	$p+4$	} $i := i+1$
	0	+M	z_3	z_5	z_3	
	0	$\neq$	$p+4$	z_{15}	0	Если $i \neq m+1$, то перейти на $p-2$
	0	П0	0	$p-2$	0	

где

z_2	0	0	1	1	0
z_4	0	0	0	0	1
z_5	0	0	$+n$	0	0
z_{10}	0	×	a	x	r
z_{11}	0	×	$a+n$	$x+n$	r
z_{13}	0	0	0	$+n$	0
z_{14}	0	ПЧ	s	0	$y+1$
z_{15}	0	ПЧ	s	0	$y+m+1$

Общий вид переменной заготовки, используемой для сравнения:

$$z_3 \quad 0 \quad \times \quad a+in \quad x+n \quad r$$

Итак, нам удалось сократить и число команд в программе, и число заготовок (заметим, что, используя особенности системы команд машин типа М-20, в последнем варианте программы можно

заменить команды $p + 3$ и $p + 4$ одной, совместив передачу управления с пересылкой числа, соответственно нужно изменить вид заготовок z_{14} и z_{15}). При программировании циклов нужно помнить, что основной источник экономии машинного времени заключается в сокращении числа команд самого внутреннего цикла. Так, в написанной программе при одном выполнении первые три команды выполняются один раз, команда $p - 2$ выполняется m раз, команды $p - 1 \div p + 3$ выполняются $m \times n$ раз и последние 6 команд — m раз, т. е. команды самого внутреннего цикла выполняются наибольшее число раз.

Рассмотрим еще один пример.

Пример 2. Составить программу транспонирования квадратной матрицы. Элементы транспонированной матрицы поместить в те же ячейки памяти, где находится заданная матрица.

Воспользуемся следующим алгоритмом:

```
for i := 2 step 1 until n do
  for j := 1 step 1 until i - 1 do
    begin r1 := a[i, j]; a[i, j] := a[j, i]; a[j, i] := r1
  end
```

В этом примере, в отличие от предыдущего, верхняя граница изменения параметра внутреннего цикла зависит от параметра внешнего цикла. Такая зависимость (обычно линейная) встречается достаточно часто. Посмотрим, как программируются такие циклы.

Будем считать, что матрица записана в памяти машины по строкам, так что адрес a_{kl} равен $a + (k - 1)n + l$. Как всегда, программирование начинаем с составления программы внутреннего цикла. Переадресацию переменных команд будем производить после их выполнения. Рабочая часть цикла, очевидно, состоит из трех операций пересылок чисел.

p	0	ПЧ	$a + (i - 1)n + j$	0	r_1	$r_1 := a_{ij}$
$p + 1$	0	ПЧ	$a + (j - 1)n + i$	0	$a + (i - 1)n + j$	$a_{ij} := a_{ji}$
$p + 2$	0	ПЧ	r_1	0	$a + (j - 1)n + i$	$a_{ji} := r_1$
	0	+M	p	z_1	p	} $j := j + 1$
	0	+M	$p + 1$	z_2	$p + 1$	
	0	+M	$p + 2$	z_3	$p + 2$	
	0	$\neq$	p	r_2	0	Если $j \neq i$, то перейти на p
	0	ПО	0	p	0	

Значения констант:

z_1	0	0	1	0	0
z_2	0	0	+n	0	1
z_3	0	0	0	0	+n

Какой вид должна иметь заготовка для сравнения в ячейке r_2 ? Параметр j изменяется до значения $i - 1$, переадресация команды

происходит после ее выполнения, т. е. мы должны прекращать выполнение цикла при $j = i$. Первый адрес команды p будет при этом иметь значение $a + (i - 1)n + i$, т. е. $a + i(n + 1) - n$. Следовательно, заготовка r_2 зависит от i , как и следовало ожидать, и имеет вид:

$$r_2 \quad 0 \quad ПЧ \quad a + i(n + 1) - n \quad 0 \quad r_1$$

Чтобы закончить программирование внутреннего цикла, нужно в его подготовительной части осуществить формирование всех переменных команд для значения $j = 1$:

$$\left. \begin{array}{l} 0 \quad ПЧ \quad r_3 \quad 0 \quad p \\ 0 \quad ПЧ \quad r_4 \quad 0 \quad p + 1 \\ 0 \quad ПЧ \quad r_5 \quad 0 \quad p + 2 \end{array} \right\} j := 1$$

где заготовки r_3 , r_4 и r_5 также зависят от параметра i :

$$\begin{array}{llllll} r_3 & 0 & ПЧ & a + (i - 1)n + 1 & 0 & r_1 \\ r_4 & 0 & ПЧ & a + i & 0 & a + (i - 1)n + 1 \\ r_5 & 0 & ПЧ & r_1 & 0 & a + i \end{array}$$

Во внешнем цикле меняется содержимое ячеек r_2 , r_3 , r_4 , r_5 . Добавим команды переадресации (поместим их перед командами формирования внутреннего цикла):

$$\left. \begin{array}{l} p_1 \quad 0 \quad +M \quad r_2 \quad z_4 \quad r_2 \\ 0 \quad +M \quad r_3 \quad z_5 \quad r_3 \\ 0 \quad +M \quad r_4 \quad z_6 \quad r_4 \\ 0 \quad +M \quad r_5 \quad z_7 \quad r_5 \end{array} \right\} i := i + 1$$

и константы

$$\begin{array}{lllll} z_4 & 0 & 0 & n + 1 & 0 & 0 \\ z_5 & 0 & 0 & +n & 0 & 0 \\ z_6 & 0 & 0 & 1 & 0 & +n \\ z_7 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Сравнение на окончание цикла по i будем проводить по содержанию ячейки r_5 :

$$\begin{array}{llll} 0 & \neq & r_5 & z_8 \quad 0 \\ 0 & ПО & 0 & p_1 \quad 0 \end{array}$$

где

$$z_8 \quad 0 \quad ПЧ \quad r_1 \quad 0 \quad a + n$$

Выпишем готовую программу, дополнив все написанные команды формированием для $i = 1$ (переадресация команд происходит до их выполнения, начинать цикл нужно со значения $i = 2$).

Программа 4

	0	ПЧ	z_9	0	r_2	} $i := 1$
	0	ПЧ	z_9	0	r_3	
	0	ПЧ	z_{10}	0	r_4	
	0	ПЧ	z_{11}	0	r_5	
p_1	0	+M	r_2	z_4	r_2	} $i := i + 1$
	0	+M	r_3	z_5	r_3	
	0	+M	r_4	z_6	r_4	
	0	+M	r_5	z_7	r_5	
	0	ПЧ	r_3	0	p	} $j := 1$
	0	ПЧ	r_4	0	$p+1$	
	0	ПЧ	r_5	0	$p+2$	
p	0	ПЧ	$a+(i-1) \times$ $\times n+j$	0	r_1	$r_1 := a_{ij}$
$p+1$	0	ПЧ	$a+(j-1) \times$ $\times n+i$	0	$a+(i-1) \times$ $\times n+j$	$a_{ij} := a_{ji}$
$p+2$	0	ПЧ	r_1	0	$a+(j-1) \times$ $\times n+i$	$a_{ji} := r_1$
	0	+M	p	z_1	p	} $j := j + 1$
	0	+M	$p+1$	z_2	$p+1$	
	0	+M	$p+2$	z_3	$p+2$	
	0	$\neq$	p	r_2	0	Если $j \neq i$, то перейти на p
	0	ПО	0	p	0	
	0	$\neq$	r_5	z_8	0	Если $i \neq n$, то перейти на p_1
	0	ПО	0	p_1	0	

Использованные константы:

z_1	0	0	1	0	0
z_2	0	0	+n	0	1
z_3	0	0	0	0	+n
z_4	0	0	n+1	0	0
z_5	0	0	+n	0	0
z_6	0	0	1	0	+n
z_7	0	0	0	0	1
z_8	0	ПЧ	r_1	0	$a+n$
z_9	0	ПЧ	$a+1$	0	r_1
z_{10}	0	ПЧ	$a+1$	0	$a+1$
z_{11}	0	ПЧ	r_1	0	$a+1$

Общий вид переменных команд:

r_2	0	ПЧ	$a+i(n+1)-n$	0	r_1
r_3	0	ПЧ	$a+(i-1)n+1$	0	r_1
r_4	0	ПЧ	$a+i$	0	$a+(i-1)n+1$
r_5	0	ПЧ	r_1	0	$a+i$

Как мы видели, при составлении этой программы зависимость предела внутреннего цикла от значения параметра внешнего цикла

особых затруднений не вызвала, программа была составлена обычным образом. Возникает вопрос, нельзя ли сократить написанную программу, применив восстановление команд? Оказывается, нет, — в этом случае зависимость верхней границы значения j от параметра i препятствует сокращению. Полное изменение переменных команд в цикле по j зависит от значения i , значит, должны зависеть от i и «константы» восстановления. Их, в свою очередь, нужно переадресовывать во внешнем цикле. Число команд в программе применением восстановления сократить не удастся.

У п р а ж н е н и я. 1. Переделать программу 4, используя восстановление по j .

2. Составить программу транспонирования квадратной матрицы так, чтобы во внутреннем цикле элементы одной из диагоналей, параллельных главной диагонали матрицы (т. е. элементы $a[i, i+k]$ при $i = 1, \dots, n-k$ и фиксированном k) переставлялись с симметричными им элементами, а во внешнем цикле осуществлялся перебор всех таких диагоналей (для $k = 1, \dots, n-1$).

3. Составить программу вычисления разделенных разностей по заданной таблице значений аргумента и функции. Аппарат разделенных разностей очень удобен для интерполяции функций одной переменной при неравноотстоящих значениях аргумента. Напомним определение разделенных разностей, одновременно дающее метод их вычисления.

Пусть заданы значения функции $f(x)$ для $n+1$ значения аргумента: $f(x_0), f(x_1), \dots, f(x_n)$. Разделенные разности первого порядка определяются формулой:

$$f(x_i, x_j) = \frac{f(x_j) - f(x_i)}{x_j - x_i}, \quad i \neq j.$$

Нас, в частности, будут интересовать лишь разности

$$\begin{aligned} f(x_0, x_1) &= \frac{f(x_1) - f(x_0)}{x_1 - x_0}; \quad f(x_1, x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}; \dots \\ \dots; f(x_{n-1}, x_n) &= \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}. \end{aligned} \quad (5)$$

Разделенные разности второго порядка определяются по формуле:

$$f(x_i, x_j, x_k) = \frac{f(x_j, x_k) - f(x_i, x_j)}{x_k - x_i}, \quad i \neq j, \quad i \neq k, \quad j \neq k,$$

в частности:

$$\begin{aligned} f(x_0, x_1, x_2) &= \frac{f(x_1, x_2) - f(x_0, x_1)}{x_2 - x_0}; \dots \\ \dots; f(x_{n-2}, x_{n-1}, x_n) &= \frac{f(x_{n-1}, x_n) - f(x_{n-2}, x_{n-1})}{x_n - x_{n-2}}. \end{aligned} \quad (6)$$

Вообще, разделенные разности m -го порядка вычисляются по формуле:

$$f(x_{i_0}, x_{i_1}, \dots, x_{i_m}) = \frac{f(x_{i_1}, \dots, x_{i_m}) - f(x_{i_0}, \dots, x_{i_{m-1}})}{x_{i_m} - x_{i_0}}.$$

Мы будем пользоваться этой формулой лишь для последовательных значений аргументов: $x_{i_0} = x_i, x_{i_1} = x_{i+1}, \dots, x_{i_m} = x_{i+m}$ т. е. формулой

$$f(x_i, x_{i+1}, \dots, x_{i+m}) = \frac{f(x_{i+1}, \dots, x_{i+m}) - f(x_i, \dots, x_{i+m-1})}{x_{i+m} - x_i}, \quad (7)$$

где m может принимать значения $m = 1, 2, \dots, n$, а i для данного m — значения $i = 0, 1, 2, \dots, n - m$.

Нашей конечной целью является вычисление последовательности разделенных разностей $f(x_0), f(x_0, x_1), f(x_0, x_1, x_2), \dots, f(x_0, x_1, \dots, x_n)$. При этом придется вычислить n разностей первого порядка (5), $n - 1$ разность второго порядка (6) и т. д. Будем считать, что значение аргумента x_i размещено, как обычно, в ячейке с адресом $x + i$, а соответствующее значение функции $f(x_i)$, вычисленное заранее, — в ячейке $f + i$. Так как из исходных значений функции нужно сохранить лишь одно $f(x_0)$, то разности первого порядка (5) можно вычислять в ячейках $f + 1, f + 2, \dots, f + n$:

$$f(x_0, x_1) \text{ — в } f + 1, \dots, f(x_{n-1}, x_n) \text{ — в } f + n.$$

Первую из этих разностей сохраним в ячейке $f + 1$, в ячейках $f + 2, \dots, f + n$ вычислим нужные нам разделенные разности второго порядка (6) и т. д., т. е. рекомендуется вести вычисления по следующей схеме:

адрес

f	$f(x_0)$			
$f + 1$	$f(x_1)$	$f(x_0, x_1)$		
$f + 2$	$f(x_2)$	$f(x_1, x_2)$	$f(x_0, x_1, x_2)$	
.				
$f + n$	$f(x_n)$	$f(x_{n-1}, x_n)$	$f(x_{n-2}, x_{n-1}, x_n) \dots f(x_0, x_1, \dots, x_n)$	

так что адресом разности $f(x_i, x_{i+1}, \dots, x_{i+j})$ является $f + i + j$.

Глава 10. ИНДЕКС-РЕГИСТР

§ 1. Система команд М-20 (продолжение)

Мы уже знаем, что кроме ячеек оперативной памяти, о которых чаще всего приходится думать программисту, данные в машине могут размещаться в ячейках специального назначения, так называемых *регистрах*. Регистры не имеют адресов и обращение к ним происходит по особым правилам. Емкость регистра зависит от его «специализации»: так, регистр признака ω состоит всего из одного разряда, регистр команды (в него поступает очередная выполняемая команда) — из 45 разрядов. *Индекс-регистр* (его еще называют *регистром адреса*) состоит из 12 разрядов, так что его значение — двенадцатиразрядное двоичное число. В дальнейшем будем обозначать этот регистр буквой I .

Познакомимся с понятием исполнительного адреса. Код команды машин типа М-20 содержит три признака модификации адресов π_1 , π_2 , π_3 (45, 44 и 43 разряды). Если какой-либо из этих признаков отличен от нуля, то при выполнении команды к числу, написанному в соответствующем адресе, прибавляется значение индекс-регистра (сложение происходит по модулю 2^{12}). Полученное число рассматривается машиной как соответствующий адрес команды и называется *исполнительным* или *модифицированным* адресом. Если обозначить i -й исполнительный адрес через A'_i , то

$$A'_i = A_i + \pi_i \cdot I \pmod{2^{12}}, \quad i = 1, 2, 3,$$

т. е.

$$A'_i = A_i \text{ при } \pi_i = 0 \text{ и } A'_i = A_i + I \pmod{2^{12}} \text{ при } \pi_i = 1.$$

До сих пор мы рассматривали команды с нулевыми значениями признаков модификации. Их исполнительные адреса совпадали с адресами, фактически записанными в команде. Посмотрим, как выполняются команды, если $\pi \neq 0$ (значения I будем записывать в восьмеричной системе, как и адреса команд).

Пусть $I = 0002$ и выполняется команда:

$$3 \quad + \quad 1020 \quad 2317 \quad 0015$$

В этой команде $\pi_1 = 0$, $\pi_2 = 1$, $\pi_3 = 1$, т. е.

$$A_1 = A_1 = 1020, A_2' = A_2 + I = 2321, A_3' = A_3 + I = 0017,$$

и при данном значении I ее выполнение равносильно выполнению команды

$$0 \quad + \quad 1020 \quad 2321 \quad 0017$$

Если $I = 7777$, то команда

$$5 \quad ПЧ \quad a \quad 0 \quad b$$

при $a \neq 0$ и $b \neq 0$ выполнится как

$$0 \quad ПЧ \quad a-1 \quad 0 \quad b-1$$

Исполнительные адреса всегда формируются до выполнения любых других действий, предписанных командой.

Благодаря модификации адресов в простых случаях можно избавиться от переменных команд. Так, если в программе потребовалась команда вида

$$0 \quad КОП \quad a+i \quad b+i \quad c+i$$

то достаточно положить $\pi_1 = \pi_2 = \pi_3 = 1$ (т. е. $\pi = 7$), записать команду в виде

$$7 \quad КОП \quad a \quad b \quad c$$

и послать в индекс-регистр значение i . Очевидно, должны существовать операции, изменяющие значение I , и операции сравнения значения I с заданными числами (для проверки на окончание цикла). Познакомимся с ними.

Для изменения содержимого индекс-регистра имеются две операции:

$$\pi \quad АИ \quad A_1 \quad A_2 \quad A_3 \quad (\text{восьмеричный код } 52)$$

$$\pi \quad КИ \quad A_1 \quad A_2 \quad A_3 \quad (72)$$

Первая из них ($АИ$) посылает в индекс-регистр значение A_2' второго исполнительного адреса выполняемой команды. Например, после выполнения команды

$$0 \quad АИ \quad 0 \quad 0001 \quad 0$$

значение I становится равно 1, независимо от того, чему оно равнялось до выполнения этой команды. В результате выполнения команды

$$2 \quad АИ \quad 0 \quad 0001 \quad 0$$

значение I увеличится на 1: $I := I + 1$ (если перед этим оно было меньше $7777_8 = 2^{12} - 1$).

Операция *КИ* посылает в индекс-регистр содержимое разрядов $24 \div 13$ (т. е. второго адреса) слова, хранящегося в ячейке с номером A'_2 . Например, если в ячейке p находится команда

$$п + a \ b \ c$$

то после выполнения операции

$$0 \ КИ \ 0 \ p \ 0$$

значение *И* будет равно b .

Кроме того, при выполнении команд

$$п \ АИ \ A_1 \ A_2 \ A_3$$

и

$$п \ КИ \ A_1 \ A_2 \ A_3$$

в ячейку с номером A'_3 засылается команда:

$$0 \ АИ \ 0 \ A'_1 \ 0$$

второй адрес которой равен первому исполнительному адресу выполняемой команды.

Пусть, например, $И = 0016$. Тогда после выполнения команды

$$4 \ АИ \ 0001 \ 0002 \ A_3$$

в ячейку A_3 зашлется команда

$$0 \ АИ \ 0 \ 0017 \ 0$$

и значение *И* будет равно 0002.

Засылка в ячейку A'_3 команды $0 \ АИ \ 0 \ A'_1 \ 0$ позволяет одновременно с присваиванием индекс-регистру нового значения запомнить его старое значение. Для этого надлежит выполнить команду $4 \ АИ \ 0 \ A_2 \ A_3$. Впоследствии, при выполнении засланной команды прежнее значение индекс-регистра может быть восстановлено.

Рассмотрим теперь группу операций перехода после цикла. Эти операции обеспечивают передачу управления по второму исполнительному адресу при выполнении определенных условий и засылку в индекс-регистр значения третьего исполнительного адреса. Если $\pi_3 = 1$, то в индекс-регистр будет послана величина

$$И_{\text{нов}} = A'_3 = И_{\text{ст}} + A_3 \pmod{2^{13}}$$

(индексы «нов» и «ст» обозначают соответственно новый и старый), т. е. при каждом выполнении команды к значению *И* будет прибавляться код A_3 . Если условия перехода по A'_3 не выполнены, то управление передается следующей по порядку команде. Выпишем операции, используемые для сравнения значения *И*; результат выполнения команд приблизительно описывается записью на АЛГОЛе,

подробное объяснение дано ниже (в скобках записаны восьмеричные коды операций):

- $I < A$ (12) $C := I; I := A_3; \text{ if } C < A_1 \text{ then go to } A_2$
 $I \geq A$ (32) $C := I; I := A_3; \text{ if } C \geq A_1 \text{ then go to } A_2$
 $1I < A$ (11) $C := I; I := A_3; \text{ if } C < A_1 \wedge \omega = 1 \text{ then go to } A_2$
 $1I \geq A$ (31) $C := I; I := A_3; \text{ if } C \geq A_1 \wedge \omega = 1 \text{ then go to } A_2$
 $0I < A$ (51) $C := I; I := A_3; \text{ if } C < A_1 \wedge \omega = 0 \text{ then go to } A_2$
 $0I \geq A$ (71) $C := I; I := A_3; \text{ if } C \geq A_1 \wedge \omega = 0 \text{ then go to } A_2$

Поясним приведенные операции. Запись

$C := I; I := A_3; \text{ if } C < A_1 \text{ then go to } A_2$

означает, что значение C индекс-регистра, с которым началось выполнение операции $I < A$, сравнивается с числом, равным первому исполнительному адресу команды. Если значение C меньше этого числа, происходит передача управления по второму исполнительному адресу (т. е. следующей выполняется команда, находящаяся в ячейке A_2); независимо от исхода сравнения в индекс-регистр заносится значение третьего исполнительного адреса. Необходимо подчеркнуть, что во всех вышеприведенных командах окончания цикла первый и третий адрес рассматриваются не как адрес некоторой ячейки, а как двенадцатиразрядное двоичное число. С числом, представленным первым исполнительным адресом, сравнивается значение индекс-регистра, число, равное A_3 , заносится в индекс-регистр.

Команда $I \geq A$ выполняется аналогично команде $I < A$, но для перехода по второму адресу необходимо выполнение условия $C \geq A_1$.

При выполнении следующих четырех операций ($1I < A$, $1I \geq A$, $0I < A$, $0I \geq A$) кроме значения C анализируется значение признака ω , оставшееся от предыдущей команды. Так, при выполнении команды

п $1I < A \quad A_1 \quad A_2 \quad A_3$

для передачи управления в ячейку A_2 необходимо одновременное соблюдение двух условий: $C < A_1$ и $\omega = 1$. Если хоть одно из этих условий не выполнено, управление передается следующей команде. Условия, необходимые для передачи управления по A_2 при выполнении операций $1I \geq A$, $0I < A$ и $0I \geq A$, ясны из записи их действия на АЛГОЛе (и из условного обозначения кода операций этих команд).

В машинах БЭСМ-4 и М-220 есть еще две операции перехода после цикла. Эти операции с кодами $I < K$ (40) и $I \geq K$ (60) похожи на операции $I < A$ и $I \geq A$ соответственно. Различие состоит лишь в том, что при их выполнении

старое значение индекс-регистра сравнивается со вторым адресом слова, выбранного из ячейки по второму исполнительному адресу, а не с самим вторым исполнительным адресом, как в случае операций $I < A$ и $I \geq A$. Можно сказать, что операция $I < K$ (или $I \geq K$) находится в таком же отношении к операции $I < A$ ($I \geq A$), в каком операция KI находится к операции AI . В ряде случаев операции $I < K$ и $I \geq K$ очень удобны и позволяют заметно сократить программы. Но мы редко будем к ним прибегать в расчете на то, что программист, умеющий составить программу, не включая в нее этих операций, сумеет и сократить ее, если эти операции предоставить в его распоряжение.

Все команды перехода после цикла, так же как операции AI и KI , не изменяют значения признака ω , выработанного предыдущими командами.

Теперь мы достаточно полно ознакомились с особенностями машин типа М-20 и можем уточнить приведенное в § 1 гл. 1 описание элементарного цикла работы машины применительно к машинам этого типа. Этот цикл складывается из следующих действий.

1. По адресу, находящемуся на регистре адреса команды, считывается из ОЗУ на регистр команды очередная команда программы.

2. Анализируется код операции, в результате чего определяется вид предстоящей операции.

3. Формируются исполнительные адреса. Это делается на специальном регистре устройства управления, называемом *сумматором адресов*. Таким образом, как в ОЗУ, так и на регистре команд команда хранится только с немодифицированными адресами. Адреса формируются поочередно.

4. Если требуется, по первому, а затем по второму исполнительному адресу на регистры арифметического устройства считываются операнды операции.

5. Выполняется сама операция.

6. Результат операции, если это требуется, записывается в ОЗУ по третьему исполнительному адресу.

7. Вырабатывается значение признака ω (иногда совпадающее со старым) и посылается на соответствующий регистр.

8. Формируется новое содержимое индекс-регистра, которое также может совпадать с предыдущим.

9. Вычисляется и посылается на регистр адреса команды значение адреса следующей команды. Если выполняемая команда не является командой передачи управления, то этот адрес получается в результате добавления единицы к его предыдущему значению.

Расчленение операции на эти действия в значительной степени условно. Многие из них могут выполняться и фактически выполняются параллельно. Так, например, действие 2 может выполняться одновременно с началом действия 3. Действие 3, в свою очередь, может продолжаться параллельно с выполнением действий 4 и 5. В действие 5 по существу входит многое из того, что выделено в действия 7, 8 и 9 (выработка значений, которые при выполнении этих действий посылаются на соответствующие регистры). Для

многих машинных операций выполнять некоторые из перечисленных действий фактически не требуется. Но все же выполнение каждой конкретной операции проще описать и легче понять, если оговорить, какие из названных действий выполнять не нужно.

Например, при выполнении всех операций перехода после цикла действия 4, 6 и 7 не выполняются, при выполнении действия 8 на индекс-регистр посылается значение третьего исполнительного адреса, а при выполнении действия 9 (которое фактически предшествует действию 8, поскольку в нем используется старое значение индекс-регистра) содержимое регистра адреса команды либо увеличивается на единицу, либо заменяется вторым исполнительным адресом.

Действия 1, 2, 3, 5 и 9 сопровождают выполнение любой машинной операции.

§ 2. Программирование с использованием индекс-регистра

Рассмотрим участок программы, соответствующий оператору цикла

for $i := a$ step b until c do S

где i — целочисленная переменная, выражения a , b и c определяют целые значения, причем значения переменной i и выражений b и c не меняются при выполнении оператора S. Пусть в командах, реализующих этот оператор, все адреса, зависящие от параметра цикла i , имеют вид $A_j + i$ (j — номер адреса в команде), где A_j уже не зависит от i . При этих условиях рассматриваемый участок программы можно составить так, чтобы он не содержал переменных команд, требующих явной переадресации по i (явной мы называем переадресацию с помощью операций $+M$ или $-M$; переадресация, связанная с изменением исполнительных адресов при изменении значения индекс-регистра, считается неявной).

Если $a \geq 0$, $b > 0$ и $c > 0$, то участок можно записать в виде:

$p-1$	0	AI	0	$+a$	0	$I := a$
p		Команды, соответствующие				} Во время выполнения этих команд текущее значение I равно i .
$\vdots$		оператору S (p — адрес ячейки,				
p_1		содержащей первую из этих команд)				
p_1+1	1	$I < A$	$c-b+1$	p	$+b$	$I := I + b$; если $I \leq c$, то на p

Команда $p-1$ присвоит индекс-регистру I значение $i = a$. В командах $p \div p_1$ все переменные адреса записываются в виде A_j , а в соответствующий признак модификации помещается 1. При этом исполнительные адреса будут иметь требуемое значение $A_j + i$. Команда $p_1 + 1$ сравнит значение I , равное i , с $c - b + 1$, и если $I < c - b + 1$ ($I \leq c - b$), то управление будет передано команде p . Одновременно значение I получит приращение b . Новое значение I удовлетворяет неравенству $I \leq c$. Если же новое

значение I больше c , то управление перейдет к команде $p_1 + 2$, и выполнение цикла закончится.

Если параметр цикла уменьшается, т. е. $b < 0$, $a > c \geq 0$, то операцию окончания цикла можно взять в виде

$$p_1 + 1 \quad 1 \quad I \geq A \quad c + |b| \quad p \quad 2^{12} - |b|$$

Значение I сравнивается с $c + |b|$. Если текущее значение I больше или равно $c + |b|$, то цикл повторится еще раз со значением I , не меньшим c . Если же текущее значение I меньше $c + |b|$, то новое его значение окажется меньше c , но цикл повторяться уже не будет. Остальные команды цикла ($p - 1 \div p_1$) следует оставить в прежнем виде (как для случая $b > 0$).

Пример 1. Вычислить сумму $s = \sum_{i=1}^n x_i$. На АЛГОЛе:

```

s := 0;
for i := 1 step 1 until n do
  s := s + x[i]

```

Как и раньше, будем считать адрес x_i равным $x + i$.

Рабочая часть программы состоит из одной команды:

$$0 \quad + \quad s \quad x + i \quad s$$

Если заслать в индекс-регистр значение i , то можно избавиться от переменного адреса. Рабочая команда будет выглядеть так:

$$p \quad 2 \quad + \quad s \quad x \quad s \quad s := s + x[i] \quad (I = i)$$

Сравнение на окончание цикла и изменение i можно совместить в одной команде:

$$1 \quad I < A \quad + n \quad p \quad 0001$$

Осталось выполнить подготовительные операции — заслать в s нуль и в I единицу. Объединив эти операции с уже написанными, получим следующую программу:

$$\begin{array}{llllll}
0 & ПЧ & 0 & 0 & s & s := 0 \\
0 & АИ & 0 & 1 & 0 & I := 1 \\
p & 2 & + & s & x & s & s := s + x_i \quad (I = i) \\
1 & I < A & + n & p & 0001 & I := I + 1; \text{ если } I - 1 < n, \text{ то перейти на } p
\end{array}$$

Эту же сумму можно получить и с помощью следующей программы:

$$\begin{array}{llllll}
0 & ПЧ & 0 & 0 & s & \\
0 & АИ & 0 & 0 & 0 & I := 0 \\
p & 2 & + & s & x + 1 & s & (I = i - 1) \\
1 & I < A & n - 1 & p & 0001 &
\end{array}$$

Однако этот вариант менее нагляден — мы изменили границы значений i , заданные алгоритмом, не получив при этом никаких преимуществ по сравнению с первым вариантом.

Если приведенный цикл — часть какой-то программы, то часто бывает необходимо сохранить значение индекс-регистра, с которым мы пришли к выполнению данного цикла. Это достигается, например, следующей программой:

	0	ПЧ	0	0	s	$s := 0$
$p-1$	4	АИ	0	1	$p+2$	Запоминание $I_{ст}$ в ячейке $p+2$, $I := 1$
p	2	+	s	x	s	$s := s + x_i$ ($I=i$)
	1	$I < A$	$+n$	p	0001	$I := I + 1$, если $I \leq n$, то перейти на p
$p+2$	0	АИ	0	$I_{ст}$	0	Восстановление $I_{ст}$

Перед выполнением программы в $p+2$ может стоять любой код — команда $p-1$ зайдет в эту ячейку команду

$p+2 \quad 0 \quad АИ \quad 0 \quad I_{ст} \quad 0$

где $I_{ст}$ — значение I , с которым начинается выполнение команды $p-1$. После выхода из цикла управление будет передано команде $p+2$, которая пошлет в индекс-регистр значение $I_{ст}$ или, как еще говорят, *восстановит* его.

Пример 2. Вычислить значение полинома

$$y = a_0 + a_1x + \dots + a_nx^n \quad (\text{адрес } a_i \text{ равен } a+i).$$

Алгоритм вычисления на АЛГОЛе:

$y := 0$; for $i := n$ step -1 until 0 do $y := y \times x + a[i]$

Начнем с программирования рабочей части цикла:

$p \quad 0 \quad \times \quad y \quad x \quad y \quad yx$
 $2 \quad + \quad y \quad a \quad y \quad y := yx + a_i$

Сравнение на окончание цикла совместим с изменением значения i . Так как сложение $I + A_3$ происходит по модулю 2^{12} , то уменьшение I на единицу равносильно прибавлению константы 7777. Значение параметра i убывает, так что в этом цикле следует воспользоваться операцией $I \geq A$. Что нужно написать в первом адресе команды окончания цикла? Конечное значение параметра цикла должно быть равно 0, но нужно помнить, что сначала происходит сравнение, а затем изменение I . Значит, последнее значение I , которое еще «не пропускает» сравнение, должно равняться 1, т. е. команда окончания цикла должна быть написана в виде

1 $I \geq A$ 0001 p 7777 $C := I$; $I := I - 1$; если $C \geq 1$, то перейти на p

(Заметим, что I не может принимать отрицательные значения, т. е. всегда $I \geq 0$.)

Дополнив полученные команды подготовительными операциями получим следующую программу:

Программа 1

	0	ПЧ	0	0	y	$y := 0$
	0	АН	0	+n	0	$I := n$
p	0	×	y	x	y	yx
	2	+	y	a	y	$y := yx + a_i \ (I=i)$
	1	$I \geq A$	0001	p	7777	$I := I - 1$; если $I \geq 0$, то перейти на p

Чуть более экономный вариант программы таков. На АЛГОЛе

```

y := a [n];
for i := n - 1 step -1 until 0 do
  y := y × x + a [i]

```

В кодах М-20:

	0	АН	0	+n	0	$I := n$
	4	ПБ	a	p+2	y	$y := a [n]$, перейти на p+2
p	0	×	y	x	y	$y \times x$
	2	+	y	a	y	$y := y \times x + a [i] \ (I=i)$
	1	$I \geq A$	1	p	7777	$I := I - 1$, если $I \geq 0$, то на p

Пример 3. Вычислить $z_i = \sqrt{x_i^2 + y_i^2}$ для $i = 1, 2, \dots, n$. Составим два варианта программы.

Вариант 1. Считаем, что адрес x_i равен $x + i$, адрес y равен $y + i$, адрес z_i равен $z + i$. Вычисления проводятся по алгоритму:

```

for i := 1 step 1 until n do
  z [i] := sqrt (x [i] ↑ 2 + y [i] ↑ 2)

```

Как написать программу такого цикла, мы уже знаем, поэтому приведем сразу готовую программу:

Программа 2

	0	АН	0	1	0	$I := 1 \ (i := 1)$
p	7	×	x	x	z	x_i^2
	6	×	y	y	r	y_i^2
	5	+	z	r	z	$x_i^2 + y_i^2$
	5	√	z	0	z	$z_i := \sqrt{x_i^2 + y_i^2}$
	1	$I < A$	+n	p	0001	$I := I + 1$, если $I < n + 1$, то перейти на p

Ячейку z_i можно было не использовать для хранения промежуточного результата, но при использовании индекс-регистра мы только увеличиваем число модифицированных адресов, не удлиняя программу, не усложняя и не замедляя ее выполнение.

В а р и а н т 2. Значения x и y заданы одним массивом a , так что адрес x_i равен $a + 2i - 1$, адрес y_i равен $a + 2i$. Как и в варианте 1, адрес z_i равен $z + i$. В данном варианте программа на АЛГОЛе описывается циклом

for $i := 1$ step 1 until n do
 $z[i] := \text{sqrt}(a[2 \times i - 1] \uparrow 2 + a[2 \times i] \uparrow 2)$

Запишем рабочую часть программы, не прибегая к модификации адресов:

0	×	$a + 2i - 1$	$a + 2i - 1$	r_1	x_i^2
0	×	$a + 2i$	$a + 2i$	r_2	y_i^2
0	+	r_1	r_2	r_1	$x_i^2 + y_i^2$
0	✓	r_1	0	$z + i$	$z_i := \sqrt{x_i^2 + y_i^2}$

Если мы хотим, чтобы в программе не было переменных команд, то при выполнении первых двух команд удобно иметь значение $H = 2i$, при выполнении последней команды должно быть $H = i$. Этого можно добиться, используя возможность запомнить значение H и затем восстановить его. Будем считать, что перед выполнением рабочей части $H = 2i$ и в ячейке i_1 хранится код команды:

i_1 0 АИ 0 + i 0

Тогда первые три команды рабочей части можно переписать в следующем виде:

6	×	$a - 1$	$a - 1$	r_1	x_i^2	} ($H = 2i$)
6	×	a	a	r_2	y_i^2	
0	+	r_1	r_2	r_1	$x_i^2 + y_i^2$	

Перед выполнением последней команды нужно изменить содержимое индекс-регистра, запомнив его значение. Это можно сделать командой

4 КИ 0 i_1 i_2 $i_2 := 2i$, $H := i$

(точнее, в i_2 засылается команда

i_2 0 АИ 0 $2i$ 0

это равносильно запоминанию $2i$ в ячейке i_2).

Цикл завершится последней из рабочих команд

1 ✓ r_1 0 z $z_i := \sqrt{x_i^2 + y_i^2}$ ($H = i$)

и командой окончания цикла

1 $H < A$ + n p 0001 $H := H + 1$; если $H \leq n$, то на p

Куда передать управление, если $i < n$? Перед началом рабочей части значение индекс-регистра должно равняться $2i$, причем

значение i увеличилось на 1. Значит, нам нужно запомнить новое значение i в ячейке i_1 , восстановить значение $H = 2i$ и увеличить его на 2, т. е. перед рабочей частью нужно поместить команды:

p	4	КИ	0	i_2	i_1	$i_1 := i; H := 2(i-1)$
	2	АИ	0	2	0	$H := H + 2$

Добавив команду засылки начального значения $i = 1$ в H и в i_2 , получим следующую программу:

Программа 3

	0	АИ	0	1	i_2	$H := 1, i_2 := 0$
p	4	КИ	0	i_2	i_1	$i_1 := i, H := i_2 (= 2(i-1))$
	2	АИ	0	2	0	$H := H + 2$
	6	×	$a-1$	$a-1$	r_1	x_i^2
	6	×	a	a	r_2	y_i^2
	0	+	r_1	r_2	r_1	$x_i^2 + y_i^2$
	4	КИ	0	i_1	i_2	$i_2 := 2i, H := i_1 (= i)$
	1	✓	r_1	0	z	$z_i := \sqrt{x_i^2 + y_i^2}$
	1	$H < A$	$+n$	p	0001	$H := H + 1$; если $H < n + 1$, то перейти на p

Цикл можно сократить на одну команду, совместив команду запоминания $H = 2i$ с увеличением этого значения на 2 (при этом начальное значение в i_2 должно равняться 2):

Программа 4

	0	АИ	2	1	i_2	$H := 1, i_2 := 2$
p	4	КИ	0	i_2	i_1	$i_1 := H, H := i_2 (= 2i)$
	6	×	$a-1$	$a-1$	r_1	x_i^2
	6	×	a	a	r_2	y_i^2
	0	+	r_1	r_2	r_1	$x_i^2 + y_i^2$
	4	КИ	2	i_1	i_2	$i_2 := H + 2; H := i_1 (= i)$
	1	✓	r_1	0	z	$z_i := \sqrt{x_i^2 + y_i^2}$
	1	$H < A$	$+n$	p	0001	$H := H + 1$; если $H < n + 1$, то перейти на p

Программу этого варианта можно составить и другим способом: одно из значений i или $2i$ хранить в H , для другого менять команду с помощью переадресации. Так как приращение (к базовому адресу) на $2i$ имеют адреса двух команд, а на i — одной, естественно хранить в индекс-регистре значение $2i$. Перед выполнением цикла переменную команду необходимо сформировать, в цикл включить переадресацию этой команды. Этот вариант программы можно написать следующим образом:

Программа 5

	0	ПЧ	z_2	0	p_1	Формирование для $i=0$
	0	АИ	0	2	0	$H := 2 (i := 1)$
p	0	+M	p_1	z_1	p_1	$i := i + 1$
	6	×	$a-1$	$a-1$	r_1	$\left. \begin{matrix} x_i^2 \\ y_i^2 \end{matrix} \right\} (H=2i)$
	6	×	a	a	r_2	
	0	+	r_1	r_2	r_1	
p_1	0	✓	r_1	0	$z+i$	$x_i^2 + y_i^2$
	1	$H < A$	$2n$	p	0002	$H := H + 2$; если $H < 2n + 2$, то перейти на p

с константами

z_1	0	0	0	0	1
z_2	0	✓	r_1	0	z

Сравним две последние программы: обе состоят из 8 команд; вторая использует две константы (z_1 и z_2), а первая — две дополнительные рабочие ячейки (i_1 и i_2). Все же следует предпочесть вторую программу, так как она будет работать быстрее — у нее в цикле на одну команду меньше.

Пример 4. Составим еще один вариант программы вычисления произведения матрицы на вектор (см. пример 9.2.1) с использованием индекс-регистра. Напомним алгоритм вычисления:

$$y_i = \sum_{j=1}^n a_{ij} x_j, \quad i = 1, 2, \dots, m$$

или

```

for  $i := 1$  step 1 until  $m$  do
  begin  $y[i] := 0$ ;
        for  $j := 1$  step 1 until  $n$  do  $y[i] := y[i] + a[i, j] \times x[j]$ 
  end

```

Рабочую часть цикла мы записывали двумя командами:

$$\begin{array}{l} 0 \times a + (i-1)n + j \quad x + j \quad r \\ 0 + \quad \quad \quad y + i \quad \quad \quad r \quad y + i \end{array}$$

(распределение памяти считаем таким же, как в примере 9.2.1).

Как выгоднее использовать индекс-регистр? Испробуем несколько способов.

1) Введем промежуточную переменную

$$k = (i-1)n + j$$

и выделим три ячейки i_1 , j_1 , k_1 для хранения следующих кодов:

i_1	0	АИ	0	+i	0
j_1	0	АИ	0	+j	0
k_1	0	АИ	0	$(i-1)n + j$	0

В первой из команд рабочей части переменные адреса зависят от различных параметров. Чтобы модифицировать переменные адреса с помощью индекс-регистра, нужно заменить эту команду двумя: одна выберет число по адресу, зависящему от первого параметра, вторая произведет заданное действие (но один из ее операндов уже будет помещен в ячейку с фиксированным адресом). Для полученных команд можно при программировании воспользоваться модификацией адресов, но перед выполнением каждой из команд мы должны послать в индекс-регистр значение соответствующего параметра. Составим программу рабочей части цикла, считая, что в ячейках i_1, j_1, k_1 хранятся нужные нам коды:

```

p      0  КИ  0  k1  0  И := k
      4  ПЧ  a  0  r   r := aij
      4  КИ  1  j1  k1 k := И + 1 (= k + 1), И := j
      2  ×   r  x   r   aijxj
      4  КИ  0  i1  j1 j := И, И := i
      5  +   y  r   y   yi := yi + aijxj

```

В этой программе операции с кодом *КИ* позволяют перед выполнением каждой рабочей команды запомнить в соответствующей ячейке значение индекс-регистра, с которым начала выполняться операция *КИ*, и присвоить *И* новое значение.

Для завершения цикла нужно сравнить значение j с n (верхней границей цикла) и, если $j < n$, увеличить j на 1 — подготовить программу к вычислению очередного произведения $a_{ij}x_j$. Удобнее всего это сделать операцией окончания цикла, послав предварительно в *И* значение j :

```

0  КИ  0  j1  0  И := j
1  И < A + n  p  1  И := И + 1, если И ≤ n, то перейти на

```

Команда *p* посылает в индекс-регистр значение k ; чтобы сохранить полученное последней командой значение $j := j + 1$ (оно понадобится в следующих командах), нужно изменить код в ячейке j_1 . Это можно сделать, записав команду *p* следующим образом:

```

p      4  КИ  0  k1  j1 j := И; И := k

```

Итак, в этом варианте мы получим следующую программу внутреннего цикла:

```

p      4  КИ  0  k1  j1 j := И; И := k
      4  ПЧ  a  0  r   r := aij
      4  КИ  1  j1  k1 k := И + 1, И := j
      2  ×   r  x   r   aijxj
      4  КИ  0  i1  j1 j := И; И := i
      5  +   y  r   y   yi := yi + aijxj
      0  КИ  0  j1  0  И := j
      1  И < A + n  p  1  И := И + 1, если И ≤ n, то перейти на

```

Подготовительные команды должны сформировать значения ячеек i_1 , j_1 и k_1 и очистить ячейку y_i . Но мы не будем завершать составление этой программы, так как очевидно, что она составлена незаконно: мы приходим к выполнению команды p со значением $I = j$, запоминаем это значение, потом восстанавливаем снова.

2) Следующий вариант программы этого же цикла позволяет исключить одну команду:

0	4	$ПЧ$	x	0	r	$r := x_j \ (I = j)$
	4	$КИ$	0	k_1	j_1	$j := I; \ I := k$
	4	$\times$	a	r	r	$a_{ij}x_j$
	4	$КИ$	1	i_1	k_1	$k := I + 1; \ I := i$
	5	$+$	y	r	y	$y_i := y_i + a_{ij}x_j$
j_1	0	$АИ$	0	$+j$	0	Восстановление $I = j$
	1	$I < A$	$+n$	p	1	$I := I + 1$; если $I \leq n$, то на p

(Так как операция $КИ$ запоминает значение индекс-регистра $I_{ст}$ в ячейке с адресом A_j в виде команды $0 \ АИ \ 0 \ I_{ст} \ 0$, то для хранения значения j можно использовать одну из ячеек, в которых помещается выполняемая программа — результатом выполнения команды j_1 и будет засылка j в I , т. е. восстановление значения j).

При составлении подготовительной части цикла нужно помнить, что на команду p мы должны перейти со значением индекс-регистра, равным начальному значению j , т. е. с $I = 1$. Для этого достаточно перед написанными командами поместить следующие:

p_1	1	$ПЧ$	0	0	y	$y_i := 0 \ (I = i)$
	4	$АИ$	0	1	i_1	$i := I, \ I := 1$

Цикл для очередного значения i мы начали выполнять с k , равным $(i - 1)n + 1$ (мы условились, что в ячейке k_1 хранится код $0 \ АИ \ 0 \ (i - 1)n + j \ 0$, перед выполнением цикла по j этот код должен быть заготовлен для $j = 1$). При выполнении внутреннего цикла (по j) значение k каждый раз увеличивается на 1, т. е. при выходе из цикла мы получили $k = (i - 1)n + 1 + n = in + 1$, т. е. то значение, которое будет нужно для выполнения цикла с $i + 1$.

Для окончания цикла по i можно воспользоваться следующими командами:

	0	$КИ$	0	i_1	0	$I := i$
	1	$I < A$	$+m$	p_1	1	$I := I + 1$, если $I \leq m$, то перейти на p_1

Осталось подготовить к работе внешний цикл — заслать начальное значение $i = 1$ в индекс-регистр и $k = 1$ в ячейку k_1 :

	0	$АИ$	1	1	k_1	$k := 1; \ I := 1$

Ячейку i_1 также можно совместить с командой восстановления значения i . Тогда, объединив все команды, получим следующую программу:

Программа 6

	0	АИ	1	1	k_1	$k := 1; I := 1$
p_1	1	ПЧ	0	0	y	$y_i := 0 (I = i)$
	4	АИ	0	1	i_1	$i := I; I := 1$
p	4	ПЧ	x	0	r	$r := x_j (I = j)$
	4	КИ	0	k_1	j_1	$j := I; I := k$
	4	$\times$	a	r	r	$a_{ij}x_j$
	4	КИ	1	i_1	k_1	$k := I + 1; I := i$
	5	+	y	r	y	$y_i := y_i + a_{ij}x_j$
i_1	0	АИ	0	$+j$	0	Восстановление $I = j$
	1	$I < A$	$+n$	p	1	$I := I + 1$; если $I \leq n$, то перейти на p
i_1	0	АИ	0	$+i$	0	Восстановление $I = i$
	1	$I < A$	$+m$	p_1	1	$I := I + 1$; если $I \leq m$, то перейти на p_1

3) Итак, нам удалось составить программу без переменных команд. Но программа внутреннего цикла получилась слишком сложной. Нельзя ли ее упростить?

Внутренний цикл реализует следующие команды:

$$\begin{array}{l} 0 \times a + (i-1)n + j \quad x + j \quad r \\ 0 + \quad \quad \quad y + i \quad \quad \quad r \quad y + i \end{array}$$

Адрес ячейки $y + i$ при выполнении внутреннего цикла не меняется. Программа будет проще, если получать значение суммы в фиксированной ячейке и затем во внешнем цикле пересылать полученную сумму в ячейку $y + i$, т. е. если составлять программу согласно следующему алгоритму, в котором внутренний цикл «очищен» (см. § 5 гл. 7, а также программу 9.2.3) от вычисления адреса переменной y_i :

```
for  $i := 1$  step 1 until  $m$  do
  begin  $s := 0$ ;
    for  $j := 1$  step 1 until  $n$  do
       $s := s + a[i, j] \times x[j]$ ;
       $y[i] := s$ 
    end
```

При выполнении внутреннего цикла удобно все время хранить в индекс-регистре значение j ; запишем программу этого цикла так:

```
 $p$       6  $\times$   $a + (i-1)n$   $x$   $r$   $a_{ij}x_j (I = j)$ 
      0  $+$   $s$   $r$   $s$   $s := s + a_{ij}x_j$ 
      1  $I < A$   $+n$   $p$  1  $I := I + 1$ ; если  $I \leq n$ , то перейти на  $p$ 
```

Внутренний цикл запрограммирован всего тремя командами, но в программе появилась переменная команда p ; команду переадресации поместим после цикла по j перед командами засылки s в y_i и окончания цикла по i :

	0	+M	p	z_1	p	$i := i + 1$
i_1	0	АИ	0	+i	0	$H := i$
	1	ПЧ	s	0	y	$y_i := s$
	1	$H < A$	+m	p_1	1	$H := H + 1$; если $H \leq m$, то перейти на p_1

Осталось написать начальные команды программы: формирование внутреннего цикла для $j = 1$, засылку нуля в s и формирование внешнего цикла для $i = 1$. В результате мы получим такую программу:

Программа 7

	0	ПЧ	z_2	0	p	} $i := 1$
	0	АИ	0	1	0	
p_1	0	ПЧ	0	0	s	$s := 0$
	4	АИ	0	1	i_1	$i := H$; $H := 1$
p	6	×	$a + (i - 1)n$	x	r	$a_{ij}x_j$ ($H = j$)
	0	+	s	r	s	$s := s + a_{ij}x_j$
	1	$H < A$	+n	p	1	$H := H + 1$, если $H \leq n$, то перейти на p
	0	+M	p	z_1	p	$i := i + 1$
i_1	0	АИ	0	+i	0	$H := i$
	1	ПЧ	s	0	y	$y_i = s$
	1	$H < A$	+m	p_1	1	$H := H + 1$, если $H \leq m$, то перейти на p_1

Программа использует следующие константы:

z_1	0	0	+n	0	0
z_2	6	×	a	x	r

Эта программа на одну команду короче, чем программа 6, но использует две дополнительные заготовки. Однако в последнем варианте внутренний цикл состоит только из 3 команд вместо 7 в программе-6, т. е. программа 7 будет выполняться значительно быстрее.

У п р а ж н е н и я. 1. Составить программу линейной интерполяции по таблице (см. пример 7.6.5).

2. Составить программу вычисления разделенных разностей (см. упражнение 9.2.3).

3. Составить программу транспонирования квадратной матрицы.

Глава 11. ПРОГРАММИРОВАНИЕ ЦИКЛОВ С НЕЛИНЕЙНОСТЯМИ

В § 7 гл. 7 говорилось, что основной прием программирования структурных циклов — переадресация — рассчитан на линейную зависимость переменных адресов от параметра цикла и на линейное — по арифметической прогрессии — изменение самого параметра на протяжении всех повторений цикла. Как то, так и другое условие может оказаться нарушенным в практических задачах. Тем не менее известные нам приемы программирования все еще могут оказаться полезными, если сочетать их с другими идеями, в частности, с теми, которые были рассмотрены в § 5 гл. 7.

Нетрудно бывает избавиться от нелинейностей в изменении параметра цикла. Во-первых, многие нелинейные законы изменения в результате тех или иных преобразований могут быть сведены к изменению по арифметической прогрессии. Во-вторых, всегда можно ввести дополнительную переменную, значение которой возрастает на единицу при каждом повторении цикла, принять ее за параметр цикла, а вычисления, определяющие последовательные значения прежнего параметра, включить в рабочую часть цикла. Поясним сказанное примером.

Пример 1. Пусть числа от 1 до n упорядочены в следующей последовательности. Сначала идут все нечетные числа в порядке возрастания, затем все числа, делящиеся на 2, но не делящиеся на 4, также в порядке возрастания, затем числа, делящиеся на 4, но не на 8, и т. д. Параметр цикла k должен пробегать эту последовательность значений. Для каждого значения k элементу массива $a[k]$ требуется присвоить порядковый номер i текущего значения k в этой последовательности. Так, для $n = 10$ параметр цикла k должен последовательно принимать значения: 1, 3, 5, 7, 9, 2, 6, 10, 4, 8 и по ходу выполнения цикла должны быть выполнены присваивания: $a[1] := 1$; $a[3] := 2$; ...; $a[8] := 10$.

На АЛГОЛЕ этот цикл можно записать следующим образом:

```
i := 0; h := 1;
for j := h while j ≤ n do
begin h := 2 × j;
  for k := j step h until n do
  begin i := i + 1; a[k] := i end
end
```

Можно сделать так, чтобы и во внешнем цикле параметр изменялся по арифметической прогрессии:

```

i := 0;
for s := 0 step 1 until log (n)/log (2) do
begin j := 2↑s; h := 2 × j;
      for k := j step h until n do
      begin i := i + 1; a [k] := i end
    end

```

Но если программа на АЛГОЛе составляется лишь затем, чтобы послужить прототипом для машинной программы, составляемой вручную, то и писать ее следует с оглядкой на структуру будущей программы на языке машины. В данном примере переменная *s* нужна только для того, чтобы вычислить очередное значение *j*, а переменная *j*, в свою очередь, лишь для того, чтобы задать начальное значение *k*. Текущее значение *k*, очевидно, целесообразнее всего держать на индекс-регистре, тогда для подготовки цикла по *k* потребуется команда

*i*₁ 0 *AI* 0 + *j* 0 *I* := *j*

Для перехода к очередному значению *j* достаточно выполнить команду

0 + *M* *i*₁ *i*₁ *i*₁ *j* := 2 × *j*

Очередное значение *h* также можно получить удвоением предыдущего. Оно понадобится в качестве приращения значения индекс-регистра, т. е. в виде третьего адреса операции *I* < *A*. Проще всего, по-видимому, хранить очередное значение *h* в виде кода

0 0 0 0 + *h*

в некоторой ячейке *h*₁, а команду с кодом *I* < *A* формировать, складывая подходящую постоянную заготовку с этим кодом. Переменная *s* становится излишней.

С учетом сказанного перепишем АЛГОЛ-программу в виде:

```

i := 0; j := h := 1;
M : h := 2 × h;
      for k := j step h until n do
      begin i := i + 1; a [k] := i end;
      j := 2 × j; if j ≤ n then go to M

```

Однако при составлении машинной программы в ее схему необходимо внести дальнейшие уточнения, которые трудно отразить в тексте на АЛГОЛе. Машинная программа будет такой:

Программа 1

	0	ПЧ	0	0	i	$i := 0$
	0	АИ	1	1	j_1	$j := 1; H := 1$
	0	ПЧ	z_1	0	h_1	$h := 1$
M	0	+M	h_1	h_1	h_1	$h := 2 \times h$
	0	+M	z_2	h_1	p_1	Формирование
p	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
	1	ПЧ	i	0	a	$a[k] := i (H = k)$
p_1	1	$H < A$	$n - h + 1$	p	+h	$H := H + h$, если $H \leq n$, то на p
	0	+M	j_1	j_1	j_1	$j := 2 \times j$
j_1	0	АИ	0	+j	0	$H := j$
	1	$H < A$	$n + 1$	M	0	Если $H < n + 1$, то на M

Текущее содержимое ячейки h_1 равно

$$0 \quad 0 \quad 2^{13} - h \quad 0 \quad +h$$

Константы:

z_1	0	0	7777	0	1
z_2	1	$H < A$	$n + 1$	p	0

В соответствии со вторым рецептом, приведенным в начале раздела, АЛГОЛ-программу данного примера можно было бы записать в виде:

```

k := 1; h := 2;
for i := 1 step 1 until n do
begin a[k] := i; k := k + h;
      if k > n then
begin k := h; h := 2 × h end
end

```

Ей может соответствовать такая машинная программа:

Программа 2

	0	АИ	0	1	0	$k := 1$
	0	ПЧ	z	0	h_1	$h := 2$
	0	ПЧ	0	0	i	$i := 0$
p	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
	1	ПЧ	i	0	a	$a[k] := i (H = k)$
h_1	2	АИ	0	+h	0	$H := H + h$
	1	$H < A$	$n + 1$	p_1	0	Если $H \leq n$, то на p_1
	0	КИ	0	h_1	0	$H := h$
	0	+M	h_1	h_1	h_1	$h := 2h$
p_1	0	—	i	n	0	Если $i < n$, то
	0	П1	0	p	0	перейти на p

с заготовкой

z	2	АИ	0	2	0

Эта программа несколько проще по своей логической схеме и поэтому ее легче составить; она использует на одну заготовку и на одну рабочую ячейку меньше, чем программа 1, но ее внутренний цикл содержит 6 команд вместо трех, так что выполняется она почти в два раза медленнее. Для начинающего программиста программу 2 можно считать вполне приемлемым решением поставленной в этом примере задачи, тогда как программист с опытом должен добиваться более высокого качества программ.

Основной вывод, который можно сделать из рассмотренного примера, состоит в том, что арифметическим операциям над переменными, которые в конечном счете служат для вычисления индексов, в машинных программах должны соответствовать различные комбинации операций над содержимым индекс-регистра, над кодами команд и заготовок.

Обратимся еще к одному примеру, иллюстрирующему эти приемы.

Пример 2. Пусть симметричная матрица A представлена в памяти своим левым нижним углом, расположенным по строкам (см. § 1 гл. 9), так что адрес элемента a_{pq} при $p \geq q$ равен $a + p(p-1)/2 + q$. Требуется найти элементы левого нижнего угла матрицы $B = A^2$, разместив их таким же образом.

Сначала разработаем общую схему вычислений, описав ее на АЛГОЛе. Цикл

```

for  $i := 1$  step 1 until  $n$  do
  for  $j := 1$  step 1 until  $i$  do
    begin  $s := 0$ ;
      for  $k := 1$  step 1 until  $n$  do
         $s := s + a[i, k] \times a[k, j]$ ;
       $b[i, j] := s$  end
    end
  end
end

```

хотя и отражает существо задачи, непригоден потому, что в АЛГОЛе нет треугольных массивов. Предположим поэтому, что массивы a и b описаны как одномерные:

$\text{array } a, b [1 : n \times (n+1)/2]$

Тогда вместо переменной $b[i, j]$, учитывая, что $i \geq j$, следует написать $b[i \times (i-1)/2 + j]$. Сложнее обстоят дела с переменными $a[i, k]$ и $a[k, j]$, так как между значениями переменных i и k , а также k и j возможны любые соотношения. Лобовое решение — воспользоваться вместо $a[i, k]$ записью

$a[\text{if } k \leq i \text{ then } i \times (i-1)/2 + k \text{ else } k \times (k-1)/2 + i]$

— чересчур громоздко, хотя и правильно. К тому же эта запись

мало поможет при переходе к машинной программе. Поэтому разобьем весь диапазон изменения переменной k на участки, так, чтобы в каждом из них выражения для индексов упростились. Придем к следующему варианту цикла по k :

```
for  $k := 1$  step 1 until  $n$  do
   $s := s + (\text{if } k \leq i \text{ then } a[i \times (i-1)/2 + k] \text{ else } a[k \times (k-1)/2 + i])$ 
   $\times (\text{if } k \leq j \text{ then } a[j \times (j-1)/2 + k] \text{ else } a[k \times (k-1)/2 + j])$ .
```

Однако и этот вариант все еще слишком громоздок. Дальнейшее упрощение получим, если (принимая во внимание, что $i \geq j$) разобьем этот цикл на три отдельных цикла:

```
for  $k := 1$  step 1 until  $j$  do
   $s := s + a[i \times (i-1)/2 + k] \times a[j \times (j-1)/2 + k]$ ;
for  $k := j+1$  step 1 until  $i$  do
   $s := s + a[i \times (i-1)/2 + k] \times a[k \times (k-1)/2 + j]$ ;
for  $k := i+1$  step 1 until  $n$  do
   $s := s + a[k \times (k-1)/2 + i] \times a[k \times (k-1)/2 + j]$ 
```

Внешне запись, может быть, усложнилась, но нам удалось избавиться от условных выражений. Рассмотрим теперь четыре индексных выражения: $i \times (i-1)/2 + k$, $k \times (k-1)/2 + i$, $j \times (j-1)/2 + k$ и $k \times (k-1)/2 + j$. По опыту § 5 гл. 7 мы знаем, что вместо прямого вычисления значений этих выражений для каждого значения k выгоднее использовать предыдущее значение, добавив к нему нужное приращение. Для первого и третьего выражений это приращение равно 1, а для второго и четвертого оно равно старому значению k . Заметим также, что при $k = i$ совпадают значения первого и второго выражений, а при $k = j$ — третьего и четвертого. С учетом этих замечаний рассматриваемый участок программы можно переписать в виде:

```
 $ik := i \times (i-1)/2$ ;  $kj := j \times (j-1)/2$ ;
for  $k := 1$  step 1 until  $j$  do
begin  $ik := ik + 1$ ;  $kj := kj + 1$ ;  $s := s + a[ik] \times a[kj]$  end;
for  $k := j+1$  step 1 until  $i$  do
begin  $ik := ik + 1$ ;  $kj := kj + k - 1$ ;  $s := s + a[ik] \times a[kj]$  end;
for  $k := i+1$  step 1 until  $n$  do
begin  $ik := ik + k - 1$ ;  $kj := kj + k - 1$ ;  $s := s + a[ik] \times a[kj]$  end
```

Для задания начальных значений переменных ik и kj заведем еще две переменные si и sj , значения которых, равные $i(i-1)/2$ и $j(j-1)/2$ соответственно, будем вырабатывать тем же способом. Еще одна переменная ij потребуется на роль индекса массива b .

Она изменяется особенно просто — увеличивается каждый раз на 1. Вся АЛГОЛ-программа принимает вид

```

si := ij := 0;
for i := 1 step 1 until n do
begin
  si := si + i - 1; sj := 0;
  for j := 1 step 1 until i do
  begin
    ik := si; kj := sj + j - 1; s := 0;
    for k := 1 step 1 until j do
    begin
      ik := ik + 1; kj := kj + 1;
      s := s + a [ik] × a [kj] end;
    for k := j + 1 step 1 until i do
    begin
      ik := ik + 1; kj := kj + k - 1;
      s := s + a [ik] × a [kj] end;
    for k := i + 1 step 1 until n do
    begin
      ik := ik + k - 1; kj := kj + k - 1;
      s := s + a [ik] × a [kj] end;
    ij := ij + 1; b [ij] := s
  end
end
end

```

Проделанные преобразования привели к тому, что над переменными, от которых явно или косвенно зависят индексные выражения, выполняются только операции сложения.

Теперь уже не сложно составить программу в кодах М-20. Ясно, что присваивания переменным *ik* и *kj* реализуются в машинной программе в виде переадресации или формирования адресов в командах, отвечающих операциям $a[ik] \times a[kj]$. Переменным *k*, *si* и *sj* должны соответствовать коды, с помощью которых осуществляются эти переадресации и формирования, переменным *i* и *j* — константы, с помощью которых переадресуются эти коды. Целесообразно также, чтобы текущие значения *i*, *j* и *k* в машинной программе были на 1 меньше значений аналогичных переменных в АЛГОЛ-программе.

У п р а ж н е н и е 1. Составить по полученной АЛГОЛ-программе программу для машины М-20.

Глава 12. ПРОГРАММИРОВАНИЕ ЛОГИЧЕСКИХ ВЫРАЖЕНИЙ

§ 1. Представление логических значений в машине

Логические значения — «истина» (**true**) и «ложь» (**false**) в отличие от чисел не имеют стандартного представления. Программистов волнует выбрать такое представление, какое ему представляется удобным в той или иной задаче. Для представления значения одной логической переменной достаточно выделить один любой разряд какой-либо одной ячейки. Таким образом, в одной ячейке можно хранить значения до 45 логических переменных. Однако, если число логических переменных, используемых в задаче, невелико, то можно отвести по целой ячейке под значение каждой из этих переменных. Наконец, для кратковременного хранения одного логического значения можно использовать регистр признака ω . Чаще всего значение ω , выработанное некоторой операцией, используется лишь следующей операцией, но иногда между выработкой признака ω и его использованием выполняется несколько операций. Это возможно при условии, что промежуточные операции не меняют значение этого признака.

Во всех случаях необходимо соглашение о том, какое именно логическое значение изображается тем или иным содержимым разряда, ячейки или регистра ω . В случае одного двоичного разряда обычно принимают, что 1 изображает логическое значение **true**, а 0 — значение **false**. Как мы увидим, это соглашение отражается в принятых наименованиях операций машины М-20. Если для хранения логического значения отведена целая ячейка, то перед программистом открывается большой выбор возможностей. Можно, например, условиться, что значение **true** изображается кодом, содержащим 1 во всех разрядах ячейки, а значение **false** — кодом, составленным только из нулей. Можно принять, что значению **true** отвечает 1, а значению **false** — 0 в одном определенном разряде, содержимое же остальных разрядов должно быть равным нулю. Можно согласиться, что значение **true** изображается тем же кодом, что число 1, а значение **false** — кодом числа 0. В разных программах тот или иной вариант может оказаться более удобным,

чем другие. Во всех упомянутых вариантах существуют коды (например, код 000 7777 0000 0000), ничего не изображающие при выбранном соглашении. Поэтому можно принять соглашение несколько иного рода — какой-то определенный код изображает одно из логических значений (например, код из 45 нулей — значение **false**), любой другой код — противоположное значение (**true**). Впрочем, наличие кодов, не изображающих ни одно из логических значений, может оказаться полезным в задачах, где допускаются вопросы, на которые кроме определенных ответов — да (**true**) и нет (**false**) — возможны и неопределенные ответы типа «не знаю» или «неизвестно». Кодам, не изображающим ни **true**, ни **false**, как раз и может быть приписан такой неопределенный смысл.

Значению признака ω естественно сопоставить то или иное логическое значение на протяжении не всей программы, а лишь каждой отдельной группы команд, где вырабатывается и используется этот признак. Так, из сказанного в § 3 гл. 7 следует, что при проверке отношения $x < y$ значению $\omega = 1$, выработанному в результате выполнения операции

$$0 - x \ y \ 0$$

естественно сопоставить значение **true**, а при проверке отношения $x \leq y$, осуществляемой с помощью операции

$$0 - y \ x \ 0$$

тому же значению $\omega = 1$ следует сопоставить значение **false**. Мы сталкивались и со многими другими примерами истолкования значения признака ω .

Перейдем теперь от изображения отдельных логических значений к изображению логических массивов. Если под каждое логическое значение отводить целую ячейку, то никаких новых особенностей по сравнению с числовыми (арифметическими) массивами логические массивы не представляют. Однако, когда в программе приходится иметь дело с логическими массивами, особенно если размеры этих массивов велики, то выделять отдельную ячейку под каждый элемент массива становится чересчур расточительно. Обычно их упаковывают так, чтобы в ячейке размещалось до 45 элементов логического массива.

Если требуется разместить в памяти одномерный логический массив (логический вектор), заданный описанием

Boolean array $b[1:n]$

то при $n \leq 45$ достаточно выделить под него одну ячейку. Естественно, чтобы соседним элементам $b[i]$ и $b[i+1]$ отвечали соседние разряды ячейки. Но и при соблюдении этого требования остается довольно большая свобода выбора — элементу $b[i]$ можно поставить в соответствие $(c+i)$ -й разряд ячейки, где $0 \leq c \leq 45 - n$,

или же $(c - i)$ -й разряд при $n < c \leq 46$ (напомним, что разряды ячейки принято нумеровать справа налево). Чаще всего значения элемента $b[i]$ размещают в i -м или в $(46 - i)$ -м (т. е. i -м слева разряде). Первый способ немного предпочтительнее.

Если для того же массива $n > 45$, то одной ячейки мало. Обычно для хранения элементов такого массива выделяют группу из m последовательных ячеек, где $45(m - 1) < n \leq 45m$. Элементу $b[1]$ отвечает 1-й разряд первой ячейки группы, элементу $b[45]$ — 45-й разряд этой ячейки, элементу $b[46]$ — 1-й разряд следующей ячейки и т. д.

Для многомерных логических массивов следует считаться с конкретными значениями границ индексов и размещать массив так, чтобы, во-первых, не расходовалось чересчур много ячеек. Во-вторых, чтобы не слишком усложнялось определение места хранения элемента массива в зависимости от значений его индексов. В частности, если при изменении значения лишь одного из индексов существуют различные элементы массива, размещаемые в пределах одной ячейки, то все такие элементы должны занимать минимальное число ячеек (одну ячейку, если это возможно, т. е. если число значений индекса не превосходит 45).

Пусть, например, логический массив задан описанием

Boolean array $q[1:m, 1:n]$

Если $mn \leq 45$, то весь массив можно разместить в одной ячейке, так что элемент $q[i, j]$ будет размещаться в разряде с номером либо $(i - 1)n + j$, либо (для другого варианта размещения) $(j - 1)m + i$. Если $mn > 45$, но $n \leq 45$ (или $m \leq 45$), то массив можно и следует разместить так, чтобы каждая его строка (соответственно, столбец) располагалась в одной ячейке. Если, кроме того, $n \leq 22$, то в одной ячейке можно разместить две или более строк, но это заметно усложнит переход от одной строки к другой. Иногда этот переход будет связан с изменением адреса ячейки, иногда — с изменением номера разряда в пределах одной ячейки. Поэтому при $\max(m, n) \leq 45$ целесообразно разместить массив так, чтобы при $m \leq n$ каждая его строка занимала отдельную ячейку, а все элементы одного столбца располагались в разрядах (разных ячеек) с одним и тем же номером. Если же $45 \geq m > n$, то для экономии места в памяти целесообразнее размещать в отдельных ячейках не строки, а столбцы массива.

У п р а ж н е н и я. 1. Пусть значение **true** изображается в программе так же, как число 1, а **false** — как число 0; x, y, z — логические переменные, a, b — целочисленные переменные. Написать команды, реализующие операторы:

- а) $y := \neg x$, б) $z := x \wedge y$, в) $z := x \vee y$, г) $z := x \supset y$,
 д) $z := x \equiv y$, е) $z := a < b$, ж) $z := \text{if } x \text{ then } a = b \text{ else } y$.

2. Предложить экономный способ размещения в памяти массива с описанием

Boolean array $\lambda[1:4, -2:2, 0:10]$.

§ 2. Логические операции и операции сдвига в системе команд машины М-20

В машине М-20, так же как в подавляющем большинстве других вычислительных машин, нет операций над отдельными логическими значениями. Оперантами так называемых *логических* операций машины М-20 являются целые машинные слова. Однако эти операции выполняются поразрядно — значение i -го разряда результата для любого i зависит только от значений i -х разрядов операндов. Если принять, как это обычно делается, что двоичные цифры 0 и 1 изображают логические значения *false* и *true*, то в каждом разряде выполняется некоторая логическая операция, по которой получает свое наименование машинная операция над целыми словами.

Логическое умножение (конъюнкция; символическое обозначение $\wedge$; восьмеричный код 55). Цифра каждого разряда результата, засылаемого по третьему адресу, определяется только в зависимости от цифр того же разряда операндов, вызванных по первому и второму адресу, по следующему правилу:

$$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1.$$

Иными словами, результат логического умножения содержит единицы лишь в тех двоичных разрядах, в которых имели единицы оба операнта. Если результат состоит из нулей во всех разрядах, то вырабатывается признак $\omega = 1$, в противном случае $\omega := 0$.

Пример:

При выполнении операции $\wedge$ были вызваны слова:

по первому адресу: 000 110 010 101 ... 111

по второму адресу: 010 101 100 111 ... 101

Результат операции: 000 100 000 101 ... 101, $\omega = 0$.

Логическое умножение широко используется для выделения из машинных слов некоторых определенных компонент. Так, например, если нужно из некоторой команды выделить ее код операции и третий адрес, то можно воспользоваться операцией логического умножения, указав в первом адресе команды адрес ячейки ОЗУ, содержащей преобразуемую команду, а во втором адресе — адрес ячейки, содержащей вспомогательную константу:

077 0000 0000 7777

(единицы во всех разрядах кода операции и третьего адреса, нули в разрядах первого и второго адреса и всех признаков модификации). Если преобразуемая команда была

π КОП A_1 A_2 A_3

то в результате логического умножения получим код:

0 КОП 0 0 A_3

Логическое сложение (дизъюнкция; $\vee$; 75). Цифра каждого разряда результата, засылаемого по третьему адресу, определяется в зависимости от цифр того же разряда операндов, вызванных по первому и второму адресу, по правилу:

$$0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1.$$

Другими словами, результат логического сложения содержит единицы в тех разрядах, где они имелись хотя бы у одного из операндов. Вырабатывается признак $\omega = 1$, если во всех разрядах результата будут нули.

Пример:

При выполнении операции $\vee$ были вызваны слова:

по первому адресу:	000 110 010 101 ... 0 111
по второму адресу:	010 101 100 111 ... 1 101
Результат операции:	010 111 110 111 ... 1 111, $\omega = 0$.

Логическое сложение можно использовать для формирования машинных слов из их составных частей. Так, например, если две ячейки памяти содержат слова

$$0 \text{ КОП } 0 \quad 0 \quad A_3$$

и

$$\pi \quad 0 \quad A_1 \quad A_2 \quad 0$$

то в результате выполнения операции логического сложения над этими словами можно сформировать команду:

$$\pi \text{ КОП } A_1 \quad A_2 \quad A_3$$

Сравнение ($\neq$; 15). При выполнении этой частично уже знакомой нам операции сравниваются соответствующие разряды слов, вызываемых по первому и второму адресам. Если значения i -х разрядов одинаковы (оба — нули или оба — единицы), то в i -й разряд результата записывается нуль; если же i -е разряды различны, то в i -й разряд результата записывается единица.

Иначе говоря, сравнение — это поразрядная операция (отрицание эквивалентности), выполняемая по правилу:

$$0 \neq 0 = 0, \quad 0 \neq 1 = 1, \quad 1 \neq 0 = 1, \quad 1 \neq 1 = 0.$$

Результат засылается в ОЗУ по третьему адресу.

Если все разряды первого операнда совпадают со всеми разрядами второго операнда, то во всех разрядах результата будут нули. В этом случае, как мы уже знаем, признак ω получает значение 1. Если хоть один разряд результата отличен от нуля, то происходит присваивание $\omega = 0$.

Если все разряды одного из операндов равны нулю, то результат совпадает с другим операндом. Если все разряды одного из операндов равны единице, то результат получается поразрядным

отрицанием (инверсией) другого операнта, т. е. заменой значения каждого из его разрядов на противоположное. Если ни в одном из разрядов не встречается 1 сразу у обоих оперантов, то результат операции сравнения совпадает с результатом логического сложения тех же оперантов.

Пр и м е р. При выполнении операции $\neq$ были вызваны слова:

по первому адресу:	101 110 111 011 ... 10 111
по второму адресу:	111 000 001 110 ... 01 110
Результат операции:	<u>010 110 110 101 ... 11 001</u>

У п р а ж н е н и я. 1. Составить таблицы логического умножения, сложения и неэквивалентности для групп из трех соседних двоичных разрядов, записанных восьмеричными цифрами.

2. Какие логические операции выполняются в каждом из разрядов при выполнении машинных операций $\wedge$, $\vee$ и $\neq$, если принять, что цифра 0 изображает логическое значение «истина», а 1 — «ложь»?

3. Условимся, что цифрами 1 в машинном слове отмечено некоторое подмножество множества всех разрядов слова. Каким теоретико-множественным операциям соответствуют машинные операции $\wedge$, $\vee$ и $\neq$? Каким комбинациям этих машинных операций соответствуют другие употребительные теоретико-множественные операции?

Из сказанного выше о представлении в машине логических маск-слов ясно, что весьма желательно уметь передвигать содержимое одного из разрядов машинного слова в другой разряд. Удовлетворению этой потребности содействуют операции сдвига. Как мы увидим впоследствии, эти операции полезны во многих случаях, так что программирование логических выражений — далеко не единственное их применение.

В машине имеются четыре операции сдвига. При выполнении всех этих операций номер ячейки, в которой находится сдвигаемый код (второй оперант), указывается во втором исполнительном адресе команды. Код, полученный в результате сдвига, посылается по третьему исполнительному адресу. Величина и направление сдвига определяется первым оперантом — семиразрядным двоичным числом k , задаваемым или в первом исполнительном адресе выполняемой команды, или в ячейке, номер которой указан в этом адресе (способ задания этого операнта определяется кодом операции). Если это число $k \geq 1\ 000\ 000_2$, то сдвиг происходит влево на число разрядов n , равное $k - 1\ 000\ 000_2$ ($k - 100_8$). Если $k \leq 100_8$, сдвиг происходит вправо на $n = 100_8 - k$ разрядов. Если считать, что сдвиг влево задается положительным числом, а сдвиг вправо — числом отрицательным, то $k = 100_8 + n$, где n — число разрядов сдвига.

Сдвиг машинного слова по первому адресу (символическое обозначение $\leftarrow SA$; восьмеричный код операции 54). При выполнении этой операции все 45 разрядов машинного слова как единое целое сдвигаются влево или вправо на указанное число разрядов. При

сдвиге часть разрядов выходит за пределы разрядной сетки ячейки и теряется; место, освободившееся от сдвинутых разрядов, заполняется нулями. Величина и направление сдвига определяются содержимым семи младших разрядов ($25 \div 31$) первого исполнительного адреса; содержимое пяти старших разрядов A_1' ($32 \div 36$) не влияет на выполнение команды. Если после сдвига во всех 45 разрядах результата окажутся нули, вырабатывается признак $\omega = 0$.

Примеры. Пусть в ячейке a находится слово, состоящее из единиц во всех 45 разрядах:

$a \quad 777 \quad 7777 \quad 7777 \quad 7777$

После выполнения операции

$0 \leftarrow CA \quad 0105 \quad a \quad b$

в ячейку b зашлется слово, сдвинутое на пять разрядов влево, т. е.

111 111 111 111 ... 111 100 100
(777 7777 7777 7740)

Признак ω получит значение 0.

Если в ячейке 3174 находится слово

100 111 010 110 101 011 010 110 011 001 100 101 011 110 011
(472 6532 6314 5363)

и мы хотим сдвинуть его на восемь разрядов вправо и записать результат в ячейку 0125, то это можно сделать командой

$0 \leftarrow CA \quad 0070 \quad 3174 \quad 0125$
($k = 100_8 - 010_8 = 070_8$).

В ячейке 0125 окажется слово

000 000 001 001 110 101 101 010 110 101 100 110 011 001 010
(001 1655 2654 6312), $\omega = 0$.

Сдвиг машинного слова по порядку числа ($\leftarrow SP; 74$). Операция выполняется так же, как предыдущая, но величина и направление сдвига определяются содержимым 37 $\div$ 43 разрядов кода, вызываемого по первому исполнительному адресу команды. Напомним, что при записи чисел в этих разрядах ячейки записывается код порядка числа — отсюда и происходит название операции (при записи команды в 37 $\div$ 43 разрядах помещаются код операции и признак модификации третьего адреса). Условия выработки признака ω — такие же, как в предыдущей операции.

Пример. Результат операции

$0 \leftarrow SP \quad 2436 \quad 3174 \quad 0125$

будет таким же, как в предыдущем примере, если в ячейке 3174 хранится то же слово, а в ячейке 2436 — слово

6 70 7126 3542 7130

или любое другое слово, имеющее вид команды с КОП = 70 и $\alpha_3 = 0$). Сдвиг определяется содержимым $37 \div 43$ разрядов этого слова, т. е. числом $k = 070_8$; $n = k - 100_8 = -010_8$, т. е. сдвиг произойдет на 8 разрядов вправо.

Сдвиг мантиссы по первому адресу ($\leftarrow MA$; 14). При выполнении этой операции сдвигаются как единое целое только $1 \div 36$ разряды второго операнта (адресная часть команды или мантисса числа). Содержимое 9 старших разрядов ($37 \div 45$) остается неизменным. При сдвиге часть разрядов выходит за пределы 36-разрядной сетки и теряется; место, освободившееся от сдвинутых разрядов, заполняется нулями. Если после сдвига в $1 \div 36$ разрядах результата находятся нули, то вырабатывается $\omega = 1$.

Как и в операции $\leftarrow SA$, величина и направление сдвига определяются содержимым семи младших разрядов первого исполнительного адреса.

П р и м е р ы. Пусть в ячейке a находится слово, состоящее из единиц во всех 45 разрядах:

7 77 7777 7777 7777

и выполняется команда

0 $\leftarrow MA$ 0075 a b .

Содержимое семи младших разрядов A_1

$k = 075_8 = 100_8 - 003$,

т. е. сдвиг произойдет вправо на 3 разряда и в ячейку b зашлется слово

7 77 0777 7777 7777

старшие 9 разрядов результата совпадают с $37 \div 45$ разрядами сдвигаемого слова; при сдвиге на 3 разряда вправо в $34 \div 36$ разрядах появятся три нуля, которые и записываются одной восьмичисленной цифрой 0), $\omega = 0$.

Выполняется операция

0 $\leftarrow MA$ 0050 p q

в ячейке p находится команда

1 ПЧ a 0 b

Сдвиг произойдет вправо на $n = 050_8 - 100_8 = -030_8 = -24_{10}$ разрядов, т. е. на два адреса (один адрес в команде занимает 12_{10} разрядов). Старшие 9 разрядов (код операции и признаки модификации)

результата будут теми же, что у сдвигаемого слова, т. е. в ячейке окажется команда

1 ПЧ 0 0 а

при этом $\omega = 0$, если $a \neq 0$.

Сдвиг мантиссы по порядку числа ($\leftarrow$ МП; 34). Как единое целое, сдвигаются $1 \div 36$ разряды слова, записанного по второму исполнительному адресу. Величина и направление сдвига задаются содержимым $37 \div 43$ разрядов слова, хранящегося по первому исполнительному адресу. Вырабатывается признак $\omega = 1$, если в $1 \div 36$ разрядах результата находятся нули.

Пример. Если в ячейке с хранится слово

4 50 l m n

то тот же результат, что в предыдущем примере, можно получить, выполняя операцию

0 $\leftarrow$ МП с р q

§ 3. Приемы программирования логических выражений

В АЛГОЛе, как и в большинстве других языков программирования, логические выражения имеют два основных применения — в условиях условных выражений и операторов и в правых частях операторов присваивания логической переменной, простой или с индексами. В зависимости от того, какой из этих случаев имеет место, приемы программирования логических выражений видоизменяются. Рассмотрим сначала, как можно осуществить присваивание простой логической переменной при условии, что под каждую логическую переменную отводится своя ячейка.

Пусть при этом значение **true** изображается машинным словом, содержащим 1 в некотором фиксированном разряде и нули в остальных разрядах, а значение **false** — словом, состоящим сплошь из нулей. Пусть, как для чисел, обозначение $\langle \text{true} \rangle$ заменяет адрес ячейки, содержащей машинное представление константы **true**. Пусть, наконец, буквы x, y, z заменяют адреса ячеек, отведенных для хранения текущих значений логических переменных, обозначаемых в формулах теми же буквами. Естественно, что присваивание $z := x$ реализуется командой пересылки:

0 ПЧ x 0 z

Выражение, содержащее только логические операции, реализуется с помощью машинных логических операций. Присваивание $z := x \wedge y$ программируется с помощью команды

0 $\wedge$ x y z

Аналогично, команда

$$0 \vee x \ y \ z$$

реализует присваивание $z := x \vee y$, а команда

$$0 \neq x \ y \ z$$

— присваивание $z := \neg (x \equiv y)$.

Хотя в машине М-20 нет операций отрицания ($\neg$), импликации ($\supset$) и эквивалентности ($\equiv$), программируются они почти столь же просто, исходя из элементарных тождеств алгебры высказываний.

$$\begin{aligned} \neg x &\Leftrightarrow \neg (x \equiv \text{true}), \\ x \supset y &\Leftrightarrow \neg x \vee y, \\ x \equiv y &\Leftrightarrow \neg (\neg (x \equiv y)). \end{aligned}$$

Таким образом, команда

$$0 \neq x \ \langle \text{true} \rangle \ z \ z := \neg (x \equiv \text{true})$$

реализует присваивание $z := \neg x$, команды

$$\begin{aligned} 0 &\neq x \ \langle \text{true} \rangle \ z \quad \neg x \\ 0 \vee z \quad y \quad z \quad z &:= \neg x \vee y \end{aligned}$$

— присваивание $z := x \supset y$, а команды

$$\begin{aligned} 0 &\neq x \quad y \quad z \quad \neg (x \equiv y) \\ 0 &\neq z \ \langle \text{true} \rangle \ z \ z := \neg (\neg (x \equiv y)) \end{aligned}$$

— присваивание $z := x \equiv y$.

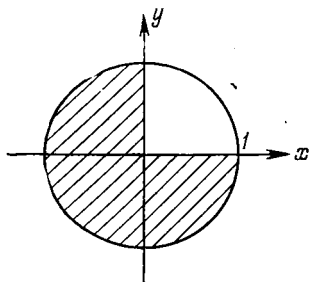
Присваивание, в правой части которого стоит отношение, программируется с помощью трех команд. Первая команда вырабатывает в зависимости от истинности отношения то или иное значение признака ω , после чего команда условного перехода по значению ω , соответствующему значению **true** отношения, заставляет обойти третью команду и одновременно осуществляет присваивание этого значения переменной левой части. Третья команда присваивает этой переменной значение **false**. Так, например, присваивание $z := a < b$, где a и b — числовые переменные, реализуется командами

$$\begin{array}{llll} p & 0 & - & a \ b \ 0 \text{ если } a < b, \text{ то } \omega := 1 \\ p+1 & 0 & П1 & \langle \text{true} \rangle p+3 \ z \ z := \text{true, если } \omega = 1, \text{ то на } p+3 \\ p+2 & 0 & ПЧ & 0 \ 0 \ z \ z := \text{false} \end{array}$$

а присваивание $z := a \neq b$ — командами

$$\begin{array}{llll} p & 0 & \neq & a \ b \ 0 \text{ если } a \neq b, \text{ то } \omega := 0 \\ p+1 & 0 & П0 & \langle \text{true} \rangle p+3 \ z \ z := \text{true, если } \omega = 0, \text{ то на } p+3 \\ p+2 & 0 & ПЧ & 0 \ 0 \ z \ z := \text{false} \end{array}$$

Любое логическое выражение, не содержащее в своем составе логических переменных с индексами и условий, может быть запрограммировано с помощью перечисленных приемов.



Пример 1. Пусть логической переменной z требуется присвоить значение **true**, если точка с координатами x и y принадлежит области, показанной на рис. 12.3.1 (включая ее границу), и **false** — в противном случае.

На АЛГОЛе решение дается следующим оператором:

Рис. 12.3.1. К примеру 12.3.1. $z := x \uparrow 2 + y \uparrow 2 \leq 1 \wedge (x \leq 0 \vee y \leq 0)$.

Чтобы составить машинную программу, введем вспомогательные логические переменные u , v и w и перепишем этот оператор в виде

$$u := x \uparrow 2 + y \uparrow 2 \leq 1; \quad v := x \leq 0; \quad w := y \leq 0; \\ z := u \wedge (v \vee w)$$

Приходим к следующей программе.

Программа 1									
p	0	$\times$	x	x	u	x^2			
	0	$\times$	y	y	v	y^2			
	0	+	u	v	u	$x^2 + y^2$			
	0	—	$\langle 1 \rangle$	u	0	Если $x^2 + y^2 \leq 1$, то $w := 0$			
	0	ПО	$\langle \text{true} \rangle$	p_1	u	$u := \text{true}$; если $w = 0$, то на p_1			
p_1	0	ПЧ	0	0	u	$u := \text{false}$			
	0	—	0	x	0	Если $x \leq 0$, то $w := 0$			
	0	ПО	$\langle \text{true} \rangle$	p_2	v	$v := \text{true}$; если $w = 0$, то на p_2			
p_2	0	ПЧ	0	0	v	$v := \text{false}$			
	0	—	0	y	0	Если $y \leq 0$, то $w := 0$			
	0	ПО	$\langle \text{true} \rangle$	p_3	w	$w := \text{true}$; если $w = 0$, то на p_3			
p_3	0	ПЧ	0	0	w	$w := \text{false}$			
	0	$\vee$	v	w	z	$v \vee w$			
	0	$\wedge$	u	z	z	$z := u \wedge (v \vee w)$			

Однако такой способ программирования чересчур прямолинеен и поэтому неэкономичен. К значительно лучшим результатам можно прийти, если воспользоваться аппаратом условных выражений и операторов. Любое присваивание вида

$$z := B$$

где B — логическое выражение, можно заменить эквивалентным присваиванием

$$z := \text{if } B \text{ then true else false}$$

Фактически мы уже прибегали к этому приему, когда B было отношением. От оператора присваивания с условным выражением в правой части можно перейти к условному оператору:

if B then $z := \text{true}$ else $z := \text{false}$

Приемы программирования таких операторов частично уже были рассмотрены в § 3 гл. 7. Там было отмечено, что для приближения к машинному языку условный оператор

if B then S_1 else S_2 ,

где S_1 и S_2 — произвольные операторы, целесообразно преобразовать к виду

**if B then go to L ;
 S_2 ; go to M ;
 L : S_1 ;
 M :**

а если желательно сохранить относительное расположение операторов S_1 и S_2 (при этом легче проследить соответствие между исходной и преобразованной программой), — то к виду

**if $\neg B$ then go to L ;
 S_1 ; go to M ;
 L : S_2 ;
 M :**

Условия **if B then** и **if $\neg B$ then** можно запрограммировать совершенно одинаково, различаться будут лишь операции условного перехода в зависимости от выработанного значения признака ω .

Итак, программирование любого логического выражения в правой части оператора присваивания или в условии произвольного условного выражения или оператора можно свести к программированию того же выражения в условии оператора вида

if B then go to L (1)

Такие операторы программируются очень просто благодаря наличию в машинном языке операций условного перехода.

Если B — простая логическая переменная, то оператор (1) программируется с помощью команд

0 $\neq$ B <true> 0 Если $B \equiv \text{true}$, то ($\omega := 1$)
 0 П1 0 L 0 перейти на L

или

0 $\neq$ B 0 0 Если $B \equiv \text{true}$, то ($\omega := 0$)
 0 П0 0 L 0 перейти на L

где буква L заменяет адрес, соответствующий метке, обозначенной той же буквой. Второй вариант имеет то преимущество, что способы

представления значения **true** более разнообразны, тогда как значение **false** обычно изображается нулевым словом (см. § 1).

Приемы программирования оператора (1), если B — отношение, были рассмотрены в § 3 гл. 7. Перейдем к случаю, когда значение B вырабатывается в результате выполнения одной из логических операций. Здесь открываются две возможности. Во-первых, любая логическая операция вырабатывает некоторое значение признака ω , которое может быть использовано в последующей операции условного перехода на метку L . Например, оператор

if $x \wedge y$ then go to L

может быть реализован командами

0	$\wedge$	x	y	0	Если $x \wedge y \equiv \text{true}$, то ($\omega := 0$)
0	ПО	0	L	0	перейти на L

Здесь, как и выше, мы предполагаем, что значению **false** соответствует нулевое слово.

Во-вторых, оператор (1) может быть подвергнут дальнейшим преобразованиям, облегчающим его программирование. Так, если выражение B имеет вид $B1 \wedge B2$, то оператор

if $B1 \wedge B2$ then go to L

может быть заменен операторами

if $\neg B1$ then go to $L1$;
if $B2$ then go to L ;

$L1$:

Оператор

if $B1 \vee B2$ then go to L

эквивалентен операторам

if $B1$ then go to L ;
if $B2$ then go to L

Оператор

if $B1 \supset B2$ then go to L

допускает замену операторами

if $\neg B1$ then go to L ;
if $B2$ then go to L

Эти преобразования целесообразны, когда выражения $B1$ и $B2$ являются отношениями или могут быть сведены к отношениям аналогичными преобразованиями.

Пример 1 (продолжение). Применим перечисленные выше приемы к программированию оператора

$z := x \uparrow 2 + y \uparrow 2 \leq 1 \wedge (x \leq 0 \vee y \leq 0)$

Сначала получаем:

if $x \uparrow 2 + y \uparrow 2 \leq 1 \wedge (x \leq 0 \vee y \leq 0)$ then $z := \text{true}$
else $z := \text{false}$

Затем:

if $x \uparrow 2 + y \uparrow 2 \leq 1 \wedge (x \leq 0 \vee y \leq 0)$ then go to L ;

$z := \text{false}$; go to M ;

L : $z := \text{true}$;

M :

Наконец,

if $x \uparrow 2 + y \uparrow 2 > 1$ then go to $L1$;

if $x \leq 0$ then go to L ;

if $y \leq 0$ then go to L ;

$L1$: $z := \text{false}$; go to M ;

L : $z := \text{true}$;

M :

Отсюда без ухищрений получаем следующую программу:

Программа 2

	0	×	x	x	u	x^2
	0	×	y	y	v	y^2
	0	+	u	v	u	$x^2 + y^2$
	0	—	$\langle 1 \rangle$	u	0	Если $x^2 + y^2 > 1$, то
	0	$\Pi 1$	0	$L1$	0	перейти на $L1$
	0	—	0	x	0	Если $x \leq 0$, то
	0	$\Pi 0$	0	L	0	перейти на L
	0	—	0	y	0	Если $y \leq 0$, то
	0	$\Pi 0$	0	L	0	перейти на L
$L1$	0	$\Pi Б$	0	M	z	$z := \text{false}$; перейти на M
L	0	$\Pi Ч$	$\langle \text{true} \rangle$	0	z	$z := \text{true}$
M						

Здесь сэкономлены 3 команды по сравнению с программой 1, но можно сэкономить еще одну команду, если совместить с присваиваниями все операции условного перехода. Хороший вариант программы таков:

Программа 3

	0	×	x	x	z	x^2
	0	×	y	y	u	y^2
	0	+	z	u	z	$x^2 + y^2$
	0	—	$\langle 1 \rangle$	z	0	Если $x^2 + y^2 > 1$, то $\omega := 1$
	0	$\Pi 1$	0	M	z	$z := \text{false}$; если $\omega = 1$, то на M
	0	—	0	x	0	Если $x \leq 0$, то $\omega := 0$
	0	$\Pi 0$	$\langle \text{true} \rangle$	M	z	$z := \text{true}$; если $\omega = 0$, то на M
	0	—	0	y	0	Если $y \leq 0$, то $\omega := 0$
	0	$\Pi 0$	$\langle \text{true} \rangle$	M	z	$z := \text{true}$; если $\omega = 0$, то на M
	0	$\Pi Ч$	0	0	z	$z := \text{false}$
M						

Преимущество программы 3 перед программой 1 не только и не столько в том, что она на 4 команды короче, а в том, что в ряде случаев нет нужды выполнять многие из оставшихся в программе команд.

У п р а ж н е н и е 1. Для различных областей, на которые окружность $x^2 + y^2 = 1$ и прямые $x = 0$ и $y = 0$ разбивают плоскость, подсчитать число команд, выполняемых во время выполнения программ 1, 2 и 3.

П р и м е р 2. Проверить, имеются ли в данном массиве чисел отрицательные. Приведем сначала полную запись алгоритма решения этой задачи на АЛГОЛе.

```
begin integer n; ввод (n);
  begin array x [1 : n]; Boolean t; integer i;
    ввод (x);
    t := false;
    for i := 1 step 1 until n do
      if x [i] < 0 then
        begin t := true; go to L end;
    L: вывод (t)
  end
end
```

Машинная программа для этой же задачи будет соответствовать лишь операторной части внутреннего блока, исключая операторы ввода и вывода. Примем, что адрес переменной $x[i]$ равен $x + i$. Значение i будем хранить в индекс-регистре.

Рабочую часть цикла можно описать одной командой

$p \quad 4 \quad + \quad x \quad 0 \quad 0$ Если $x_i < 0$, то $\omega := 1$

Выполнение цикла нужно продолжить, если $x_i \geq 0$ (в этом случае $\omega = 0$) и $i < n$ (для текущего значения i), т. е. нужно прибегнуть к следующей команде окончания цикла:

$p+1 \quad 1 \quad 0 \text{ И } < A \quad + n \quad p \quad 1 \quad i := i + 1$; если $i \leq n \wedge \omega = 0$, то перейти на p

Для подготовки цикла к работе нужно заслать в индекс-регистр единицу.

Из цикла мы выйдем, если $\omega = 1$ (встретилось отрицательное число) или $i = n$ при $\omega = 0$ (все числа просмотрены и оказались неотрицательными). Команда окончания цикла не меняет значения признака ω , и после выхода из цикла можно разделить эти два случая по значению ω . Получим следующую программу:

Программа 4

	0	АИ	0	1	0	$i := 1$
p	4	+	x	0	0	Если $x_i \geq 0$, то $\omega := 0$
	1	$0 \text{ И } < A$	$+n$	p	1	$i := i + 1$; если $\omega = 0 \wedge i \leq n$, то на p
	0	ПО	0	p_1	t	$t := 0$ (при $\omega = 0$)
	0	ПЧ	$\langle 1 \rangle$	0	t	$t := 1$ (при $\omega = 1$)
p_1						выход

Здесь 1 представляет значение **true**, а 0 — значение **false** переменной t . Присваивать t какое-либо значение перед входом в цикл, как в АЛГОЛ-программе, здесь было бы неэкономно.

До сих пор в логических выражениях, которые мы рассматривали, участвовали только простые логические переменные. Обратимся теперь к выражениям, содержащим переменные с индексами. Пусть в задаче фигурируют три логических массива, которые могут быть описаны следующим образом:

Boolean array u, v, w [$1:n$]

где, для простоты, $n \leq 45$. Каждый из этих массивов может быть размещен в одной ячейке. Будем считать, что значение **true** переменной $u[i]$ представлено единицей в i -м справа разряде ячейки, адрес которой в программе заменим той же буквой u . Такое же допущение сделаем относительно массивов v и w .

Заметим прежде всего, что некоторые операции над такими логическими массивами, довольно сложно изображаемые средствами языка АЛГОЛ 60, очень просто выполняются машиной. Например, цикл

for $i := 1$ step 1 until n do
 $w[i] := u[i] \wedge v[i]$

реализуется одной командой

$0 \wedge u \ v \ w$

а цикл

for $i := n$ step -1 until 1 do
 $w[i] := w[i] \vee (u[i] \equiv v[i])$

— тремя командами

$$\left. \begin{array}{l} 0 \not\equiv u \ v \ r \quad r[i] := u[i] \not\equiv v[i] \\ 0 \not\equiv r \ t \ r \quad r[i] := \neg r[i] \\ 0 \vee w \ r \ w \quad w[i] := w[i] \vee r[i] \end{array} \right\} i = 1, \dots, n$$

где r — рабочая ячейка, а t — вспомогательная константа, содержащая 1 в разрядах с 1-го по n -й.

Пример 3. Имеется система из n некоторых устройств. Каждому устройству соответствует логическая переменная q_i , принимающая значение **true**, если i -е устройство неисправно. В некотором режиме работы этой системы требуется, чтобы определенные устройства, входящие в состав системы, были работоспособны. Эти устройства отмечены значениями **true** переменных p_i . Требуется определить, возможна ли работа системы в данном режиме. Если нет, то следует перейти на выполнение части программы, начинающейся с метки L .

Каждой переменной q_i и p_i сопоставим один разряд. Если $n \leq 45$, все переменные q_i разместим в одной ячейке (свободные разряды,

если они есть, заполним нулями). Адрес этой ячейки в программе заменим буквой q . Аналогично, значения переменных p_i будем хранить в разрядах ячейки p . Одному и тому же значению индекса i должны соответствовать разряды ячеек q и p с одинаковыми номерами (правило соответствия между значениями индекса и номерами разрядов несущественно).

Чтобы ответить на интересующий нас вопрос, достаточно вычислить значение величины

$$\alpha = \bigvee_{i=1}^n (q_i \wedge p_i)$$

или

$$\alpha = \exists i (1 \leq i \leq n \wedge q_i \wedge p_i)$$

(аналог скалярного произведения). Если все выделенные устройства исправны, то значение α равно **false**, если хотя бы одно неисправно, то $\alpha \equiv \text{true}$. Под переменную α отведем целую ячейку и примем, что нулевое слово в этой ячейке изображает значение **false**, а любое ненулевое — значение **true**.

В случае $n \leq 45$ для вычисления α достаточно одной команды

$k \quad 0 \quad \wedge \quad q \quad p \quad \alpha$ Если $\alpha \equiv \text{true}$, то $\omega := 0$

После этого операция перехода по $\omega = 0$ позволит перейти на соответствующий участок программы:

$k+1 \quad 0 \quad \text{ПО} \quad 0 \quad L \quad 0$ Если $\omega = 0$, то на L

Если $n > 45$, то каждый из массивов q и p займет m ячеек ($m \geq 2$), начиная с ячеек $q+1$ и $p+1$ соответственно. Программу вычисления α можно составить, например, так:

	0	ПЧ	0	0	α	$\alpha := \text{false}$
	0	АН	0	1	0	$H := 1$
k	6	$\wedge$	q	p	r	
	0	$\vee$	α	r	α	Если $\alpha \equiv \text{true}$, то $\omega := 0$
	1	$H < A$	$+m$	k	1	$H := H + 1$, если $H \leq m$, то на k
	0	ПО	0	L	0	Если $\omega = 0$, то на L

По-прежнему используется то обстоятельство, что операция окончания цикла сохраняет значение признака ω .

Полученную программу можно сократить. Если выяснилось, что хотя бы одно из интересующих нас устройств неисправно, то дальше проверку можно не проводить. Кроме того, при $n \leq 45$ в ячейке α сохранялась осмысленная информация — какие именно из выделенных устройств неисправны. Сейчас содержимое ячейки α такой дополнительной информации не несет и команда $k+1$ (если изменить следующую за ней команду) — лишняя. Более экономная

программа такова:

	0	АИ	0	1	0	$H := 1$
k	6	$\wedge$	q	p	α	Если $\alpha \equiv \text{true}$, то $\omega := 0$
	1	$1H < A$	~~$+m$~~	k	1	$H := H + 1$; если $\omega = 1 \wedge H \leq m$, то на k
	0	ПО	0	L	0	Если $\omega = 0$, то на L

Столь просто программируются лишь циклы, в рабочей части которых встречаются только переменные с одинаковыми индексами, относящиеся к массивам одного и того же строения, одинаково размещенным в памяти. Достаточно среди этих переменных появиться одной простой переменной, как программирование сразу осложнится. В общем случае значения переменных, входящих в логическое выражение, представлены разрядами, занимающими различное положение в ячейках. При программировании таких выражений приходится исследовать и изменять содержимое отдельных разрядов ячейки и прибегать к операциям сдвига.

В качестве типичного примера рассмотрим, как программируются операторы

if $u[i]$ then go to L
 $u[i] := z$
 $z := u[i]$

при условии, что простая логическая переменная z представлена, например, 1-м разрядом ячейки z (остальные разряды заполнены нулями), переменная $u[i]$ представлена i -м разрядом ячейки u (остальные разряды содержат значения других элементов массива u), а текущее значение целочисленной переменной i ($1 \leq i \leq 45$) находится на индекс-регистре. Нам понадобится одна вспомогательная константа t , содержащая лишь одну единицу в 1-м разряде:

t 000 0000 0000 0001

Оператор if $u[i]$ then go to L реализуется командами

p	4	$\leftarrow CA$	0077	t	r	$r_i := 1, r_j := 0$ для $j \neq i$ ($H = i$)
	0	$\wedge$	u	r	0	Если $u_i = 1$, то ($\omega := 0$)
	0	ПО	0	L	0	перейти на L

В команде p первый исполнительный адрес равен $77 + H = 100_8 + (i - 1)$. Таким образом, содержимое ячейки t сдвигается на $i - 1$ разряд влево и единственная единица попадает в i -й разряд ячейки r . С ее помощью исследуется значение i -го разряда ячейки u . Ячейка типа ячейки r , содержащая 1 в интересующем нас разряде, называется *щупом*.

Основная трудность при программировании оператора $u[i] := z$ состоит в том, чтобы сохранить содержимое всех разрядов ячейки u , кроме i -го. Для этого сначала следует погасить содержимое i -го разряда (выполнить присваивание $u[i] = \text{false}$), а затем, сдвинув зна-

чение z в i -й разряд, выполнить поразрядную операцию логического сложения. Программа оказывается довольно громоздкой:

$$\left. \begin{array}{llllll} 4 & \leftarrow CA & 0077 & t & r & r_i := 1, r_j := 0 \\ 0 & \neq & u & r & r & r_i := 1 - u_i, r_j := u_j \\ 0 & \wedge & u & r & u & u_i := 0, u_j := u_j \\ 4 & \leftarrow CA & 0077 & z & r & r_i := z, r_j := 0 \\ 0 & \vee & u & r & u & u_i := z, u_j := u_j \end{array} \right\} \begin{array}{l} 1 \leq j \leq 45, \\ j \neq i \end{array}$$

Оператор $z := u[i]$ следует предварительно преобразовать к виду:

if $u[i]$ then $z := \text{true}$ else $z := \text{false}$

после чего его программирование не составляет труда:

```

4 ← CA 0077    t    r  r_i:=1, r_j:=0 для j ≠ i
0    ∧        u    r    0  Если u_i=1, то ω:=0
p    0  П0      t    p+2  z  z:=true; если ω=0, то на p+2
0    ПЧ        0    0    z  z:=false.
```

Здесь снова использован шуп (ячейка r).

Описанные выше приемы представления логических величин и программирования логических выражений применяются в программах в разнообразных сочетаниях. Чтобы найти наиболее целесообразное (простое, но эффективное) сочетание этих приемов для решения сложной задачи, часто приходится рассмотреть довольно много вариантов программы.

Пример 4. Задан массив чисел $a[1:n]$ и логический массив $p[1:n]$. Значения $p_i = \text{true}$ выделяют некоторые из элементов a_i массива a . Требуется найти сумму u выделенных элементов и сумму v всех остальных, т. е. составить машинную программу, отвечающую следующему алгоритму на АЛГОЛе:

```

begin integer n, i; ввод (n);
  begin array a[1:n]; Boolean array p[1:n]; real u, v;
    ввод (a, p); u:=v:=0;
    for i:=1 step 1 until n do
      if p[i] then u:=u+a[i] else v:=v+a[i];
    вывод (u, v)
  end
end
```

Будем считать, что элемент a_i размещен в ячейке с адресом $a+i$. Для хранения значения одной логической переменной p_i выделим один разряд. Пусть, для разнообразия, значению $i=1$ соответствует 45-й разряд ячейки, значению $i=2$ — 44-й разряд и т. д.

Рассмотрим отдельно два случая:

а) $n \leq 45$ (весь логический массив размещен в одной ячейке p);

б) $n > 45$ (логический массив занимает $m \geq 2$ последовательных ячеек, начиная с ячейки $p + 1$, так что, например, значению $i = 46$ отвечает 45-й разряд ячейки $p + 2$).

С л у ч а й а. Составление программы начнем, как обычно, с рабочей части цикла. Для исследования значения переменной p_i воспользуемся щупом, содержащим 1 в разряде с номером 46 — i и нули во всех остальных разрядах. Команды рабочей части цикла можно написать следующим образом (считаем, что в индекс-регистре находится текущее значение i):

k_1	0	$\wedge$	p	r	0	Если $p_i \equiv \text{false}$, то ($\omega := 1$)
	0	П1	0	k	0	перейти на k
	2	+	u	a	u	$u := u + a_i$ (при $\omega = 0$)
	0	ПБ	0	$k + 1$	0	Обход команды k
k	2	+	v	a	v	$v := v + a_i$ (при $\omega = 1$)

Для завершения цикла нужно подготовить щуп r к проверке следующего значения p_i (сдвинуть его на один разряд вправо), увеличить значение i на 1 и проверить, не закончен ли цикл. Это можно сделать командами:

$k + 1$	0	$\leftarrow$ CA	0077	r	r	$i := i + 1$
	1	$H < A$	$+n$	k_1	1	$i := i + 1$; если $i \leq n$, то на k_1

Исследование значений p_i начинается с 45-го разряда ячейки p , соответствующего $i = 1$. Для этого в подготовительной части цикла в ячейку r надо заслать слово z_1 400 0000 0000 0000. Кроме того, нужно послать значение $i = 1$ в индекс-регистр и подготовить к суммированию ячейки u и v . Все это выполняют команды:

	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	0	0	v	$v := 0$
	0	ПЧ	z_1	0	r	} $i := 1$
$k_1 - 1$	0	АН	0	1	0	

Итак, мы получили следующую программу:

Программа 5

	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	0	0	v	$v := 0$
	0	ПЧ	z_1	0	r	} $i := 1$
	0	АН	0	1	0	
k_1	0	$\wedge$	p	r	0	Если $p_i \equiv \text{false}$, то
	0	П1	0	k	0	перейти на k
	2	+	u	a	u	} при $\omega = 0$
	0	ПБ	0	$k + 1$	0	
k	2	+	v	a	v	$v := v + a_i$ (при $\omega = 1$)
	0	$\leftarrow$ CA	77	r	r	} $i := i + 1$, если $i \neq n$, то перейти на k_1
	1	$H < A$	$+n$	k_1	1	

Константа (щуп для $i = 1$):

z_1 4 00 0000 0000 0000

С л у ч а й б. Пусть теперь $n > 45$ и массив p размещается в ячейках $p + 1 \div p + m$. Таким же образом, как в предыдущей программе, будем поочередно исследовать значения переменных p_i . Рабочая часть программы будет такой же, только содержимое очередной ячейки $p + j$ нужно предварительно передать в фиксированную ячейку p_1 (добавится одна подготовительная команда). В остальном команды $k_1 \div k + 1$ не изменятся. Но в отличие от предыдущей программы, в заключительной части цикла нужно проверить, не исчерпаны ли все значения p_i , хранящиеся в данной ячейке. Вспомним, что команда $\leftarrow SA$ вырабатывает признак $\omega = 1$, если во всех разрядах результата находятся нули. Мы получим $r = 0$ ($\omega = 1$), если единичка щупа выйдет за разрядную сетку, т. е. если исследованы все значения переменных очередной ячейки $p + j$ и нужно в p_1 переслать очередную часть массива p . Пусть в ячейке j_1 хранится код:

j_1 0 АИ 0 $+j$ 0

где j — номер ячейки, в которой находится очередная часть массива p . Программу можно завершить командами:

$k+1$	0	$\leftarrow SA$	0077	r	r	$i := i + 1$
	0	ПЮ	0	k_2	0	Если $\omega = 0$, то на k_2
	4	КИ	0	j_1	i_1	$i := H$; $H := j$
	4	ПЧ	p	0	p_1	Очередная часть массива p
	0	ПЧ	z_1	0	r	$r := \langle 400\ 0000\ 0000\ 0000 \rangle$
	4	КИ	1	i_1	j_1	$j := H + 1$; $H := i$
k_2	1	$H < A$	$+n$	k_1	1	

Осталось дополнить подготовительную часть цикла командой засылки нужного кода в ячейку j_1 . Так как содержимое ячейки $p + 1$ пересылается в p_1 подготовительными командами, в ячейку j_1 нужно послать значение $j = 2$. Эту засылку можно совместить с присваиванием H значения 1.

Запишем полученную программу:

Программа 6

	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	0	0	v	$v := 0$
	0	ПЧ	$p+1$	0	p_1	} $i := 1$
	0	ПЧ	z_1	0	r	
	0	АИ	2	1	j_1	$j := 2$, $H := 1$
k_1	0	$\wedge$	p_1	r	0	Если $p_i = \text{false}$, то ($\omega := 1$) перейти на k
	0	П1	0	k	0	
	2	$+$	u	a	u	$u := u + a_i$ ($H = i$)
	0	ПБ	0	$k+1$	0	

k	2	+	v	a	v	$v := v + a_i$
$k + 1$	0	$\leftarrow CA$	0077	r	r	$i := i + 1$
	0	ПО	0	k_2	0	Если $\omega = 0$, то на k_2
	4	КИ	0	j_1	i_1	$i := I$; $I := j$
	4	ПЧ	p	0	p_1	
	0	ПЧ	z_1	0	r	
	4	КИ	1	i_1	j_1	$j := I + 1$; $I := i$
k_2	1	$I < A$	$+ n$	k_1	1	$I := I + 1$; если $I \leq n$, то на k_1

где

z_1 400 0000 0000 0000

Посмотрим, можно ли сократить полученную программу. В программе два раза встречается команда пересылки z_1 в r , две команды пересылки $p + j$ в p_1 (для $j = 1$ и для $j > 1$). Попробуем их совместить. Команды подготовки цикла исключить или переместить нельзя. Но можно записать их в таком виде, чтобы их можно было использовать для пересылки $p + j$ при любом j . Для этого программу подготовительной части цикла составим так:

	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	0	0	v	$v := 0$
	0	АИ	1	1	j_1	$j := 1$; $I := 1$
k_2	4	КИ	0	j_1	i_1	$i := I$; $I := j$
	4	ПЧ	p	0	p_1	
	0	ПЧ	z_1	0	r	
	4	КИ	1	i_1	j_1	$j := I + 1$, $I := i$

Программа рабочей части (команды $k_1 \div k + 1$), очевидно, не изменится. Как записать команды окончания цикла? Если после сдвига $\omega = 0$, то это значит, что еще не все 45 разрядов очередной ячейки $p + j$ проверены, и если $i < n$, нужно исследовать очередной разряд (передать управление на команду k_1). Это осуществляется командой

$k + 2$ 1 0 $I < A$ $+ n$ k_1 1

Если же управление передано команде $k + 3$, то это значит, что 1) либо $i = n$ — исчерпаны все значения параметра i , значение ω нас уже не интересует;

2) либо $i < n$, но $\omega = 1$ — исследованы все разряды ячейки $p + j$.

Во втором случае выполнение программы нужно продолжить, передав управление на команду k_2 . Проверку и передачу управления произведет команда

$k + 3$ 1 $I < A$ $n + 1$ k_2 0

Напомним, что операции окончания цикла посылают в индекс-регистр значение третьего исполнительного адреса независимо от того, выполняются ли

условия передачи управления по второму адресу или нет. Это значит, что команда $k + 3$ начинает выполняться с I , равным третьему исполнительному адресу команды $k + 2$, т. е. $I = i + 1$. Если команда $k + 2$ начала выполняться с $I = n$, то после ее выполнения $I = n + 1$. Поэтому мы и сравниваем в команде $k + 3$ значение I с $n + 1$. Если $i < n + 1$, нам не нужно менять значение I . Поэтому в третий адрес команды $k + 3$ мы записываем 0, в признак модификации третьего адреса 1.

Мы получили новый вариант программы, короче на две команды:

Программа 7

	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	0	0	v	$v := 0$
	0	АИ	1	1	j_1	$j := 1; I := 1$
k_2	4	КИ	0	j_1	i_1	$i := I; I := j$
	4	ПЧ	p	0	p_1	$(I = j)$
	0	ПЧ	z_1	0	r	
	4	КИ	1	i_1	j_1	$j := I + 1; I := i$
k_1	0	$\wedge$	p_1	r	0	Если $\omega = 1$, то
	0	П1	0	k	0	перейти на k
	2	$+$	u	a	u	$u := u + a_i \ (I = i)$
	0	ПБ	0	$k + 1$	0	
k	2	$+$	v	a	v	$v := v + a_i$
$k + 1$	0	$\leftarrow$ СА	0077	r	r	$i := i + 1$
	1	$0I < A$	$+n$	k_1	1	$I := I + 1$; если $I \leq n \wedge \omega = 0$, то на k_1
	1	$I < A$	$n+1$	k_2	0	Если $I \leq n \ (\wedge \omega = 1)$, то на k_2

Константа:

z_1	4	0	0	0	0

§ 4. Применение логических операций и сдвигов для преобразования произвольных слов

Логические операции и сдвиги удобны и эффективны для исследования и изменения не только отдельных двоичных разрядов, но и целых групп разрядов.

Так, например, в коде любой команды содержится 7 групп разрядов, играющих более или менее самостоятельную роль: код операции, три адреса и три признака модификации адресов. Нередко возникает потребность извлечь эти группы из одних кодов команд или заготовок и вставить в другие коды. Рассмотрим пример (довольно типичный), когда команда целиком формируется из частей, выделяемых из других кодов команд.

Пример 1. Пусть в ячейках s_1, s_2, s_3, s_4 находятся следующие коды команд:

s_1	$\pi_1' \pi_2' \pi_3'$	K_1	A_1'	A_2'	A_3'
s_2	$\pi_1'' \pi_2'' \pi_3''$	K_2	A_1''	A_2''	A_3''
s_3	$\pi_1''' \pi_2''' \pi_3'''$	K_3	A_1'''	A_2'''	A_3'''
s_4	$\pi_1^{IV} \pi_2^{IV} \pi_3^{IV}$	K_4	A_1^{IV}	A_2^{IV}	A_3^{IV}

В ячейке s нужно получить код

$$\pi_1 \pi_2 \pi_3 \quad K \quad A \quad B \quad C$$

где $A = A'_2$ и $\pi_1 = \pi'_2$, $B = A'_1$ и $\pi_2 = \pi'_1$, $C = A'_1''$ и $\pi_3 = \pi'_1''$ (иначе говоря, первый, второй и третий адреса команды s и их признаки модификации совпадают соответственно со вторым, первым и также первым адресами команд s_1 , s_2 , s_3 с их признаками модификации), а код операции $K = K_4$.

Для того чтобы выделить часть слова, применим операцию логического умножения. Заранее приготовим константы для выделения кода операции, первого, второго и третьего адресов с их признаками модификации, а именно

z_0	0 77	0000	0000	0000
z_1	4 00	7777	0000	0000
z_2	2 00	0000	7777	0000
z_3	1 00	0000	0000	7777

После выполнения команды

$$0 \wedge s_3 \quad z_0 \quad s$$

в ячейке s получим код:

$$0 \quad K_4 \quad 0 \quad 0 \quad 0$$

Таким же образом из кода команды, заданного в ячейке s_1 , выделим второй адрес с его признаком модификации:

$$0 \wedge s_1 \quad z_2 \quad r \quad r := \langle 0\pi'_2 0 0 0 A'_2 0 \rangle$$

Нам нужно получить значение A'_2 в первом адресе и соответствующее значение признака модификации первого адреса, т. е. признак модификации кода r нужно сдвинуть на один разряд влево, содержащее его второго адреса — на 12 разрядов влево. Это можно осуществить за две операции:

$$\left. \begin{array}{l} 0 \leftarrow CA \quad 0101 \quad r \quad r \\ 0 \leftarrow MA \quad 0113 \quad r \quad r \end{array} \right\} \langle \pi'_2 00 0 A'_2 0 0 \rangle$$

(В результате выполнения первой из этих команд все разряды слова r сдвинутся на один разряд влево, вторая сдвинет влево на $13_8 = 11_{10}$ разрядов только содержимое $1 \div 36$ разрядов, т. е. адресную часть кода команды.)

Полученный код объединим с ранее выделенным:

$$0 \vee s \quad r \quad s \quad \langle \pi'_2 00 K_4 A'_2 0 0 \rangle$$

Аналогичным образом формируем второй и третий адреса (сначала выпишем уже полученные команды):

Программа 1

0	$\wedge$	s_4	z_0	s	«000 K 0 0 0»
0	$\wedge$	s_1	z_2	r	«0 π_1 0 0 0 A 0»
0	$\leftarrow CA$	0101	r	r	} « π_1 00 0 A 0 0»
0	$\leftarrow MA$	0113	r	r	
0	$\vee$	s	r	s	« π_1 00 K A 0 0»
0	$\wedge$	s_2	z_1	r	« π_2 00 0 B 0 0»
0	$\leftarrow CA$	0077	r	r	} «0 π_2 0 0 0 B 0»
0	$\leftarrow MA$	0065	r	r	
0	$\vee$	s	r	s	} ($65_8 = 100_8 - 13_8$)
0	$\wedge$	s_3	z_1	r	
0	$\leftarrow CA$	0076	r	r	} «00 π_3 0 0 0 C»
0	$\leftarrow MA$	0052	r	r	
0	$\vee$	s	r	s	} ($52_8 = 100_8 - 26_8$; $26_8 = 22_{10}$)
					« $\pi_1\pi_2\pi_3$ K A B C»

Изложенные приемы типичны для формирования команд.

Те же приемы могут применяться для анализа и формирования произвольных кодов. На многих машинах типа М-20 существует возможность вводить и обрабатывать тексты. *Текст* представляет собой последовательность *литер* — букв, цифр, знаков операций, скобок и других печатных знаков. Каждая литера кодируется 7-разрядным двоичным числом. Код буквы А, например, имеет вид $0100000_2 = 040_8$. В одной ячейке умещается 6 кодов литер, занимающих разряды с 1-го по 42-й. Подробнее об этом будет сказано в гл. 14.

Пример 2. Пусть в ячейках $a + 1, a + 2, \dots$ располагается некоторый текст. Признаком конца текста является 7-разрядный код $1011011_2 = 133_8$, число ячеек, занятых текстом, заранее не известно. Требуется определить, сколько раз в этом тексте встречается литера А. Текст должен быть сохранен.

Программа для этого примера представляет собой двойной цикл. Во внутреннем цикле просматриваются литеры одной ячейки слева направо. Если встретится код 133, работа программы заканчивается, если встретится код 040, к счетчику числа вхождений литеры А в данный текст добавляется 1. Во внешнем цикле осуществляется последовательный перебор ячеек, занятых текстом. Содержимое этих ячеек поочередно передается в рабочую ячейку r . После исследования очередной литеры (во внутреннем цикле) содержимое ячейки r сдвигается на 7 разрядов влево и выделяются его разряды с 1-го по 42-й. Если все они заполнены нулями, то в ячейке не осталось литер, подлежащих исследованию, и выполнение внутреннего цикла заканчивается. Это простейший способ организовать выход из внутреннего цикла.

Программа 2

	0	АИ	0	1	0	$H := 1$
	0	ПБ	0	p_4	s	$s := 0$; перейти на p_4
p_1	2	АИ	0	1	0	$H := H + 1$
p_2	0	$\wedge$	r	z_1	r_1	Выделение очередной литеры
	0	$\neq$	r_1	z_2	0	Если код литеры = 133,
	0	П1	0	$p_4 + 1$	0	то выход
	0	$\neq$	r_1	z_3	0	Если код литеры $\neq 040$,
	0	П0	0	p_3	0	то перейти на p_3 ,
	0	+	s	$\langle 1 \rangle$	s	иначе $s := s + 1$
p_3	0	$\leftarrow CA$	107	r	r	Если не все литеры исследованы, то ($\omega := 0$) перейти на p_2
	0	$\wedge$	r	z_4	0	
	0	П0	0	p_2	0	
p_4	4	ПБ	a	p_1	r	Выборка очередной ячейки, на p_1

Программа использует константы:

z_1	0	77	4000	0000	0000
z_2	0	55	4000	0000	0000
z_3	0	20	0000	0000	0000
z_4	0	77	7777	7777	7777

У п р а ж н е н и я. 1. Проверить, совпадают ли коды операций команд, хранящихся в ячейках p и q .

2. В ячейках $a + 1, a + 2, \dots, a + n$ располагается текст, в котором могут встречаться скобки: «(» (код 022₈) и «)» (код 023₈). Проверить, находятся ли эти скобки в обычном соответствии.

Глава 13. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ МАШИНЫ М-20 (ПРОДОЛЖЕНИЕ)

§ 1. Выполнение арифметических операций в машине

Продолжим начатый в § 4 гл. 2 и § 2 гл. 5 разбор методов выполнения арифметических операций в машине. Далеко не каждое число может быть представлено в машине. Например, такое «простое» число, как $1/3$, может быть представлено в машине М-20 лишь приближенно, целое число n может быть представлено точно, если $|n| < 2^{36}$, приближенно, если $2^{36} \leq |n| < 2^{63}$, и не может быть представлено с удовлетворительным приближением, если $|n| \geq 2^{63}$. Обычное (нормализованное) представление числа x в машине — это представление его в виде

$$x' = \pm 2^p \cdot m, \quad (1)$$

где p — целое, m — правильная 36-разрядная двоичная дробь, $1/2 \leq m < 1$. Числа вида (1), а также машинный нуль будем называть *машинными*. *Соседние* машинные числа — это либо

$$\pm 2^p \cdot m \text{ и } \pm 2^p (m + 2^{-36})$$

(при $m < 1 - 2^{-36}$), либо

$$\pm 2^p (1 - 2^{-36}) \text{ и } \pm 2^{p+1} \cdot 1/2.$$

В обоих случаях разность между машинным числом x' вида (1) и соседним машинным числом не превосходит по абсолютной величине $2^p \cdot 2^{-36}$. Это дает основание считать машинное число x' вида (1) *хорошим* приближением для произвольного вещественного числа x , если

$$|x - x'| \leq 2^p \cdot 2^{-37}, \quad (2)$$

и *приемлемым* приближением, если

$$|x - x'| \leq 2^p \cdot 2^{-36}. \quad (3)$$

Хорошие приближения существуют, таким образом, лишь для чисел x , удовлетворяющих неравенству

$$2^{-64} (1/2 - 2^{-37}) \leq |x| \leq 2^{63} (1 - 2^{-37}).$$

Кроме того, для чисел, удовлетворяющих неравенству

$$|x| \leq 2^{-64} \cdot 2^{-37} = 2^{-101},$$

хорошим приближением можно считать машинный нуль, для которого $p = -64$ и $m = 0$. Не имеют хороших приближений не только большие числа ($|x| > 2^{63} - 2^{26}$), но и малые числа из диапазона $2^{-101} < |x| < 2^{-65} - 2^{-101}$.

Результат арифметической операции (в обычном смысле) над двумя машинными числами очень часто не является машинным числом. Более того, этот результат может даже не иметь хорошего машинного приближения. Поэтому при выполнении машинной арифметической операции над машинными числами надо установить, существует ли хорошее (или хотя бы приемлемое) машинное приближение для точного результата соответствующего арифметического действия в обычном понимании, и если существует, образовать это приближение. Сложность этой задачи состоит в том, что на выполнение операции надо потратить как можно меньше времени (для получения высокого быстродействия) и обойтись при этом по возможности малым количеством физических элементов, из которых состоит арифметическое устройство машины (для удешевления машины и повышения ее надежности).

Проектируя арифметическое устройство машины, конструкторы обычно добиваются, чтобы хорошее приближение к точному результату вырабатывалось (если это возможно) лишь для операций над нормализованными операндами. Если же операнды не нормализованы, то машинная операция дает обычно осмысленный результат, но в чем-то отступающий от наилучшего возможного.

Разберем на примере операций сложения и вычитания, какими средствами это достигается. При этом нам все-таки придется опустить ряд деталей технической реализации.

Выполнение операции начинается с выборки операндов из ячеек оперативной памяти с заданными в команде исполнительными адресами на регистры арифметического устройства. Арифметическое устройство имеет три регистра, предназначенные (грубо говоря) для выборки двух операндов и для формирования результата. Последний регистр называется *сумматором*. В любом из регистров каждому из 45 разрядов ячейки соответствует один двоичный разряд. Эти разряды называются *основными*. Они нумеруются, как и разряды ячеек, справа налево числами от 1 до 45. Разряд 1 называется также *младшим основным разрядом* мантиссы, разряд 36 — *старшим основным разрядом* мантиссы, разряды 37—43 — *разрядами порядка*, разряды 44 и 45 — соответственно *разрядом знака* и *признака* числа. Кроме основных разрядов, регистры АУ имеют дополнительные разряды, назначение которых будет объяснено ниже.

Перед выборкой операндов во все разряды всех регистров записываются нули (это называется *очисткой регистров*). Операнды

выбираются в основные разряды соответствующих им регистров точно в том виде, в каком они хранились в памяти. Нормализация операндов не производится, хотя она и могла бы содействовать повышению точности результата. Но, как уже было сказано, меры к сохранению точности принимаются лишь для нормализованных операндов. Отказ от нормализации операндов приводит к экономии времени, а также расширяет, как мы увидим на примерах, область полезного применения некоторых арифметических операций.

После выборки операндов определяется, какую операцию надо фактически выполнять над мантиссами, и фиксируется ожидаемый знак результата. Если выполняется операция сложения и знаки операндов совпадают, или выполняется операция вычитания, а знаки различны, то мантиссы надо складывать. Если же складываются числа разных знаков или вычитаются числа с одинаковыми знаками, то в дальнейшем из мантиссы первого операнта надо будет вычесть мантиссу второго операнта. Во всех случаях *ожидаемый знак* результата — это знак первого операнта. Впоследствии он может измениться.

Далее определяется разность порядков операндов. Если эта разность равна нулю, то одноименные разряды мантисс (т. е. разряды, соответствующие одной и той же степени числа 2 в двоичной записи числа) располагаются в разрядах с одним и тем же номером (т. е. «друг под другом»). Если же разность порядков отлична от нуля, то, чтобы добиться такого расположения, надо выполнить так называемое *выравнивание порядков*. *Ожидаемый порядок* результата равен большему из порядков операндов.

При выравнивании порядков мантисса числа с меньшим порядком (какого именно, определяется по знаку разности порядков) сдвигается в своем регистре вправо на число разрядов, равное абсолютной величине разности порядков. При этом младшие разряды мантиссы (коротко — ее *хвост*) не могут сохраниться в основных разрядах регистра. Хвост сдвинутой мантиссы изображает число, не превосходящее единицы младшего основного разряда после выравнивания мантисс (т. е. $2^p \cdot 2^{-36}$, где p — порядок операнта, мантисса которого не сдвигалась). Но это число может быть больше половины единицы этого разряда (больше $2^p \cdot 2^{-37}$), и если мы хотим получить хорошее приближение к точному результату, то надо произвести такую проверку. Благодаря тому, что числа изображаются в двоичной системе счисления, для этой проверки достаточно исследовать лишь старшую двоичную цифру хвоста. Чтобы сохранить эту цифру, к каждому из регистров арифметического устройства добавлен *младший дополнительный разряд*, куда и попадает эта цифра при сдвиге. Младший дополнительный разряд регистра, мантисса в котором не сдвигалась, содержит цифру 0.

После выравнивания порядков можно приступить к сложению или вычитанию мантисс. Рассмотрим сначала случай, когда мантиссы

складываются. Если выравнивание порядков не производилось, то все цифры мантисс сохранились в основных разрядах регистров и не надо принимать никаких дополнительных мер для получения хорошего приближения к результату, — он и так будет точным. Если же одна из мантисс сдвигалась, то надо произвести *округление*. Если старшая цифра хвоста сдвинутой мантиссы равна нулю, то хвостом можно просто пренебречь, если же эта цифра равна 1, то для хорошего округления надо добавить 1 к младшему основному разряду суммы мантисс. Этого результата можно достичь, если поместить цифру 1 в младший дополнительный разряд того регистра, мантисса в котором не сдвигалась при выравнивании, и складывать не 36-разрядные, а 37-разрядные мантиссы, включая их младшие дополнительные разряды. У результата надо сохранить 36 цифр в его основных разрядах.

Однако надо считать за тем, что сумма мантисс может оказаться больше 1. Если это так, то при сложении мантисс возникает единица переноса из старшего основного разряда. Чтобы ее сохранить, к сумматору добавляется еще один — *старший дополнительный* разряд, куда и попадает эта единица. В действительности этих старших дополнительных разрядов два, и не только у сумматора, но и у обоих разрядов операндов, но их назначение мы разбирать не будем.

Пусть теперь мантиссы требуется вычитать. И в этом случае не надо принимать никаких дополнительных мер по сохранению точности, если выравнивание порядков не производилось. Если же оно производилось и обе исходные мантиссы были нормализованы, то сдвинутая мантисса всегда оказывается меньше мантиссы, не подвергавшейся сдвигу. Для получения абсолютной величины разности мантисс надо сдвинутую мантиссу вычесть из той, которая не сдвигалась. Если старшая цифра хвоста сдвинутой мантиссы равна 0, то разность (в основных разрядах) не требует округления, если же эта цифра равна 1, то для хорошего округления из разности мантисс надо вычесть 1 младшего основного разряда. Это произойдет автоматически, если вычитать 37-разрядные мантиссы, сохранив цифру 0 в младшем дополнительном разряде мантиссы, не подвергавшейся сдвигу (при наличии 1 в младшем дополнительном разряде вычитаемого произойдет заем 1 из младшего основного разряда).

Итак, можно сформулировать следующий принцип округления результата сложения или вычитания мантисс. Если выполнены следующие условия:

- а) производилось выравнивание порядков,
- б) мантиссы фактически складываются,
- в) обе мантиссы были отличны от нуля,

то в младший дополнительный разряд регистра, мантисса в котором не сдвигалась, следует поместить 1. Если хоть одно из этих усло-

вий не выполнено, то цифра младшего дополнительного разряда в обоих регистрах операндов сохраняется неизменной. После сложения или вычитания мантисс код в основных разрядах сумматора будет хорошим приближением к точному результату.

В действительности при вычитании мантисс вычитается не меньшая мантисса из большей (чтобы выяснить, которая из мантисс больше, надо затратить незначительно меньше усилий, чем на само вычитание), а мантисса второго операнда из мантиссы первого. Если результат отрицателен (это устанавливается по цифре старшего дополнительного разряда), он получается в дополнительном коде. В этом случае ожидаемый знак результата изменяется на противоположный и получившийся на сумматоре код мантиссы заменяется его дополнением до 1 (с учетом старшего дополнительного разряда — до 2).

Итак, определены знак, порядок и мантисса результата. Признак результата полагается равным 1, если хотя бы один из операндов имел признак 1. Но мантисса может оказаться ненормализованной и даже равной нулю. В последнем случае результат полагается равным машинному нулю, т. е. во все разряды результата с 1-го по 44-й засылается цифра 0. Ненулевым может остаться лишь 45-й разряд, представляющий признак результата. Если же мантисса не равна нулю, то результат надо нормализовать.

После сложения мантисс их сумма может оказаться больше 1. В этом случае содержимое всех разрядов мантиссы сдвигается на один разряд вправо. В старший основной разряд мантиссы попадает цифра старшего дополнительного разряда, равная единице. Сдвиг мантиссы вправо означает ее уменьшение вдвое. Чтобы оставить величину числа неизменной, к порядку результата добавляется 1. Все это называется *нормализацией вправо*.

После нормализации вправо порядок результата может оказаться равным 64_{10} (чтобы можно было это обнаружить, к 7-разрядному сумматору порядков присоединяется *дополнительный старший разряд* порядка). Для числа с таким порядком хорошего приближения не существует. Поэтому считается, что завершить выполнение операции невозможно. Происходит *автоматический останов* машины («авост») из-за *переполнения* сумматора порядков.

При нормализации вправо цифра младшего основного разряда мантиссы сдвигается в младший дополнительный разряд. Новое содержимое основных разрядов мантиссы может уже не быть хорошим приближением к точному результату. Чтобы получить хорошее приближение, необходимо дополнительное преобразование результата, называемое *доокруглением*.

У п р а ж н е н и е 1. Что нужно сделать с мантиссой, чтобы осуществить доокругление? Может ли после доокругления вновь потребоваться нормализация?

Если мантиссы операндов вычитались, а также если они складывались, но операнды не были нормализованными, мантисса на сумматоре может оказаться меньше $1/2$, т. е. цифра старшего основного разряда может быть не равна 1. В этом случае производится так называемая *нормализация влево*. Она заключается в том, что мантисса сдвигается до тех пор, пока в старший основной разряд не попадет единица (случай, когда все цифры мантиссы равны нулю, уже был рассмотрен). Порядок результата уменьшается на единицу после каждого сдвига мантиссы на один разряд (иначе говоря, из порядка вычитается число разрядов, на которое была сдвинута мантисса).

У п р а ж н е н и е 2. Показать, что если при сложении или вычитании нормализованных чисел мантиссы фактически вычитались, то машинный результат, если он отличен от нуля, является по крайней мере приемлемым приближением к точному результату.

После нормализации влево порядок результата может оказаться меньше —64. В этом случае, как и при получении нулевой разности мантисс, результат заменяется машинным нулем с сохранением признака в 45-м разряде. Ранее было отмечено, что существует группа отличных от нуля малых чисел, которые не имеют хороших машинных приближений. Поэтому на некоторых машинах (но не на машинах типа М-20) при получении результата с порядком меньшим, чем наименьший представимый в машине, происходит автоматический останов по *исчезновению* порядка (в противоположность его переполнению для больших чисел).

После нормализации результат (его основные разряды) записывается в ячейку оперативного запоминающего устройства с номером, равным третьему исполнительному адресу выполняемой команды.

Рассмотрим примеры выполнения в машине арифметических операций. Для того чтобы не выписывать в примерах все 45 двоичных разрядов, условно будем рассматривать числа с 6 разрядами мантиссы и 4 разрядами порядка (5 разрядов кода порядка).

Сначала рассмотрим сложение двух положительных чисел с одинаковыми порядками:

Знак	Порядок	Мантисса
+	0	10000
	0	10001
	0	10000
	0	011000 (число ненормализованное)
	0	10000
		1 001001
		Старш. дополн. разряд

Мантиссы складываются поразрядно, порядок суммы совпадает с порядками слагаемых. При сложении мантисса суммы оказалась больше единицы. Происходит нормализация вправо. В результате

нормализации получится следующее число:

$$\begin{array}{c} \text{результат} \\ \hline 0\ 10001\ 100100\ 1 \end{array}$$

единица младшего разряда сдвигается за разрядную сетку и теряется. Округление не производилось, так как не было выравнивания порядков.

Теперь рассмотрим пример операции, в которой участвуют числа с разными порядками. Пусть вычитаются следующие числа:

$$\begin{array}{ll} 1\ 10000 & 110001 \text{ (число с нулевым порядком)} \\ 0\ 01111 & 110001 \text{ (порядок числа равен } -1) \end{array}$$

Операнды имеют разные знаки, поэтому вычитание заменяем сложением. Ожидаемый знак результата (—) определяется знаком первого слагаемого. Сначала происходит выравнивание порядков — мантисса второго слагаемого сдвигается вправо на один разряд. В младший дополнительный разряд первого слагаемого для округления результата посылается 1. В нашем примере будут складываться следующие числа:

$$\begin{array}{r} \begin{array}{cccc} 1 & 10000 & 110001 & 1 \text{— младш. дсполн. разряд} \\ + & (1) & 10000 & 011000 & 1 \\ \hline 1 & 10000 & 1 & 001010 & 0 \end{array} \end{array}$$

После нормализации вправо получим следующий результат сложения с округлением:

$$1\ 10001\ 100101$$

(доокругление не потребовалось, так как цифра младшего дополнительного разряда перед нормализацией вправо была равна нулю).

Коротко рассмотрим остальные арифметические операции. При умножении двух чисел с 36-разрядными мантиссами получается точное двоичное число с мантиссой, имеющей 72 разряда. В арифметическом устройстве машин типа М-20 именно такая мантисса результата и вырабатывается в качестве промежуточного результата операции умножения. Порядок результата равен сумме порядков сомножителей. Так как мантисса каждого из сомножителей меньше 1, то и мантисса произведения меньше 1, но она может быть и меньше $1/2$, т. е. результат может быть ненормализованным. В этом случае все 72 разряда мантиссы результата сдвигаются на один разряд влево и из порядка результата вычитается 1. Если оба сомножителя были нормализованными числами, в результате умножения тоже получится нормализованное число. Для округления числа, представленного основными (старшими 36-ю) разрядами результата, к мантиссе добавляется число, изображающееся цифрой 1 в 37-м (младшем дополнительном) разряде. Если порядок произведения меньше, чем —64, или мантисса результата равна 0, то в ячейку по третьему адресу посылается 0. Если порядок результата больше 63, то происходит автоматический останов машины.

Деление в машине М-20 возможно, если мантисса делимого меньше удвоенной мантиссы делителя (для нормализованных чисел это условие всегда выполняется). Если это условие не выполнено, то происходит автоматический останов («авост») машины. «Авост» происходит и в случае, когда результат не может быть представлен в машине, т. е. если порядок результата больше 63, в частности, при попытке деления на нуль.

При делении вычисляются 37 разрядов мантиссы, а затем производится округление прибавлением 1 в младший дополнительный (37-й) разряд.

Аналогично выполняется операция извлечения корня. В ней автоматический останов происходит лишь в случае отрицательного операнда.

Нормализация результата операций деления и извлечения корня не производится. Если операнды нормализованы, то результат автоматически получается нормализованным.

§ 2. Модификации арифметических операций

Кроме операции обычного сложения (восьмеричный код 01, символическое обозначение $+$), существуют следующие модификации операции сложения:

- $+a$ (21) — сложение с блокировкой округления; при выполнении этой операции никогда не посылается 1 в младший дополнительный разряд;
- $,+a$ (41) — сложение с блокировкой нормализации; при выполнении этой операции не производится нормализация влево, однако нормализация вправо, если она нужна, происходит обязательно;
- $,+a$ (61) — сложение с блокировкой нормализации и округления.

Такие же модификации существуют для операций вычитания, вычитания модулей и умножения. Приведем их символические обозначения и восьмеричные коды:

	Блокировка округления	Блокировка нормализации	Блокировка округления и нормализации
Вычитание	$-a$ (22)	$, -$ (42)	$, -a$ (62)
Вычитание модулей	$ - - a$ (23)	$, - $ (43)	$, - a$ (63)
Умножение	$\times a$ (25)	$, \times$ (45)	$, \times a$ (65)

(К основному коду операции прибавляется соответственно 20, 40 и 60.)

Очевидно, что при выполнении операции вычитания модулей мантиссы операндов всегда вычитаются и поэтому добавления 1

в младший дополнительный разряд не происходит, т. е. команды $| - | a$ и $| - | a$ не отличаются от $| - |$ и $| - |$ соответственно.

Результат операций сложения, вычитания и вычитания модулей, если нормализация не заблокирована, всегда получается нормализованным. Результат операции умножения может оказаться ненормализованным (в случае ненормализованных операндов), так как сдвиг влево при нормализации производится не более чем на один разряд.

П р и м е р ы:

$$\begin{array}{r} + 0 \ 10001 \ 11011 \\ + 1 \ 10010 \ 10010 \end{array}$$

Знаки операндов различны, производится вычитание мантисс, ожидаемый знак результата $+$, при выравнивании порядков мантисса первого операнда сдвигается вправо на 1 разряд:

$$\begin{array}{r} 0 \ 10010 \quad - \quad 011011 \ 1 \\ (1) \ 10010 \quad - \quad 100101 \ 0 \\ \hline 0 \ 10010 \quad 1 \ 110110 \ 1 \end{array}$$

Мантисса разности отрицательна, знак результата изменяется на обратный, мантисса заменяется дополнительным кодом 0 001001 1 ($1,1101101 + 0,0010011 = 10,0000000$). Поскольку в коде операции предусмотрена блокировка нормализации, нормализация влево не делается. Результат:

$$1 \ 10010 \ 001001$$

(единица младшего дополнительного разряда отбрасывается).

Рассмотрим вычитание тех же чисел с блокировкой округления и нормализации

$$\begin{array}{r} - 0 \ 10001 \ 11011 \\ - 1 \ 10010 \ 10010 \end{array}$$

После выравнивания порядков:

$$\begin{array}{r} 0 \ 10010 \quad + \quad 011011 \ 1 \\ (1) \ 10010 \quad + \quad 100101 \ 0 \\ \hline 0 \ 10010 \quad 1 \ 000000 \ 1 \end{array}$$

Нормализация вправо производится, несмотря на признак блокировки нормализации в коде операции.

Результат:

$$0 \ 10011 \ 100000$$

В машинах типа М-20 существует операция *выдачи младших разрядов произведения* ($\times b$; 47), позволяющая получить младшие разряды мантиссы произведения. Команда записывается в виде:

$$\pi \times b \ 0 \ 0 \ a$$

и имеет смысл, если непосредственно перед нею выполнялась операция $\times a$ или $\times a$ (округление, как мы видели, портит содержи-

мое младших разрядов). При выполнении этой команды по третьему исполнительному адресу посылаются признак и порядок произведения и вторые 36 разрядов мантиссы произведения. Значения первого и второго исполнительных адресов команды на выполнение операции не влияют.

Выполнение пары операций $\times a$ (или $, \times a$) и $\times b$ подробнее можно описать следующим образом. Пусть перемножаются числа $\pm 2^{p_1} \cdot m_1$ и $\pm 2^{p_2} \cdot m_2$. Абсолютные величины мантисс сомножителей можно представить как целые числа, умноженные на 2^{-36} : $m_1 = 2^{-36} \cdot M_1$, $m_2 = 2^{-36} \cdot M_2$. Произведение исходных чисел равно $\pm 2^p \cdot m$, где: $p = p_1 + p_2$, $m = m_1 m_2$ при $m_1 m_2 \geq 1/2$, а также если в коде первой операции пары предусмотрена блокировка нормализации, $p = p_1 + p_2 - 1$, $m = 2m_1 m_2$ при $m_1 m_2 < 1/2$. Абсолютную величину m мантиссы произведения, содержащую 72 двоичных разряда, можно представить в виде $m = 2^{-72} \cdot M$, где M , равное $M_1 M_2$ или $2M_1 M_2$, — целое число ($M < 2^{72}$). Пусть N_1 и N_2 — соответственно частное и остаток от деления M на 2^{36} : $M = 2^{36} N_1 + N_2$, $N_1, N_2 < 2^{36}$. Обозначим $n_1 = 2^{-36} \cdot N_1$, $n_2 = 2^{-36} \cdot N_2$. Результатом операции $\times a$ ($, \times a$) будет число $2^p \cdot n_1$, результатом операции $\times b$ — число $2^p \cdot n_2$. Обоим результатам приписывается один и тот же знак, признак и, как только что было указано, порядок p . Если все 36 разрядов мантиссы результата операции $\times b$ равны нулю, то вырабатывается признак $\omega = 1$, в противном случае $\omega = 0$.

Операции деления и извлечения корня имеют еще по одной модификации:

$\backslash a$ (24) — деление с блокировкой округления,

$\sqrt{a}$ (64) — извлечение корня с блокировкой округления.

Если операция умножения, деления или извлечения корня выполняется без блокировки округления и ее результат отличен от нуля, то он всегда является хорошим приближением к точному результату. Разумеется, если операнды не были нормализованы, то после нормализации результата он может уже перестать быть хорошим приближением к точному результату (его машинный порядок p уменьшится, а следовательно, изменится правая часть неравенства (13.1.2), тогда как левая часть не меняет своего значения при нормализации числа x').

Следующие две операции можно рассматривать как модификации операций $+AP$ и $-AP$.

Сложение порядка с порядком ($+ПП$; 26). При выполнении этой операции оба операнда считаются из ОЗУ по первому и второму исполнительному адресу. Мантисса результата совпадает с мантиссой (включая ее знак) второго операнда. Порядок результата равен сумме порядков операндов, если она не превосходит 63, в противном случае происходит «авост». Если сумма порядков операндов меньше -64 , то в качестве результата операции вырабатывается машинный ноль. Признак результата (цифра 45-го разряда)

вырабатывается так же, как для всех арифметических операций, оба операнта которых считываются из ОЗУ. Признак ω вырабатывается так же, как при операции умножения.

Вычитание порядка из порядка ($-ПП$; 66). Эта операция отличается от предыдущей лишь тем, что при ее выполнении порядок первого операнта вычитается из порядка второго операнта. Полученная разность принимается за порядок результата (если это возможно).

При выполнении этих операций сложение или вычитание порядков операндов производится в той части арифметического устройства, где складываются или вычитаются порядки при операциях умножения и деления. Там же происходит сложение (вычитание) порядков при выполнении операции $+АП$ ($-АП$). Но при этом приходится пересылать содержимое семи младших разрядов первого исполнительного адреса из того регистра, где он вычисляется (сумматора адресов), в разряды порядка первого операнта. На это тратится дополнительное время, поэтому операции $+ПП$ и $-ПП$ выполняются существенно быстрее, чем операции $+АП$ и $-АП$. Таким образом, из четырех способов реализации оператора $y := x/2$

0	/	x	$\langle 2 \rangle$	y
0	$\times$	x	$\langle 1/2 \rangle$	y
0	$-АП$	101	x	y
0	$-ПП$	$\langle 1 \rangle$	x	y

каждый следующий заметно быстрее предыдущего. Аналогично, команда

0 +ПП $\langle 1 \rangle$ x y

быстрее любой другой выполняет присваивание $y := 2 \times x$, хотя и ненамного быстрее, чем команда

0 + x x y

работающая быстрее, чем

0 +АП 0101 x y

§ 3. Примеры применения модификаций арифметических операций

Пример 1. Пусть в ячейке x находится число, меньшее чем 2^{36} , и мы хотим вычислить его целую часть (если $x \geq 2^{36}$, то целую часть такого числа в машине точно представить нельзя).

В математике целая часть x ($[x]$) определяется, как целое число, удовлетворяющее следующим неравенствам:

$$[x] \leq x < [x] + 1.$$

В машине вычислить $[x]$ для $x < 0$ одной операцией нельзя; для отрицательных чисел будем вычислять $-[-x]$.

В ячейку c поместим константу:

$$c \ 1 \ 45 \ 0000 \ 0000 \ 0000 \quad (2^{37} \cdot 0)$$

и выполним операцию

$$0 \ + \ a \ x \ c \ y$$

При выравнивании порядков мантисса числа x сдвинется так, что младшая единица целой части $|x|$ попадет в младший дополнительный разряд, все разряды дробной части пропадут. Затем результат нормализуется, и в ячейку y попадет $\text{sign } x \cdot [|x|]$ в виде нормализованного числа.

Пример 2. Пусть в ячейке n хранится неотрицательное целое число, меньшее 2^{12} . Как заслать это число на индекс-регистр? Для этого достаточно выполнить две операции

$$\begin{array}{l} 0 \ ,+ \ c_1 \ n \ r \ \text{«}130 \ 0000 \ +n \ 0000\text{»} \\ 0 \ KI \ 0 \ r \ 0 \ I := n \end{array}$$

где c_1 — адрес константы

$$1 \ 30 \ 0000 \ 0000 \ 0000 \quad (2^{21} \cdot 0)$$

В результате выравнивания порядков при сложении разряд единиц числа n займет положение младшего разряда второго адреса, и так как нормализация заблокирована, код результата в ячейке r будет иметь указанный в примечании вид.

Пример 3. Обратная посылка содержимого индекс-регистра в ячейку в виде нормализованного целого числа может быть осуществлена командами:

$$\begin{array}{l} \quad \quad \quad (AI) \\ 4 \ AI \quad 0 \quad 0 \ n \ \text{«}0 \ 52 \ 0000 \ I \ 0000\text{»} \\ 0 \ +AI \ 0156 \ n \ n \ \text{«}1 \ 30 \ 0000 \ I \ 0000\text{»} \\ 0 \ + \quad n \quad 0 \ n \ n := I \quad (\text{нормализация}) \end{array}$$

Если последнюю команду заменить на

$$0 \ + \ n \ c_1 \ n$$

при прежней константе c_1 , то сложение произойдет быстрее (не потребуется тратить время на выравнивание порядков).

Пример 4. Пусть в ячейке x хранится положительное число, меньшее 1. Старшая цифра его разложения в десятичную дробь равна $\alpha_1 = [10x]$. Чтобы определить следующие цифры, желательно заменить число x числом $x' = \{10x\}$ (скобки $\{ \}$ — символ дробной части). Один из простейших способов нахождения α_1 и x' состоит в следующем. Предварительно денормализуем число x , так, чтобы его порядок стал равен нулю. Это можно сделать, выполнив команду

$$0 \ ,+ \ c_2 \ x \ x$$

где c_2 — адрес константы

1 00 0000 0000 0000 $(2^9 \cdot 0)$

Более экзотический способ:

0 $\leftarrow$ МП $x \ x \ x$ Сдвиг мантиссы

0 — ПП $x \ x \ x$ Присваивание нулевого порядка

После этого выполним две команды

0 $\times a \ c_3 \ x \ \alpha \ \alpha_1$ в младших разрядах

0 $\times b \ 0 \ 0 \ x \ x := \{10x\}$

где c_3 — адрес константы

1 00 0000 0000 0012 $2^9 \cdot (10 \cdot 2^{-36})$

Понять, почему эти команды действительно приводят к желаемому результату, проще всего, рассмотрев результат умножения десятичной дроби 0,00008 на пятиразрядную правильную десятичную дробь x . Пусть десятичная запись произведения имеет вид

$$0,0000\beta_5\beta_6\beta_7\beta_8\beta_9\beta_{10}.$$

Тогда $\beta_5 = [8x]$, а десятичная дробь вида

$$0,\beta_6\beta_7\beta_8\beta_9\beta_{10}$$

(младшие разряды произведения) численно равна $\{8x\}$.

Пример 5. Пусть в ячейках m_1 и n_1 хранятся коды:

(АН)
 m_1 0 52 0000 m 0000
 n_1 0 52 0000 n 0000

Требуется сформировать заготовку с произвольным кодом операции, содержащую произведение $mn < 2^{12}$ в разрядах первого адреса. Для этого достаточно выполнить команды

0 $\times a \ m_1 \ n_1 \ 0$ «0 24 0000 0000 0000»

0 $\times b \ 0 \ 0 \ z$ «0 24 mn 0000 0000»

У п р а ж н е н и я. 1. Написать программу вычисления $[x]$ для произвольного x из диапазона $-2^{36} + 1 \leq x < 2^{36}$.

2. Написать программу вычисления $\{x\}$ для x из того же диапазона (без потери точности).

Глава 14. ВВОД, ВЫВОД И ОБМЕН С ВНЕШНЕЙ ПАМЯТЬЮ

§ 1. Устройства ввода-вывода машин типа М-20

Устройства ввода и вывода вместе с магнитными барабанами и магнитофонами принято называть *внешними устройствами* машины. Машина М-20 — это довольно устаревшая машина, обладающая сравнительно бедным набором внешних устройств. Но это облегчает нашу задачу — познакомиться с основными методами работы с этими устройствами. Более того, даже для машин типа М-20 мы не будем изучать организацию работы с внешними устройствами во всех деталях.

Машины типа М-20 имеют лишь один вид устройств ввода — читающее устройство для ввода с перфокарт. *Перфокарта* (рис. 14.1.1) — это листок специальной плотной бумаги определенного формата, на котором в определенных позициях могут быть пробиты отверстия. Эти позиции располагаются на пересечении 12 строк и 80 колонн, однако пробивки лишь в 47 колоннах принимаются во внимание при вводе данных в машину. На рис. 14.1.1 последняя непустая (10-я) строка перфокарты содержит пробивки во всех этих позициях, которые мы будем называть *рабочими*. Перфокарта поступает в устройство ввода своей верхней широкой стороной. В этом устройстве поочередно исследуются строки перфокарты. Каждая строка может содержать код одного из трех видов: адресный код, слово, подлежащее записи в ячейку ОЗУ, и признак конца вводимого массива. Эти коды различаются по пробивкам в двух колоннах (18-й и 80-й) перфокарты, называемых *маркерами*: *основным* и *дополнительным*. Дополнительный маркер отличает *управляющие* коды — адресный код и признак конца ввода — от *основных* кодов, которые должны быть записаны в память. Остальные 45 из 47 рабочих колонн перфокарты соответствуют 45 разрядам ячейки. Они нумеруются так же, как разряды ячейки.

За один прием (по одной команде ввода) вводится колода перфокарт, содержащая произвольное число кодов. Последним из этих кодов должен быть *признак конца вводимого массива*, содержащий как дополнительный, так и основной маркер. Если признак конца не встретился, то читающее устройство продолжает работать,

даже если колода перфокарт кончилась. Напротив, если признак конца ввода пробит на одной из перфокарт в середине колоды, то после обнаружения этого кода читающим устройством ввод прекращается. Оставшиеся карты могут быть введены, лишь если поступит новая команда ввода. На рис. 14.1.1 признак конца ввода пробит в 10-й строке перфокарты.

Адресный код не имеет основного маркера. Из 45 возможных пробивок в основных разрядах адресного кода 12 пробивок в разрядах 36 ÷ 25 (разрядах первого адреса при записи команды) определяют адрес ячейки, в которую запишется первый следующий за этим адресным кодом основной код. Следующие основные коды попадают в следующие ячейки с последовательно возрастающими номерами. Так продолжается до тех пор, пока на перфокартах не встретится признак конца ввода или новый адресный код, определяющий новый начальный адрес для ввода следующей группы основных кодов. Число таких групп (и, следовательно, число адресных кодов) во вводимом массиве может быть любым. Число основных кодов в группе может меняться от 1 до 4095 (число ячеек ОЗУ). На рис. 14.1.1 перфокарта содержит два адресных кода (1-я и 8-я строки) и две группы основных кодов (строки со 2-й по 7-ю и группа из одной 9-й строки).

Место ввода очередной группы может перекрываться с участками памяти, на которые вводились предшествующие группы. Ясно, что это очень облегчает внесение исправлений в программу. Если нужно исправить одну или несколько следующих друг за другом команд, то нет необходимости перебивать всю перфокарту с этими командами. Достаточно подготовить перфокарту с адресным кодом, содержащим адрес первой из этих команд, и правильными кодами команд. Эта перфокарта помещается в конце массива, перед признаком конца вводимого массива. Признак конца обычно пробивают на отдельной перфокарте в ее первой строке, оставляя остальные строки пустыми. Пустая строка не содержит никаких маркеров, этим она отличается от основного кода, обязательно снабжаемого основным маркером. Таким образом, строка перфокарты, содержащая единственную пробивку в основном маркере, представляет основной код, при вводе которого соответствующая ячейка ОЗУ заполнится нулями во всех разрядах.

Упражнение 1. Что произойдет после ввода в машину перфокарты, изображенной на рис. 14.1.1?

Для пробивки перфокарт служат специальные устройства, называемые *устройствами подготовки данных*. Поскольку они не имеют непосредственной связи с машиной, они не относятся к числу ее внешних устройств. На машинах типа М-20 применяются два вида устройств подготовки данных. Устройства одного вида позволяют пробивать коды команд и десятичных чисел, не считая, конечно,

управляющих кодов. Устройства другого вида дают возможность перфорировать также *тексты* — произвольные последовательности *литер* (печатных знаков).

Перед перфорацией программа переписывается в восьмеричных обозначениях на бланках специальной формы (см. § 5 гл. 5 и гл. 6). Если программа состоит из нескольких программных массивов (см. § 6 гл. 7), то перед каждым из них в бланк записывается адресный код с адресом первого слова массива. Адресный код отмечается символом **A** в специальной графе бланка. Строка перфокарты, содержащая признак конца массива, отмечается символом **Σ**. В табл. 1 приведена часть бланка, по которой была пробита перфокарта, изображенная на рис. 14.1.1.

Таблица 1

А	π	КОП	A ₁	A ₂	A ₃		Примечания *
			0021			A	
0021	0	10	0100	0000	0000		Ввод
2	0	50	0013	0000	7777		} Запись на барабан
3	0	70	0100	0000	0000		
4	0	50	4411	0000	7777		
5	0	70	0100	0022	0000		} Фиктивное чтение с барабана
0026	0	56	0000	0100	0000		
			0022			A	Переход на начало
0022	0	50	0015	0000	7777		Исправление
	7	77	7777	7777	7777	Σ	

Первый столбец бланка («Адрес команды») и последний («Примечания») не перфорируются и служат лишь для справок.

В таком же виде, в каком записываются команды, на бланках могут быть записаны, а потом отперфорированы и введены в машину любые машинные слова (заготовки, логические векторы и пр.). В частности, так же записываются и перфорируются восьмеричные коды двоичных чисел (см. § 1 гл. 5). Когда говорят о перфорации чисел, обычно имеют в виду особый режим перфорации, позволяющий перфорировать и затем вводить в машину десятичные числа в так называемом *двоично-десятичном представлении*. В отличие от двоичного это представление не связано с конструкцией машины и определяется только особенностями работы устройств подготовки данных. Два вида устройств подготовки данных для машины М-20 используют различные представления десятичных чисел. Мы опишем лишь одно из них.

В этом представлении 44-й разряд ячейки отводится под знак числа, 43-й разряд — под знак десятичного порядка (в обоих случаях знак минус кодируется единицей, а плюс — нулем). В $42 \div 41$ разрядах записывается старшая цифра десятичного порядка, в $40 \div 37$ разрядах — младшая цифра. Далее следует 9 групп по 4 разряда, в которых записываются цифры мантиссы числа. Каждая цифра изображается в отведенных для нее разрядах ее двоичным кодом. Старший (45-й) разряд ячейки, как и при записи чисел в двоичном коде, отводится под признак числа.

Если обозначить код знака мантиссы s_q , а код знака порядка s_p , то слово

π	s_q	s_p	β_2	β_1	α_1	α_2	...	α_9
45 p.		43 p.	42 и 41 p.	40—37 p.	36—33 p.	32—29 p.	...	4—1 p.

(1)

изображает число

$$(1 - 2s_q) \cdot 10^{(1-2s_p)(10\beta_2 + \beta_1)} \cdot (\alpha_1 \cdot 10^{-1} + \alpha_2 \cdot 10^{-2} + \dots + \alpha_9 \cdot 10^{-9}).$$

На бланке десятичное число записывают следующим образом:

A	π	КОП	A_1	A_2	A_3	Примечание
	$\pm \pm$	$\pm \beta_2 \beta_1$	$\alpha_1 \alpha_2 \alpha_3$	$\alpha_4 \alpha_5 \alpha_6$	$\alpha_7 \alpha_8 \alpha_9$	Число

Слово «Число» в графе «Примечания» служит указанием перфораторщице о необходимости переключить устройство подготовки данных в режим перфорации десятичных чисел. Числа 5; $-0,001$; 3,14159265 (с признаком) запишутся на бланке следующим образом:

A	π	КОП	A_1	A_2	A_3	Примечание
	++	+01	500	000	000	} Числа
	+-	-02	100	000	000	
	-+	+01	314	159	265	

Перфокарта, на которой пробиты эти числа, изображена на рис. 14.1.2. Ясно, что эта запись очень далека от принятого в машинах М-20 способа представления двоичных чисел. Перевод из

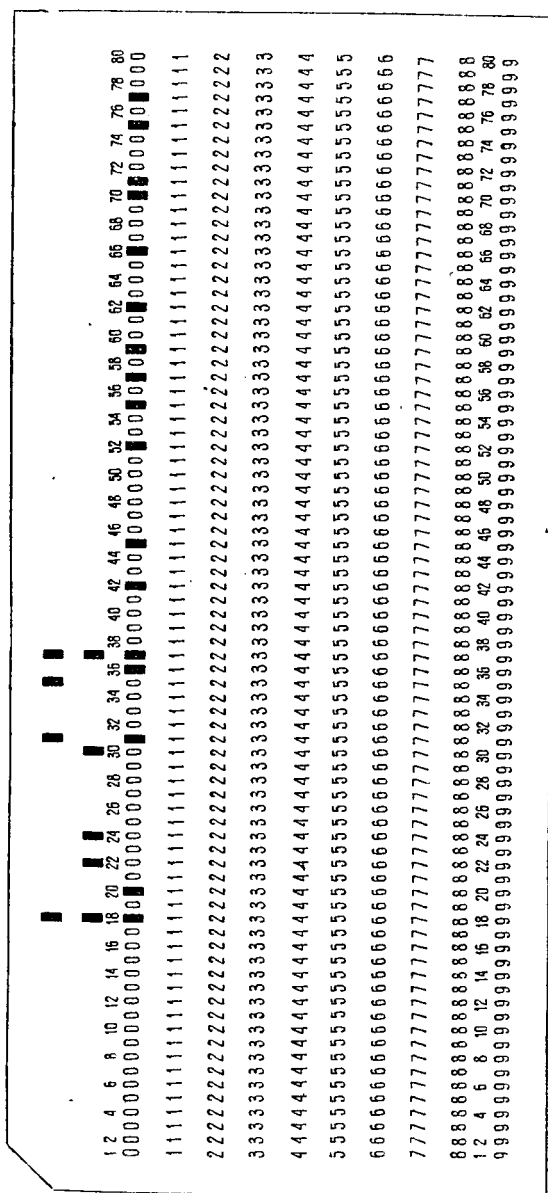


Рис. 14.1.2. Пробивка чисел.

одного представления в другое осуществляется по специальным программам, работу которых мы разберем позже (см. § 5).

При пробивке текста (если используется устройство, позволяющее это делать) в одной строке перфокарты размещается от одного до шести кодов литер. Код первой литеры занимает разряды с 42-го по 36-й, второй — разряды с 35-го по 29-й и т. д. Разряды с 45-го по 43-й остаются свободными, их содержимое не принимается во внимание при работе с текстами. О приемах обработки текстов очень кратко было сказано в § 4 гл. 12. Возможность вводить и обрабатывать тексты чрезвычайно важна в большинстве применений ЭВМ. Достаточно напомнить, что без нее было бы недоступно применение алгоритмических языков. Вывод результатов работы программы в виде связного текста или таблиц также облегчает восприятие этих результатов человеком.

Перейдем к описанию устройств вывода. Для вывода на печать применяются два вида устройств. Одно из них служит для печати на узкой (80 мм) бумажной ленте содержимого ячеек памяти в один столбец. Устройство может работать в двух режимах. В режиме *восьмеричной* печати содержимое каждой ячейки печатается в виде 15 восьмеричных цифр, т. е. в таком виде, в каком записываются на бланках для перфорации команды и двоичные числа. В режиме *десятичной* печати содержимое ячейки делится на группы так, как при вводе десятичных чисел в двоично-десятичном представлении (см. выше), и печатается в таком же виде, в каком эти числа записываются на бланках. Каждой из одноразрядных групп π , s_q и s_p соответствует при печати знак + или —, двухразрядной группе β_2 — четверичная цифра (0, 1, 2 или 3), четырехразрядным группам β_1 , α_1 , ..., α_9 — десятичные цифры. Однако четырехразрядные группы могут, в общем случае, содержать не только коды десятичных цифр. Коду 1010 при выводе на печать соответствует знак +, коду 1011 соответствует знак —, кодам 1100, 1101, 1110 и 1111 — отсутствие литеры (пробел). Таким образом, осмысленная и однозначно расшифровываемая печать в этом режиме получается лишь в том случае, если выводимые ячейки заполнены двоично-десятичными кодами чисел.

Другое устройство печати, так называемое АЦПУ (алфавитно-цифровое печатающее устройство), служит для печати на широкой бумажной ленте произвольных текстов. Эти тексты могут быть составлены из тех же литер, которые можно отперфорировать на картах. В одной печатной строке умещается 128 литер. Для печати литеры текста кодируются семиразрядными двоичными числами, которые размещаются по 6 в ячейке, так же как при вводе с перфокарт. Информация об одной печатной строке занимает до 22 ячеек. Не каждое семиразрядное двоичное число кодирует какую-либо литеру. Некоторые свободные коды используются для управления печатью. С их помощью можно перейти к печати следующей строки,

не закончив заполнение литеры текущей строки, создавать разрывы в строке и интервалы между строками, а также накладывать очередную печатаемую строку на предыдущую. При таком наложении можно отпечатать отсутствующие в алфавите устройства литеры $\leq$ (наложение литеры $<$ и подчеркивания), $\geq$, $\equiv$, Φ (0 и /), $\div$ и т. п.

Кроме вывода на печать, возможен вывод на перфокарты. За один прием можно вывести содержимое программного массива длиной не более 510 ячеек. Каждой ячейке соответствует одна строка перфокарты. Между разрядами ячейки и колонками перфокарты сохраняется то же соответствие, что при вводе с перфокарт. Каждая выводимая строка перфокарты снабжается основным маркером. Есть возможность в конце перфорируемого массива отперфорировать признак конца массива (строку с основным и дополнительным маркерами). Адресные коды отперфорировать невозможно. Таким образом, понятие массива вывода совпадает с понятием программного массива, тогда как массив ввода, благодаря наличию в нем адресных кодов, может состоять из нескольких программных массивов.

§ 2. Внешние запоминающие устройства

Машина М-20 может иметь в своем составе до четырех магнитных барабанов. Этим барабанам в программах даются двоичные номера 00, 01, 10 и 11. На каждом барабане можно записать до 4096_{10} машинных слов в ячейки, снабженные номерами от 0000 до 7777_8 . Полный адрес ячейки на барабане получается объединением номера барабана с номером ячейки. За один прием можно записать на барабан из ОЗУ или считать с барабана в ОЗУ программный массив любой длины от 1 до 4095_{10} слов. Соседним ячейкам ОЗУ соответствуют при этих операциях соседние ячейки барабана. Начальные адреса массивов в ОЗУ и на барабане не обязаны совпадать. Каждый магнитный барабан как бы склеен в кольцо — за его ячейкой с номером 7777 следует ячейка с номером 0000. Таким образом, программный массив, заполняющий ячейки с 0001 по 0200 ОЗУ, может быть записан в ячейки $7701 \div 0100$ любого барабана (или считан из этих ячеек). Оперативное запоминающее устройство кольца не образует, но лишь потому, что в ячейку ОЗУ с адресом 0000 нельзя записать или считать из нее ничего, кроме нулевого слова.

К машине М-20 можно подключить до четырех магнитофонов, снабженных номерами 00, 01, 10 и 11. На каждом магнитофоне можно устанавливать и снимать катушки с магнитной лентой. Машина не в состоянии различать разные магнитные ленты, поэтому забота о том, чтобы на магнитофонах стояли ленты, соответствующие выполняемой программе, ложится на плечи программиста и оператора, работающего за пультом машины.

На каждой магнитной ленте можно записать до 4095_{10} групп машинных слов, называемых *зонами*. Каждая зона может состоять из любого числа слов от 1 до 4096_{10} , однако общий объем данных на ленте (сумма числа слов по всем зонам) ограничен фактической длиной ленты и составляет на разных моделях машин от 75 тысяч до 1 миллиона слов.

Ленты могут двигаться в двух направлениях: *прямом* и *обратном*. Для нормальной работы с лентами требуется, чтобы при движении ленты в прямом направлении номера зон, записанных на ленте, возрастали. Допускается, чтобы номера соседних зон разнились более чем на единицу.

Запись, имеющаяся на ленте, может быть стерта (без участия машины, как на обычном магнитофоне). Для записи на чистую (стертую) ленту существует особый режим работы с лентой, называемый режимом *записи с разметкой*. За один прием (по одной команде) можно записать с разметкой одну новую зону. Запись происходит при движении ленты в прямом направлении на то место ленты, которое перед началом выполнения команды находилось под записывающими головками магнитофона. Для нормальной работы машины требуется, чтобы этот участок ленты был чист. Поэтому если размечается не самая первая зона, то операцию разметки следует выполнять после разметки или считывания последней из имеющихся на ленте зон.

На уже размеченной (не обязательно до конца) ленте можно считать в ОЗУ содержимое любой из зон или записать в любую из зон содержимое некоторого участка ОЗУ. Чтение или запись начинается обязательно с первого слова зоны. Длина считываемого или записываемого участка может быть меньше числа слов в зоне, но обычно зону считывают или переписывают полностью. Начальный адрес считываемого или записываемого массива в ОЗУ может быть любым, однако ячейка с адресом 0009 не должна участвовать в операции *обмена* (под этим термином объединяют все операции передачи данных из ОЗУ во внешнюю память и обратно).

На других моделях машин типа М-20 (кроме самой М-20) емкость магнитных барабанов и число магнитофонов может быть больше, чем было сказано выше, но мы не будем рассматривать эту возможность (так же как возможность увеличенной емкости ОЗУ).

§ 3. Контроль операций ввода, вывода и обмена

Операции ввода и вывода, в которых участвуют электромеханические устройства, и операции обмена, при которых происходит запись или считывание с поверхности движущегося магнитного слоя, обладают существенно меньшей надежностью, чем операции, в которых участвуют чисто электронные арифметическое устройство и устройство управления машиной и оперативное запоминающее

устройство на ферритах. Поэтому операции ввода, вывода и обмена с внешней памятью обычно сопровождаются контролем. Действия по контролю этих операций находят отражение в соответствующих командах. На некоторых машинах контролируются и операции в центральном устройстве, но этот контроль осуществляется чисто схемными средствами, без участия программы, и о нем мы больше упоминать не будем.

В связи с организацией контроля операций ввода, вывода и обмена рассмотрим еще три операции, существующие в системе команд машин типа М-20.

Циклическое сложение (+Ц; 07). У этой операции два операнта, считываемые, как обычно, по первому и второму исполнительному адресу команды. Каждый из этих операндов делится на две части, одна из которых образована старшими девятью разрядами слова, другая — всеми оставшимися. Соответствующие части обоих операндов складываются, как целые двоичные числа. Однако, если при сложении частей возникает единица переноса из старшего (45-го или 36-го) разряда части, то она добавляется к младшему (37-му или 1-му) разряду соответствующей части результата.

Пример 1. Если операция циклического сложения выполняется над словами

301	2615	3730	1147
434	7772	0004	7776

то сумма старших разрядов будет равна 735 и в таком виде войдет в результат. Сумма младших разрядов равна 1 2607 3735 1145. Получившаяся единица 37-го разряда добавится к младшему разряду. Окончательно, результат операции равен

735	2607	3735	1146

Результат операции записывается в ячейку по третьему исполнительному адресу команды. Если один из операндов операции циклического сложения равен

777	7777	7777	7777

то результат операции, как правило, совпадает с другим операндом.

У п р а ж н е н и я. 1. Каковы исключения из этого правила?

2. Показать, что операция циклического сложения коммутативна и ассоциативна, т. е.

$$a +_{\text{Ц}} b = b +_{\text{Ц}} a,$$

$$(a +_{\text{Ц}} b) +_{\text{Ц}} c = a +_{\text{Ц}} (b +_{\text{Ц}} c).$$

Ассоциативность операции циклического сложения позволяет говорить о *циклической сумме* любого числа слов.

Циклическое вычитание (—Ц; 27). Отличие этой операции от предыдущей состоит в том, что соответствующие части слов-операн-

тов не складываются, а вычитаются. Если при вычитании старших разрядов частей потребуется произвести заем единицы, то он делается из младшего разряда части.

Пр и м е р 2. Пусть над теми же кодами, что в примере 1, выполняется операция циклического вычитания. Поскольку обе части второго операнта больше, чем соответствующие части первого операнта, для каждой части следует произвести заем единицы из младшего разряда, т. е. заменить части первого операнта числами 1300 и 1 2615 3730 1146. В результате получим слово

644 2623 3723 1150

Сравнение с остановом ($\ncong a$; 35). Эта операция выполняется в точности так же, как операция $\ncong$, но если операнды различаются хоть в одном разряде, то машина останавливается. Если пустить ее вновь, то выполнение программы будет продолжаться так, как будто выполнялась операция сравнения без останова ($\cong$). Сравнение с остановом полезно в тех случаях, когда сравниваются слова, которые при нормальной работе машины (и программы) обязаны совпадать.

При всех операциях ввода, вывода и обмена в машинах типа М-20 передаваемые данные проходят через регистры арифметического устройства. За время передачи в этом устройстве вычисляется так называемая *контрольная сумма* всех передаваемых слов. Контрольная сумма программного массива — это циклическая сумма всех слов этого массива. Существенно, что в образовании контрольной суммы участвуют все разряды всех передаваемых слов и искажение хотя бы одного из этих разрядов приводит к изменению контрольной суммы. Более того, при искажении многих разрядов вероятность получения неискаженного значения суммы чрезвычайно мала.

Схема контроля такова. При выводе или записи во внешнюю память некоторого программного массива одновременно в арифметическом устройстве подсчитывается его контрольная сумма. Обычно эта контрольная сумма также выводится или записывается во внешнюю память. Так, 45 основных разрядов признака конца массива, выводимого на перфокарты, как раз и представляют контрольную сумму этого массива, а собственно признаком конца массива на перфокартах служат два маркера, сопровождающие эту сумму. При выводе на печать контрольная сумма печатается в таком же виде, как содержимое любой выводимой ячейки. Правильно прочесть и впоследствии использовать отпечатанную сумму можно лишь, если вывод идет в режиме восьмеричной печати. При записи массива на барабан или на ленту его контрольная сумма записывается в ячейку барабана или зоны ленты, непосредственно следующую за ячейками, в которые записывается сам массив. Таким образом, длина массива, записанного на барабан или ленту

с контролем, на единицу превышает число ячеек, занятых этим массивом в ОЗУ. Это следует иметь в виду при распределении внешней памяти.

При вводе массива или чтении его из внешней памяти также подсчитывается его контрольная сумма. Эта сумма сравнивается с суммой, вводимой или считываемой вместе с массивом. Контрольная сумма массива, вводимого с перфокарт, представлена 45 основными разрядами признака конца массива. Контрольная сумма массива, записанного на барабане или на ленте, считывается из ячейки, следующей за ячейками, занятыми самим массивом. Если сумма, подсчитанная в АУ, не совпадает с введенной или считанной суммой, то либо выполнение операции приостанавливается, либо происходит условный переход по адресу, указанному в команде ввода или чтения. Кроме того, подсчитанная сумма записывается в ОЗУ по адресу, также указанному в команде.

Действия по контролю операций ввода, вывода или обмена могут быть *заблокированы*. При выводе или записи это означает, что подсчитанная сумма не выводится или, соответственно, не записывается вместе с массивом. При вводе можно заблокировать остановку машины при несовпадении подсчитанной и введенной сумм или условный переход по несовпадению сумм. При чтении в ОЗУ из внешней памяти блокировка контроля означает, что контрольная сумма из внешней памяти не считывается, а поэтому и ни с чем не сравнивается, управление всегда переходит к следующей команде программы. Чтение с контролем массива, записанного во внешнюю память без контроля, никогда не может быть успешным (если, конечно, пренебречь крайне маловероятной возможностью случайного совпадения слова, хранившегося в ячейке внешней памяти непосредственно вслед за записанным массивом, с циклической суммой этого массива).

Очень полезна бывает возможность заблокировать занесение данных в ОЗУ при вводе или при чтении из внешней памяти. При этом происходит только суммирование поступающих в АУ кодов и сравнение вычисленной контрольной суммы с введенной или считанной. Таким способом легко проверить правильность выведенного или записанного перед этим во внешнюю память массива, ничего не портя в ОЗУ. При несовпадении сумм вывод или запись массива можно повторить.

В более современных машинах применяются другие схемы контроля. Кроме того, для контроля используется не центральное арифметическое устройство, а специальные вычислительные узлы, входящие в состав местных устройств управления вводом, выводом и обменом. Это позволяет сделать схему управления этими операциями значительно более гибкой, совместить их во времени с работой центрального процессора и ускорить выполнение программы.

§ 4. Операции ввода, вывода и обмена

Для ввода с перфокарт можно прибегнуть к одной из двух операций ввода. Операция *ввода с полным контролем* (символическое обозначение *ВВ*; восьмеричный код 10) выполняется следующим образом. Запускается читающее устройство и начинается ввод установленной в нем колоды перфокарт (до первого адресного кода), так, как будто перед ней был пробит адресный код с адресом, совпадающим с первым исполнительным адресом выполняемой команды. После поступления адресного кода ввод продолжается так, как было описано в § 1. При поступлении признака конца ввода содержащаяся в этом признаке контрольная сумма сравнивается с контрольной (циклической) суммой, накопленной в сумматоре. При несовпадении контрольных сумм машина останавливается, а если пустить ее вновь, то управление будет передано по второму исполнительному адресу команды. По третьему исполнительному адресу записывается в ОЗУ контрольная сумма, подсчитанная в арифметическом устройстве.

Операция *ввода с блокировкой останова (ВВБО; 30)* отличается от предыдущей лишь тем, что при несовпадении контрольных сумм управление передается по второму исполнительному адресу без остановки машины.

При выполнении операции *ВВ* можно также заблокировать остановку машины, если хотя бы один из адресных кодов содержит 1 в 1-м разряде (младшем разряде третьего адреса). При выполнении обеих операций можно заблокировать не только остановку, но и переход по второму исполнительному адресу, для этого в одном из адресных кодов надо вписать 1 в 12-й разряд кода. Наконец, запись в ОЗУ блокируется, если первый исполнительный адрес равен нулю или если адресный код содержит 1 в 13-м разряде (младшем разряде второго адреса). Действие этой блокировки распространяется только на группу основных кодов до следующего адресного кода или (для последней группы) до конца вводимого массива.

Пример 1. Пусть выполняется группа команд

Программа 1

p	0	<i>ВВ</i>	0000	$p+1$	r
$p+1$	0	<i>ВВБО</i>	0001	$p+2$	r_1
$p+2$	0	$\neq$	r	r_1	0
$p+3$	0	<i>ПО</i>	r_1	$p+1$	r

а на читающем устройстве стоит одна перфокарта, изображенная на рис. 14.1.1 и в табл. 14.1.1. Несмотря на то, что в команде p первый исполнительный адрес равен нулю, ввод начнется под действием адресного кода, пробитого в первой строке перфокарты. В ячейки 0021 ÷ 0026 введутся записанные в табл. 14.1.1 коды. Затем под действием второго адресного кода (8-я строка перфокарты) в ячейку 0022 вместо ранее введенного слова введется новый код.

Содержимое ячеек станет таким:

0021	0 10	0100	0000	0000
0022	0 50	0015	0000	7777
0023	0 70	0100	0000	0000
0024	0 50	4411	0000	7777
0025	0 70	0100	0022	0000
0026	0 56	0000	0100	0000

Контрольная сумма, пробитая в 10-й строке перфокарты, неверна и не совпадет с суммой, подсчитанной при вводе. Поэтому машина остановится. Если теперь введенную карту снова установить в читающее устройство и пустить машину, то подсчитанная контрольная сумма запишется в ячейку r , а управление будет передано команде $p + 1$. По этой команде карта введется вновь в те же ячейки. Несмотря на несовпадение сумм, остановки не будет, и сумма, подсчитанная при втором вводе, запишется в ячейку r_1 . Следующей будет выполняться команда $p + 2$. Она сравнит содержимое ячеек r и r_1 и, если оба ввода прошли правильно, выработает значение 1 признака ω . Следующая команда пропустит на выполнение команды $p + 4$. Если же одна из операций ввода выполнится неправильно, и суммы, записанные в ячейки r и r_1 , окажутся различными, то команда $p + 3$ передаст управление по адресу $p + 1$ на повторный ввод. Последняя контрольная сумма перешлется из ячейки r_1 в ячейку r .

Однако, если программа была написана в расчете на то, что цикл $p + 1 \div p + 3$ будет повторяться до совпадения контрольных сумм при двух последовательных вводах одного и того же массива, то этот расчет не оправдается. Почти мгновенно после окончания второго ввода управление (при несовпадении сумм в ячейках r и r_1) будет передано вновь команде $p + 1$, и читающее устройство начнет работать раньше, чем оператор машины успеет снова установить на нем ошибочно введенную карту. Начнется либо ввод другого массива (на читающем устройстве может быть установлено сразу несколько массивов, каждый со своим признаком конца), либо устройство будет работать вхолостую.

Чтобы в подобных случаях оператор имел возможность подготовить повторный ввод неправильно введенных перфокарт, в команде $p + 2$ следует воспользоваться операцией сравнения с остановом ($\neq a$). Таким образом, разумно составленная программа ввода массива с неизвестной контрольной суммой может быть такой:

Программа 2

p	0	BB	a	$p + 1$	r	Первоначальный ввод
$p + 1$	0	$BBBO$	a	$p + 2$	r_1	Повторный ввод
$p + 2$	0	$\neq a$	r	r_1	0	Останов при несовпадении сумм
$p + 3$	0	$П0$	r_1	$p + 1$	r	$r := r_1$, на повторный ввод (при несовпадении сумм)

Здесь a — адрес ячейки, в которую должно попасть первое слово вводимого массива, если в нем нет соответствующего адресного кода.

У п р а ж н е н и я. 1. В чем недостаток программы

p	0	BB	a	$p+1$	r
$p+1$	0	BB	a	$p+2$	r_1
$p+2$	0	$\neq$	r	r_1	0
$p+3$	0	$II0$	r_1	$p+1$	r

также предназначенной для ввода массива с неизвестной контрольной суммой?

2. Составить программу для ввода массива, правильная контрольная сумма которого известна и пробита в составе признака конца ввода этого массива.

Операции вывода на машине М-20 мы рассмотрим совместно с операциями обмена. По существу, все они являются вариантами одной и той же операции. Эта операция задается двумя командами. Первая команда (символическое обозначение Ma ; восьмеричный код 50) лишь подготавливает операцию и только вторая команда (Mb ; 70) вызывает ее выполнение. Можно рассматривать эту пару команд как одну шестиадресную команду.

Пара команд Ma — Mb напоминает пару команд $\times a$ — $\times b$, задающих операцию умножения с удвоенной разрядностью результата. Но в случае умножения первая из команд пары имеет и самостоятельное значение. Напротив, команда Ma , если за ней не следует команда Mb , не имеет никаких последствий. Попытка выполнить команду Mb без предшествующей ей команды Ma приводит к «авосту».

Адреса (исполнительные) команд Ma и Mb имеют, как правило, следующий смысл. Первый адрес команды Mb задает начало, а третий адрес команды Ma — конец участка оперативной памяти (программного массива), участвующего в операции обмена. Второй адрес команды Ma задаст либо начальный адрес группы ячеек на барабане, либо номер зоны на ленте, участвующих в операции. Вторым адресом команды Mb — это адрес команды, которой передается управление в случае несовпадения контрольных сумм при чтении с барабана или ленты. Если выполняется запись во внешнюю память или вывод, то значение этого адреса не играет никакой роли. Третий адрес команды Mb — это адрес ячейки ОЗУ, в которую записывается контрольная сумма, подсчитанная в арифметическом устройстве. Наконец, первый исполнительный адрес команды Ma — это так называемое *условное число*, определяющее вид (вариант) операции. Впрочем, два младших разряда этого адреса (26-й и 25-й разряды кода команды), по существу, к условному числу не относятся. Они содержат номер либо барабана, либо магнитофона, участвующего в операции обмена. Таким образом эти разряды как бы наращивают второй адрес команды Ma , указывая, к какому именно барабану или ленте он относится. В случае вывода

на печать или на перфорацию значение этих разрядов не влияет на выполнение операции.

Собственно условным числом является содержимое разрядов с 36-го по 27-й команды *Ma*. Каждый из этих разрядов играет более или менее самостоятельную роль, и лишь в редких случаях содержимое разряда по-разному влияет на выполнение операции в зависимости от значения других разрядов условного числа.

Единица в 28-м разряде означает, что выполняется операция обмена с барабаном. Единица в 29-м разряде — что выполняется обмен с лентой по имеющейся на ней разметке. Единица в 30-м разряде служит признаком операции записи на ленту с разметкой. Единица в 31-м разряде означает вывод на печать, а в 32-м разряде — вывод на перфорацию. В этих пяти разрядах должна содержаться, как правило, лишь одна единица. Если эта единица стоит в 28-м или в 29-м разряде, то содержимое 27-го разряда определяет вид операции обмена. Единица в этом разряде задает операцию записи на барабан или ленту, а ноль — операцию чтения. При операциях обмена с лентой единица в 34-м разряде означает, что для поиска нужной зоны лента начинает двигаться в обратном направлении. При нуле в этом разряде поиск начинается при прямом направлении движения ленты. Однако, если в процессе поиска обнаруживается, что задано ошибочное направление движения ленты, то оно меняется на противоположное, и нужная зона все равно будет найдена (если она есть на ленте, в противном случае наступит «авост»), но поиск займет больше времени.

Содержимое 33-го, 35-го и 36-го разрядов задает различные блокировки. Единица в 35-м разряде блокирует весь контроль — запись контрольной суммы во внешнюю память или ее вывод, а в случае чтения — сравнение контрольных сумм и его возможные последствия (останов машины и переход по второму адресу команды *Mb* после повторного пуска). Подсчет контрольной суммы никогда не блокируется. Предотвратить ее запись в ОЗУ можно, только задав в качестве адреса записи (третьего исполнительного адреса команды *Mb*) нулевой адрес.

Единица в 36-м разряде блокирует только занесение данных в МОЗУ при операции чтения, это позволяет осуществлять *фиктивное чтение* для контроля правильности выполненной перед этим операции записи во внешнюю память.

Наименее специфичен 33-й разряд условного числа. При операции чтения из внешней памяти единица в 33-м разряде блокирует останов машины по несовпадению контрольных сумм. При выводе на печать единица в 33-м разряде означает, что вывод (на узкую ленту) производится в режиме восьмеричной печати (см. § 1), единица в 30-м разряде (исключение из общего правила, запрещающего наличие двух единиц в разрядах 28 ÷ 32) означает, что выполняется печать текста на АЦПУ.

В сжатом виде роль разрядов условного числа отображена в следующей таблице.

№ разряда	Смысл «1» в разряде
36	Блокировка ОЗУ
35	Блокировка контроля
34	Обратное направление
33	Блокировка останова или восьмеричная печать
32	Перфорация
31	Печать
30	Разметка ленты или печать на АЦПУ
29	МЛ (обмен с лентой)
28	МБ (обмен с барабаном)
27	Запись

Операции вывода не случайно выполняются по тем же командам, что операции обмена. На машине М-20 вывод начинается с того, что выводимый массив записывается из ОЗУ на специальную секцию одного из магнитных барабанов (номер этого барабана определяется схемно и его не надо указывать в команде *Ма*). По окончании записи начинается собственно печать или перфорация данных из этой секции, называемой *буфером*. Во время печати или перфорации работа программы продолжается, если только не потребуется выполнить другую операцию вывода. Выполнение этой операции откладывается до завершения предыдущей. Такое совмещение работы устройств вывода и центральных устройств машины позволяет заметно ускорить работу машины. Современные вычислительные машины допускают еще более широкое и разнообразное совмещение во времени работы различных устройств.

Буфер размещается на барабане дополнительно к его 4096 основным ячейкам. Емкость буфера — 512 ячеек (45-разрядных). При выводе с незаблокированным контролем вслед за выводимым массивом в буфер записывается, а затем печатается или перфорируется его контрольная сумма. В отдельной ячейке буфера вслед за контрольной суммой или за концом массива (при выводе с блокировкой контроля) помещается признак конца вывода. Таким образом, за один прием можно вывести массив длиной не более 510 (или 511, если контроль заблокирован) ячеек. Признак конца вывода только управляет работой печатающего устройства или перфоратора и не имеет ничего общего с признаком конца массива на перфокартах. Если вывод происходит на перфорацию и с контролем, то контрольная сумма массива отперфорируется вместе с двумя маркерами и образует признак конца массива. При выводе на печать с контролем контрольная сумма печатается в том же виде, как содержимое любой другой ячейки. Такой вывод имеет смысл

в случае восьмеричной печати, позволяющей восстановить все разряды этой суммы. При выводе без контроля контрольная сумма не печатается и не перфорируется, следовательно, в выведенном массиве перфокарт отсутствует и признак конца.

Рассмотрим теперь несколько примеров организации вывода и обмена.

Пример 2. Пусть в ячейках $a + 1 \div a + n$ располагается массив чисел, переведенных в двоично-десятичное представление. Требуется отпечатать этот массив на узкой ленте.

Следует отдельно рассмотреть два случая: случай, когда известно $n \leq 511$, и случай произвольного n . В первом случае массив можно отпечатать за один прием. Программа печати состоит всего из двух команд:

```
0 Ма  УЧ  0  a+n
0 Мб  a+1  0   0
```

где УЧ — условное число, вид которого мы определим немного позже. Адреса $a + 1$ и $a + n$ задают начало и конец выводимого массива в ОЗУ. Нулевой второй адрес команды Ма означает, что массив перед началом печати записывается в буфер, начиная с его самой первой ячейки (имеющей адрес 0000). Иное расположение массива в буфере не имело бы смысла. Второй адрес команды Мб безразличен, так как при выводе сравнение контрольных сумм не проводится (подсчитанную сумму просто не с чем сравнивать) и условный переход по несовпадению сумм не производится. Сохранять подсчитанную при выводе контрольную сумму, по-видимому, также незачем, поэтому и третий адрес команды Мб равен нулю.

Перейдем к первому адресу команды Ма. Два младшие его разряда (25-й и 26-й) безразличны, так как в операции вывода на печать или на перфорацию буфер не считается частью какого-либо определенного барабана. В 27-й разряд помещаем нуль, так как единица в этом разряде имеет смысл лишь в операциях записи на барабан или ленту, а не в операциях вывода. Из следующих пяти разрядов лишь один 31-й разряд должен содержать единицу, так как вывод производится на узкую ленту. В 33-м разряде должен стоять нуль, потому что печатаются десятичные числа. Печать контрольной суммы не имеет смысла, и чтобы ее исключить, в 35-й разряд следует поместить единицу. В разряды 34-й (движение ленты в обратном направлении) и 36-й (блокировка обращения к ОЗУ), разумеется, следует вписать нули. Таким образом, первый адрес команды Ма должен иметь вид

010 001 000 000

или, в восьмеричной системе, 2100.

Обратимся теперь к случаю произвольного n . Массив следует разбить на части, содержащие не более 511 слов каждая, и отпе-

печатать каждую часть в отдельности. Алгоритм печати может быть таким:

P. Печать n чисел из ячеек $a + 1 \div a + n$.

P1. $i := 0$, перейти на P3.

P2. Отпечатать числа из ячеек $a + i - 510 \div a + i$.

P3. $i := i + 511$, если $i < n$, то перейти на P2.

P4. Отпечатать числа из ячеек $a + i - 510 \div a + n$.

Заманчиво было бы воспользоваться следующей программой:

Программа 4

P1	0	$I \geq A$	0	P3	0	$I := 0$; на P3
P2	1	Ma	2100	0	a	} Печать чисел из ячеек $a + I - 510_{10} \div a + I$
	4	Mб	$a - 776$	0	0	
P3	1	$I < A$	$n - 777$	P2	0777	$I := I + 511_{10}$; если $I < n$, то на P2
P4	0	Ma	2100	0	$a + n$	} Печать чисел из ячеек $a + I - 510_{10} \div a + n$
	4	Mб	$a - 776$	0	0	

Но, к сожалению, она будет работать правильно, если только $n \geq 511_{10}$. Для $n < 511_{10}$ адрес $n - 777_8$ в команде P3 должен был бы быть отрицательным, но это невозможно. Фактически он будет равен $n + 7001_8$ и сравнение содержимого индекса-регистра с этим значением приведет не к тем последствиям, которые мы ожидаем. Улучшенный вариант программы таков:

Программа 5

P1	0	$I \geq A$	0	P3	0777
P2	1	Ma	2100	0	$a - 777$
	4	Mб	$a - 1775$	0	0
P3	1	$I < A$	$+n$	P2	0777
P4	0	Ma	2100	0	$a + n$
	4	Mб	$a - 1775$	0	0

Здесь все значения I увеличены на 511_{10} по сравнению с предыдущим вариантом.

У п р а ж н е н и е 3. При каких значениях n будет правильно работать этот вариант программы?

П р и м е р 3. В § 5 гл. 1 упоминалась так называемая пусковая программа, предназначенная для организации ввода в машину произвольных программ. Составим такую пусковую программу для ввода в машину М-20 произвольной замкнутой программы (так называется программа, не обращающаяся ни к каким другим программам, в том числе и к стандартным библиотечным подпрограммам — см. гл. 15). Эта пусковая программа должна предварительно заполнить всю оперативную память, не занятую ею самой, словами, состоящими из единиц во всех разрядах, ввести программу с пер-

фокарт, записать содержимое ОЗУ на 1-й магнитный барабан, проверить правильность записи фиктивным считыванием и передать управление на начало введенной программы. Адрес a_n начала программы не может быть известен пусковой программе, поэтому предположим, что в начале колоды перфокарт, на которых пробита основная программа, находится перфокарта с командой

$$0 \text{ ПБ } 0 \text{ } a_n \text{ } 0 \quad (1)$$

за которой следуют программные массивы, каждый из которых снабжен адресным кодом. На последней перфокарте колоды должен быть пробит признак конца ввода с правильной контрольной суммой.

Пусть пусковая программа располагается в ОЗУ, начиная с ячейки 0001. Эта программа может быть, например, такой:

Программа 6						
0001	0	АИ	0	0015	0	Заполнение всех разрядов ОЗУ единицами, начиная с ячейки 0015
2	1	ПЧ	0014	0	0	
3	1	И < А	7777	0002	0001	
4	0	ВВ	0001	0004	0000	Ввод с контролем до совпадения сумм
5	0	Ма	0015	0	7777	
6	0	Мб	0014	0	0002	Запись на 1-й барабан с контролем
7	0	Ма	4411	0	7777	
0010	0	Мб	0014	0005	0003	Фиктивное чтение с 1-го барабана с контролем и блокировкой останова
1	0	$\neq a$	0002	0003	0	
2	0	ПО	0	0005	0	Переход на повторение записи при несовпадении сумм при записи и чтении
3	0	ПБ	0	0001	0	
4	7	77	7777	7777	7777	Переход на начало программы

Сама пусковая программа рассчитана на размещение в определенном месте памяти и на работу с программным массивом, также имеющим фиксированное расположение (в ячейках 0014 ÷ 7777). Поэтому она составлена сразу в истинных адресах в отступление от наших обычных правил. Работа программы ясна из покомандных примечаний, дополнительно можно отметить лишь следующее. Участок программы, расположенный в ячейках 0001 ÷ 0003, выполняется однократно, после чего эти три ячейки используются как рабочие. В ячейку 0001 вводится команда (1), через которую команда 0013 передает управление на начало программы. Предполагается, что вводимая программа размещается в ячейках с адресами, не меньшими 0014, однако когда она начнет работать, ячейки 0001 ÷ 0013 она может использовать в качестве рабочих ячеек.

В ячейки 0002 и 0003 пусковая программа помещает контрольные суммы, подсчитанные при записи на барабан и при фиктивном считывании. Затем эти суммы сравниваются (команда 0011) и при несовпадении управление вновь передается на запись. Есть ли необходимость в этом дополнительном контроле, если перед этим

при фиктивном чтении подсчитанная сумма уже совпала с суммой, подсчитанной и записанной на барабан во время записи массива? Оказывается, этот контроль вовсе не является избыточным. Возможны неисправности в канале записи на барабан, при которых во все ячейки барабана вместо слов записываемого массива и его контрольной суммы ничего не записывается (содержимое ячеек не меняется) или записываются нулевые слова. Кроме того, на пульте машины есть переключатели, с помощью которых можно отключить цепи записи (или записи и чтения) любого из барабанов. Если из-за неисправности машины или по невниманию оператора запись не состоялась, а при решении предыдущей задачи в те же ячейки барабана был записан совсем другой массив (например, при работе той же пусковой программы), то при фиктивном чтении контроль ничего не обнаружит. То же самое произойдет, если при записи на барабан поступали нулевые слова (контрольная сумма любого массива нулевых слов представляет собой также нулевое слово). В обоих случаях сравнение контрольных сумм, фактически подсчитанных во время обеих операций обмена, может обнаружить, что запись на барабан произошла неверно. Операция $\neq a$ в команде 0011 имеет перед операцией $\neq$ то преимущество, что оператор машины заметит остановку машины скорее, чем многократное циклическое повторение участка пусковой программы 0005 $\div$ 0012. В случае остановки оператор (предполагается, что он знает особенности работы пусковой программы) должен проверить положение переключателей первого магнитного барабана. Если запись или чтение были отключены, то оператор должен их включить и вновь пустить машину. Если и при включенном барабане происходит останов при выполнении команды 0011, значит, машина неисправна.

Наконец, может возникнуть вопрос, для чего перед вводом программы все ячейки ОЗУ, не занятые пусковой программой, заполняются единицами во всех разрядах, а не какими-нибудь другими словами. Этот прием очистки памяти перед вводом программы не связан с выполнением операций ввода, вывода или обмена. Он лишь способствует обнаружению ошибок в программе. Если во время работы программы управление будет передано по неверному адресу и очередная команда поступит из ячейки, не относящейся к программе и, следовательно, заполненной единицами, то машина остановится, так как операция с кодом $111\ 111_2 = 77_8$ — это останов машины. Слово, состоящее из одних единиц, изображает самое большое по абсолютной величине представимое в машине число. Если из-за ошибки в программе такое слово будет выбрано в качестве операнда арифметической операции, то с большой вероятностью произойдет «авост» при выполнении этой или одной из следующих операций. Кроме того, результат операции будет снабжен признаком (единицей в 45-м разряде), который будет переходить

ко всем числам, полученным с использованием этого результата. Так как в большинстве программ признаки чисел не используются, то появление признака у одного из чисел, полученных в итоге работы программы, обычно свидетельствует о какой-то ненормальности.

Пример 4. Организация контроля вычислений методом *двойного просчета*.

В ответственных программах, особенно если они требуют много времени для своего выполнения, приходится считаться с возможностью случайного *сбоя* (неверного выполнения одной из операций) даже в самых надежных устройствах машины. Кроме того, программа может содержать параметры, целесообразные значения которых можно установить только на основании результатов работы программы, причем метод подбора значений этих параметров заранее также неясен. В этих условиях целесообразно вести вычисления, периодически запоминая во внешней памяти состояние программы (предполагаем, что она занимает только оперативную память машины). После каждой записи состояния оперативной памяти (назовем его *контрольной точкой*) во внешнюю память вычисления до следующей контрольной точки проводятся дважды с восстановлением предыдущей контрольной точки перед вторым просчетом. Если результаты обоих просчетов совпадают, то можно зафиксировать очередную контрольную точку. Если результаты различны, то делается третий просчет и в качестве контрольной точки принимается его результат, если он совпал с одним из предыдущих. Если же все три результата разошлись, то выполнение программы следует прекратить, так как машина, по-видимому, неисправна.

Контролировать состояние оперативной памяти можно разными способами. Можно, в частности, применить тот же прием, который используется для контроля состояния участков внешней памяти — подсчитывать и сравнивать контрольные суммы всех слов, находящихся в оперативной памяти, кроме тех, которые относятся к самой программе контроля.

Если задаться целью контролировать только правильность вычислений, то можно фиксировать контрольные точки, записывая контролируемый участок оперативной памяти на один и тот же барабан. Однако для обеспечения возможности возобновить вычисления с любой из пройденных контрольных точек, изменив какие-либо параметры программы, следует сохранять контрольные точки в последовательных зонах одной из магнитных лент.

Прежде чем двигаться дальше, читателю рекомендуется самому составить алгоритм рассматриваемого метода контроля вычислений. На основании полученного опыта он сможет убедиться, что из общего алгоритма целесообразно выделить две самостоятельные части:

А. Продвижение от одной контрольной точки до другой.

А1. Возвратиться в основную программу и продолжить ее выполнение до следующей контрольной точки.

А2. Вычислить в ячейке s контрольную сумму всех слов участка оперативной памяти, занятого основной программой и ее рабочими ячейками.

В. Повторный просчет после k -й контрольной точки.

В1. Восстановить содержимое оперативной памяти считыванием из k -й зоны магнитной ленты.

В2. Выполнить алгоритм А.

Эти два алгоритма аналогичны описаниям процедур в АЛГОЛе, а шаг В2 представляет собой своего рода оператор процедуры. Теперь с помощью алгоритмов А и В запишем весь алгоритм контроля.

С. Контроль вычислений методом двойного просчета.

С1. $k := 1$ (подготовка к записи первой контрольной точки).

С2. Записать содержимое контролируемого участка ОЗУ в k -ю зону магнитной ленты (фиксация контрольной точки).

С3. Выполнить алгоритм А, $s_1 := s$ (первый просчет).

С4. Выполнить алгоритм В (второй просчет).

С5. Если $s = s_1$, то перейти на шаг С9.

С6. $s_2 := s$, выполнить алгоритм В (третий просчет).

С7. Если $s = s_1$ или $s = s_2$, то перейти на шаг С9.

С8. Выход (останов).

С9. $k := k + 1$, перейти на шаг С2.

Перейдем к детализации алгоритма (впрочем, можно считать, что алгоритмы В и А уже детализируют некоторые шаги алгоритма С). Для каждой контрольной точки на шаге А2 вычисляется (независимо от выполнения операций обмена) циклическая (контрольная) сумма контролируемого участка оперативной памяти. Целесообразно хранить в отдельной ячейке s_0 эту сумму для последней записанной на ленте контрольной точки. Это позволит ускорить выполнение шагов С2 и В1 и повысить их надежность. Шаг С2 после детализации примет вид:

С2. Фиксация контрольной точки.

С2.1. Записать содержимое контролируемого участка ОЗУ в k -ю зону ленты; контрольную сумму, полученную при записи, поместить в ячейку r .

С2.2. Если $r \neq s_0$, то перейти на шаг С2.1.

С2.3. Фиктивно считать k -ю зону ленты, полученную сумму поместить в ячейку r .

С2.4. Если $r \neq s_0$, то перейти на шаг С2.1.

Шаг В1 можно детализировать следующим образом:

В1. Восстановление предыдущей контрольной точки.

В1.1. Считать k -ю зону ленты в оперативную память, полученную сумму поместить в ячейку r .

В1.2. Если $r \neq s0$, то перейти на шаг В1.1.

В1.3. Вычислить в ячейке s циклическую сумму контролируемого участка ОЗУ.

В1.4. Если $s \neq s0$, перейти на шаг В1.1.

Подшаги В1.3 и В1.4 могут показаться лишними. Однако они позволяют убедиться, что все слова основной программы были не только верно считаны с ленты в арифметическое устройство (где подсчитывалась сумма r), но и верно переданы в ячейки ОЗУ.

Детализированный шаг С2 использует содержимое ячейки $s0$, поэтому шаг С1 должен быть уточнен следующим образом:

С1. Подготовка к записи первой контрольной точки.

С1.1. Вычислить в ячейке s циклическую сумму контролируемого участка ОЗУ.

С1.2. $s0 := s$; $k := 1$.

Следует также уточнить шаг С9:

С9. $s0 := s$, $k := k + 1$, перейти на шаг С2.

Теперь можно приступить к переводу этого алгоритма на машинный язык при следующих дополнительных предположениях. Контролируемый участок ОЗУ занимает ячейки с адресами от a до z . Основная программа по достижении очередной контрольной точки всегда передает управление программе контроля по одному и тому же адресу С1 с засылкой команды возврата в некоторую ячейку, адрес которой мы заменим буквой R . Значение индекс-регистра, с которым завершился очередной период работы основной программы, также входит в понятие «состояние основной программы». Будем запоминать это значение в ячейке i , которая (так же, как ячейка R) должна располагаться в пределах контролируемого участка ОЗУ.

Шаг С1 выполняется один раз за все время работы основной программы совместно с программой контроля. Это позволяет в дальнейшем использовать участок программы, реализующий подшаг С1.1, как подпрограмму, по которой будут выполняться аналогичные действия на шаге А2 и подшаге В1.3.

В итоге получаем следующую программу. В качестве примечаний к ее командам указаны лишь номера шагов или подшагов алгоритмов С, В и А и действия над содержимым индекс-регистра.

Программа 7

С1	4	АН	0	a	i	$i := H_{\text{осн}}; H := a$
	0	ПЧ	0	0	s	} С1.1 (А2, В1.3)
p	2	+Ц	s	0	s	
	1	$H < A$	z	p	1	
$R1$	0	ПЧ	s	0	$s0$	} С1.2
	0	АН	0	1	0	
С2	2	Ма	0024	0	z	} С2.1 ($H = k$)
	0	Мб	a	0	r	

	0	$\neq$	r	$s0$	0	} C2.2
	0	$\Pi 0$	0	C2	0	
	2	Ma	5420	0	z	} C2.3 ($H=k$)
	0	$M6$	a	C2	r	
	0	$\neq$	r	$s0$	0	} C2.4
	0	$\Pi 0$	0	C2	0	
	4	$KИ$	0	i	k	$k := H; H := H_{\text{очн}}$
C3	0	ΠB	$C3+1$	R	R1	A } C3
	0	$\Pi Ч$	s	0	$s1$	
C4	0	ΠB	C5	B	R2	B (C4)
C5	0	$\neq$	s	$s1$	0	} C5
	0	$\Pi 1$	s	C9	$s2$	
	0	ΠB	C7	B	R2	B } C6
C7	0	$\neq$	s	$s1$	0	} C7
	0	$\Pi 1$	0	C9	0	
	0	$\neq a$	s	$s2$	0	
	0	$\Pi 0$	s	B	$s2$	
C9	0	$\Pi Ч$	s	0	$s0$	} C9
	0	$KИ$	0	k	0	
	1	$H \geq A$	0	C2	1	
B	0	$KИ$	0	k	0	$H := k$
	2	Ma	1420	0	z	} B1.1 ($H=k$)
	0	$M6$	a	$B+1$	r	
	0	$\neq$	r	$s0$	0	} B1.2
	0	$\Pi 0$	0	$B+1$	0	
	0	AI	0	a	0	} B1.3
	0	ΠB	q	$C1+1$	$R1'$	
q	0	$\neq$	s	$s0$	0	} B1.4
	0	$\Pi 0$	0	$B+1$	0	
	0	$KИ$	0	i	0	} B2
	0	ΠB	R2	R	R1	

В этой программе предусмотрены два условных перехода по несовпадению контрольных сумм при чтении с ленты. На шаге C2.3 переход делается на повторную запись, так как несовпадение сумм может быть следствием неверной первоначальной записи на ленту. На шаге B1.1, который выполняется после того, как правильность записи была всесторонне проверена, переход делается на повторное чтение, так же как после проверок на шаге B1.2 и B1.4. Операции чтения с ленты (шаги C2.3 и B1.1) выполняются после того, как проводилась запись или чтение той же зоны ленты. Естественно, что поиск зоны следует производить при обратном направлении движения ленты. Напротив, операция записи на ленту (шаг C2.1) выполняется в нормальном случае после обращения к предыдущей зоне и поиск зоны производится при прямом направлении движения. Возможны, однако, случаи, когда шаг C2.1 выполняется после

шага С2.2, С2.3 или С2.4 (если контроль обнаруживает, что запись произведена неверно). В этих случаях также следовало бы искать зону при обратном направлении движения. Но так как направление движения сменится при поиске автоматически, то можно не удлинять программу и пойти на некоторое замедление ее работы в этих редких (при исправной машине) случаях. Команда С7 + 3 позволяет, пустив машину после останова, вновь уйти на повторный просчет.

У п р а ж н е н и я 4. Работу программы контроля можно несколько ускорить, если записывать контрольные точки не только на ленте, но и на барабане, и для восстановления оперативной памяти перед повторным просчетом производить чтение с барабана, а не с ленты. Кроме того, на ленте можно сохранять не каждую контрольную точку, а, скажем, одну из 10. Составить соответствующий вариант программы.

5. Составить программу разметки магнитной ленты при следующих условиях. На ленте должно быть 250_{10} зон по 1000_{10} ячеек (не считая ячейки для контрольной суммы) в каждой зоне. Зонам должны быть даны последовательные четные номера от 2 до 500_{10} . При разметке зона заполняется сначала нулевыми словами, а затем словами, состоящими из одних единиц. После каждой записи делается контрольное чтение. Если после двух попыток правильная запись не удастся, то на том же участке ленты размечается (без контроля) зона с нечетным номером, на единицу меньшим номера той зоны, которую разметить не удалось, и делается попытка разметить эту зону на следующем участке ленты. Так можно обойти участки ленты с дефектами магнитного слоя.

§ 5. Ввод и вывод чисел

Выше уже говорилось о том, что при вводе чисел в машину обычно приходится переводить их из двоично-десятичного представления, описанного в § 1, в двоичное (машинное) представление, а при выводе — осуществлять обратное преобразование. Алгоритмы перевода были разобраны в § 2 гл. 2. Машинная реализация этих алгоритмов основывается на операциях и приемах программирования, рассмотренных в гл. 12 и 13. В этом параграфе мы более детально разберем несколько алгоритмов и программ перевода. Они интересны не только с точки зрения решения этой конкретной задачи, но и как примеры подхода к программированию задач, чуть более сложных, чем те, которыми мы занимались в предыдущих разделах.

Рассмотрим сначала методы перевода чисел с фиксированной запятой. Забудем на время, как используются разряды ячейки для представления чисел в машине М-20. Будем интересоваться только содержимым младших 36 разрядов ячейки, которые обычно отводятся для записи абсолютной величины мантиссы числа. Из множества возможных способов зафиксировать запятую в этом двоичном числе остановимся пока на одном, когда запятая ставится непосредственно справа от младшего разряда. Это значит, что мы читаем содержимое этих разрядов, как целое число n из диапазона

Рассматриваемым двоичным разрядам (точнее, записанным в них цифрам) дадим обозначения (слева направо) $b_{35}, b_{34}, \dots, b_1, b_0$, а десятичным цифрам, представленным четверками двоичных разрядов, — обозначения $d_8, d_7, \dots, d_0$. При этом двоичная цифра b_i изображает число 2^i , а десятичная цифра d_j — число 10^j . Между введенными обозначениями существует следующая связь:

$$n = \sum_{i=0}^{35} b_i \cdot 2^i,$$

$$n' = \sum_{j=0}^8 d_j \cdot 10^j.$$

$$\begin{aligned} & d_8, \\ & d_8 \cdot 10 + d_7, \\ & \dots\dots\dots \\ & d_8 \cdot 10^8 + d_7 \cdot 10^7 + \dots + d_0 = n', \end{aligned}$$
$$l_i = d_{i-1} \cdot 2^{4(i-1)} + \dots + d_1 \cdot 2^4 + d_0.$$
$$k_i < 10^{9-i} < 16^{9-i} = 2^{4(9-i)}.$$

Для его записи в двоичной системе достаточно 4 ($9 - i$) двоичных разрядов. Можно использовать разряды, которые ранее были заняты цифрами $d_8, \dots, d_i$. Если так поступить, то содержимое ячейки,

в старших разрядах которой размещается число k_i , а в младших — число l_i , будет изображать целое число

$$m_i = k_i \cdot 2^{4i} + l_i.$$

При этом двоичная запись числа

$$m_8 = k_8 \cdot 2^{4 \cdot 8} + l_8 = d_8 \cdot 2^{32} + d_7 \cdot 2^{28} + \dots + d_0$$

совпадает с исходной двоично-десятичной записью, а число

$$m_0 = k_0 \cdot 2^0 + l_0 = n'$$

является окончательным результатом перевода.

Таким образом, перевод можно осуществить в виде цикла, при очередном повторении которого значение m_i преобразуется в m_{i-1} . Между числами m_i и m_{i-1} существует простое соотношение. Их разность равна

$$\begin{aligned} m_i - m_{i-1} &= k_i \cdot 2^{4i} + l_i - k_{i-1} \cdot 2^{4(i-1)} - l_{i-1} = \\ &= (16k_i - k_{i-1}) \cdot 2^{4(i-1)} + l_i - l_{i-1}. \end{aligned}$$

Легко проверить, что $k_{i-1} = 10k_i + d_{i-1}$ и $l_i = d_{i-1} \cdot 2^{4(i-1)} + l_{i-1}$, откуда

$$m_i - m_{i-1} = (6k_i - d_{i-1}) \cdot 2^{4(i-1)} + d_{i-1} \cdot 2^{4(i-1)} = 6k_i \cdot 2^{4(i-1)},$$

так что

$$m_{i-1} = m_i - 6k_i \cdot 2^{4(i-1)}. \quad (1)$$

Пусть в младших 36 разрядах ячейки n помещается число m_i , а в ячейках z_1 и z_2 — заготовки

$$\begin{array}{llllll} z_1 & 777 & 7777 & 7777 & 7777 & \\ z_2 & 100 & 3000 & 0000 & 0000 & 2^0 \cdot 6/16 \end{array}$$

Выполним команды

$$\begin{array}{llllll} 0 & \leftarrow SA & 100_8 + 4i & z_1 & r & \text{Единицы в старших } 4(9-i) \\ & & & & & \text{разрядах мантиссы} \\ 0 & \wedge & n & r & r & k_i \cdot 2^{4i} \\ 0 & , \times & z_2 & r & r & 6k_i \cdot 2^{4(i-1)} \\ 0 & - M & n & r & n & m_i - 6k_i \cdot 2^{4(i-1)} \end{array}$$

Из формулы (1) следует, что эта группа команд заменяет запись числа m_i в младших 36 разрядах ячейки n записью числа m_{i-1} . Если повторить эти команды в цикле для $i = 8, 7, \dots, 1$, то в разрядах мантиссы ячейки n получится изображение числа n' . От этого нестандартного изображения легко перейти к обычному представлению этого числа, если приформировать к нему код порядка 144₈, изображающий порядок 2³⁶, а затем нормализовать полученное машинное число.

Итак, следующая программа решает нашу задачу:

Программа 1

	0	AI	0	40	0	$I := 32_{10} \ (i := 8)$
p	4	$\leftarrow CA$	100	z_1	r	Единицы в разрядах k_i
	0	$\wedge$	n	r	r	$k_i \cdot 2^{4i}$
	0	$, \times$	z_2	r	r	$6k_i \cdot 2^{4(i-1)}$
	0	$-M$	n	r	n	m_{i-1}
	1	$I \geq A$	10	p	7774	$I := I - 4$; если $I \geq 4$, то на p
p_1	0	$+M$	z_3	n	n	$2^{36} \cdot (n \cdot 2^{-36})$
	0	$+$	n	0	n	Нормализация

Константы:

z_1	777	7777	7777	7777
z_2	100	3000	0000	0000
z_3	144	0000	0000	0000

Программу 1 можно усовершенствовать и упростить. Очень легко видоизменить ее так, чтобы она переводила в машинное представление также отрицательные числа и числа, снабженные признаком. Действительно, программа 1 не меняет содержимого старших девяти разрядов ячейки n . Поэтому, если в 45-й разряд этой ячейки перед началом работы программы поместить признак числа, в 44-й разряд — код знака числа, разряды с 43-го по 37-й заполнить нулями, а команду p_1 заменить командой

$$p_1 \quad 0 \quad \vee \quad n \quad z_3 \quad n$$

то переведенное число сохранит заданный признак и знак. Целое число, которое будет переводиться в двоичную систему по этой программе, должно быть записано на бланке в виде

$$\pm \pm + \quad 00 \quad d_8 d_7 d_6 \quad d_5 d_4 d_3 \quad d_2 d_1 d_0 \quad (2)$$

где первый знак «+» или «-» обозначает признак, а второй — знак числа.

Очень часто целые числа, вводимые в машину, невелики и содержат всего 2—3 значащие цифры. При этом многие из чисел $k_8, k_7, \dots$, вычисляемых в ходе перевода, равны нулю, и выполнять команды $p + 2$ и $p + 3$ не имеет смысла. Можно поместить перед ними команду условного перехода по $\omega = 1$ на заключительную команду цикла. Если переводимое число имеет лишь s значащих цифр $d_{s-1}, \dots, d_0$, а цифры $d_8, \dots, d_s$ равны нулю, то в 9- s случаях из 8 операций, $\times$ и $-M$ не будут выполняться, но при каждом из 8 повторений цикла придется выполнить добавочную операцию П1. Общее число операций, выполняемых при переводе одного числа,

изменится на $2s - 10$. В подавляющем большинстве случаев (при $s < 5$) добавление команды условного перехода сократит время работы программы.

Наконец, можно сэкономить константу z_3 , если команду p_1 написать в виде

$$p_1 \quad 0 + \text{Ц} \quad n \quad z_1 \quad n$$

а константу z_1 заменить на

$$z_1 \quad 1 \ 44 \ 7777 \ 7777 \ 7777$$

(изменение старших 9 разрядов этой константы не влияет на результат работы команд $p \div p + 1$).

С этими переделками программа перевода целых чисел примет вид:

Программа 2

	0	АН	0	40	0	$H := 32_{10}$
p	4	$\leftarrow MA$	100	z_1	r	
	0	$\wedge$	n	r	r	$k_i \cdot 2^i$ (если $k_i = 0$, то $\omega := 1$)
	0	П1	0	p_1	0	
	0	$, \times$	z_2	r	r	$6k_i \cdot 2^i (i-1)$
	0	$-M$	n	r	n	m_{i-1}
p_i	1	$H \geq A$	10	p	7774	$H := H - 4$; если $H \geq 4$, то на p
	0	$+ \text{Ц}$	n	z_1	n	Приписывание порядка
	0	$+$	n	0	n	Нормализация

при константах

$$\begin{aligned} z_1 & 1 \ 44 \ 7777 \ 7777 \ 7777 \\ z_2 & 1 \ 00 \ 3000 \ 0000 \ 0000 \end{aligned}$$

У п р а ж н е н и я. 1. Составить программу перевода целого числа n из машинного представления в двоично-десятичное представление (2).

2. Может показаться, что быстрее перевести целое число из двоично-десятичного представления в машинное, чем это делает программа 2, невозможно. На самом деле это не так. Показать, что программа

Программа 3

	0	АН	0	0	0
p	2	$\wedge$	n	z	r
	2	$, \times$	r	c	r
	0	$-M$	n	r	n
	1	$H < A$	3	p	1
	0	$\vee$	n	z_1	n
	0	$+$	n	0	n

использующая константы

z	0 00	0360	7417	0360	Группы по 4 единицы и 4 нуля
$z+1$	0 00	0377	0017	7400	8 единиц и 8 нулей
$z+2$	0 00	0377	7760	0000	16 единиц и 16 нулей
$z+3$	0 00	7400	0000	0000	4 единицы и 32 нуля
z_1	1 44	0000	0000	0000	Порядок $+36_{10}$
c	1 00	3000	0000	0000	$(16-10)/16$
$c+1$	1 00	4700	0000	0000	$(16^2-10^2)/16^2$
$c+2$	1 00	6617	0000	0000	$(16^4-10^4)/16^4$
$c+3$	1 00	7640	5037	0000	$(16^8-10^8)/16^8$

выполняет в точности ту же работу, что программа 2, за 19 операций, тогда как программа 2 требует от 35 до 51 операции. Однако более быстрая программа занимает, как это часто случается, несколько больше места.

Перейдем к другому случаю перевода чисел с фиксированной запятой. Пусть как в двоичном, так и в десятичном представлении запятая фиксируется перед старшим разрядом, т. е. переводимые числа являются правильными дробями. Перевод из десятичного представления в двоичное осложнений не вызывает. Можно перенести (мысленно) в переводимом десятичном числе запятую на 9 разрядов вправо, перевести полученное целое число в двоичную систему так, как это делалось в программе 2 или 3, а затем разделить полученный результат на 10^9 .

При обратном переводе правильной двоичной дроби x в десятичное представление надо позаботиться о сохранении точности. Введем обозначения:

$$\begin{aligned} x_0 &= x + 0,5 \cdot 10^{-9}, \\ v_i &= \{10^i \cdot x_0\}, \quad i = 0, 1, \dots, 8, \\ d_i &= [10v_{i-1}], \quad i = 1, 2, \dots, 9, \\ x_i &= d_1 \cdot 2^{-4} + \dots + d_i \cdot 2^{-4i} + v_i \cdot 2^{-4i}, \quad i = 1, \dots, 8, \\ x_9 &= d_1 \cdot 2^{-4} + \dots + d_9 \cdot 2^{-36}. \end{aligned}$$

Будем считать, что $x_0 < 1$. В противном случае число x не может быть записано с хорошей точностью в виде правильной 9-разрядной десятичной дроби. Во введенных обозначениях d_i — это i -я цифра после запятой записи числа x в десятичной системе. Число x_9 имеет двоичную запись, совпадающую с округленной двоично-десятичной записью числа x , иначе говоря, его можно считать результатом перевода. Легко убедиться в справедливости соотношения

$$x_{i+1} = x_i + \frac{3}{8} v_i \cdot 2^{-4i}, \quad i = 0, 1, \dots, 8.$$

В рассматриваемом сейчас представлении чисел (правильных дробей) с запятой, фиксированной перед старшим разрядом, число $v_i \cdot 2^{-4i}$ изображается младшими 4 ($9-i$) разрядами ячейки. Все

сказанное обосновывает правильность следующей программы перевода правильных дробей из двоичного в двоично-десятичное представление:

Программа 4

	0	, + a	x	z ₁	x	x ₀ := x + 0,5 · 10 ⁻⁹
	0	ПЧ	z ₂	0	r ₁	i := 0
p	0	Λ	x	r ₁	r	v _i · 2 ⁻⁴ⁱ
	0	, × a	r	z ₃	r	3/8 · v _i · 2 ⁻⁴ⁱ
	0	— M	x	r	x	x _{i+1}
	0	← МА	74	r ₁	r ₁	{ i := i + 1; если i ≤ 8, то перейти на p
	0	ПО	0	p	0	
	0	Λ	x	z ₂	x	Очистка разрядов порядка

Константы:

z ₁	1 00	0000	0000	0043	2 ⁰ · (0,5 · 10 ⁻⁹)
z ₂	6 00	7777	7777	7777	
z ₃	1 00	3000	0000	0000	3/8

Программа 4 настолько похожа на программу 2, что она не требует дальнейших пояснений.

У п р а ж н е н и е 3. Основываясь на программе 4, составить программу перевода целых чисел из двоичной системы в десятичную, работающую быстрее, чем программа, приведенная в решении упражнения 1.

Перейдем к переводу чисел с плавающей запятой. Будем предполагать, что число задано в нормализованном виде (в двоичном или в двоично-десятичном представлении, описанном в § 1 гл. 13) и результат перевода также должен быть нормализован. Уже на примере чисел с фиксированной запятой мы видели, что знак и признак числа доставляют мало забот составителю программ перевода — они без изменений переходят из одного представления в другое. Напротив, порядок числа заставляет много повозиться. Не существует простого соотношения между двоичным и десятичным порядком. Кроме того, десятичный порядок записывается в прямом коде, а двоичный — в дополнительном. Знак порядка кодируется по-разному в этих двух представлениях. Наконец, абсолютная величина десятичного порядка сама должна быть записана в двоично-десятичном представлении. Следует учесть и то, что при переводе из двоичного представления в десятичное порядок результата определяется только после округления мантиссы.

С перевода из двоичного в десятичное представление мы и начнем. Алгоритм перевода (недетализированный) может быть таким.

Д. *Отыскание двоично-десятичного представления у двоичного (машинного) числа x.*

Д1. *Определение признака π, и знака s_y результата.*

Д2. *Определение десятичного порядка p_y и мантиссы m_y (в двоичном представлении).*

Д3. *Перевод мантиссы в двоично-десятичное представление.*

Д4. *Перевод порядка в двоично-десятичное представление.*

Займемся детализацией шагов этого алгоритма. На шаге Д1 целесообразно проверить, не равно ли нулю переводимое число x (довольно вероятный случай). Если $x = 0$, то можно положить $y = 0$ и выйти из алгоритма Д. На этом же шаге целесообразно выполнить некоторые подготовительные операции для последующих шагов. Одна из этих операций — это присваивание $x := |x|$.

Задача шага Д2 — представить число $x > 0$ в виде $x = 10^{p_y} \cdot x'$, где $0,1 \leq x' < 1$, а p_y — целое число. Для этого можно воспользоваться алгоритмом:

Д2. *Определение p_y и x' .*

Д2.1. $p_y := 0$, $x' := x$.

Д2.2. *Если $x' \geq 1$, то перейти на Д2.4.*

Д2.3. $x' := x' \times 10$, $p_y := p_y - 1$, *перейти на Д2.2.*

Д2.4. $x' := x'/10$, $p_y := p_y + 1$.

Д2.5. *Если $x' \geq 1$, то перейти на Д2.4.*

Упражнение 4. Нельзя ли упростить алгоритм Д2, записав шаги Д2.2 и Д2.3 в виде:

Д2.2. $x' := x' \times 10$, $p_y := p_y - 1$.

Д2.3. *Если $x' < 1$, то перейти на Д2.2.*

(при этом выбрасывается одна операция безусловного перехода).

Если учесть, что мы рассматриваем представление чисел лишь с конечной точностью, то на шагах Д2.2 и Д2.5 следовало бы сравнивать значение x' не с 1, а с числом $1 - 0,5 \cdot 10^{-9}$ — наибольшим числом, допускающим хорошее приближение в виде правильной девятеразрядной дроби. Но при машинной реализации такое сравнение потребовало бы большего числа операций (почему?), так что лучше случай $1 - 0,5 \cdot 10^{-9} < x' < 1$ обнаруживать на шаге Д3.

В основу шага Д3 можно было бы положить алгоритм, реализованный в программе 4, но для разнообразия мы используем другой, столь же быстрый алгоритм. Его детали ясны из приведенной ниже программы 5. Заметим только, что при выполнении команд $q_3 - 4 \div q_3 + 1$ мы не интересуемся значением старших девяти разрядов ячейки x . Если в результате выполнения команд $q_3 - 3$ и $q_3 - 2$ обнаруживается, что $x' > 1 - 0,5 \cdot 10^{-9}$, то цикл $q_3 - 1 \div q_3 + 4$ не выполняется, так как результат его заранее известен. Соответствующее этому случаю значение $m_y = 0,1_{10}$ заготавливается командой $q_0 + 3$ на шаге Д1. В этом случае необходимо также предусмотреть увеличение p_y на 1.

Детальный алгоритм шага Д4 основан на следующей зависимости между десятичным порядком p_y и кодом порядка k_y (целым числом,

записанным в разрядах $43 \div 47$ ячейки y):

$$k_y = \begin{cases} p_y & \text{при } 0 \leq p_y < 10, \\ p_y + 6 & \text{при } 10 \leq p_y \leq 19, \\ 64 - p_y & \text{при } -10 < p_y < 0, \\ 64 - p_y + 6 & \text{при } -19 \leq p_y \leq -10 \end{cases}$$

или

$$k_y = 32 + (p_y - 32) \cdot s + t,$$

где

$$s = \begin{cases} 1 & \text{при } p_y \geq 0, \\ -1 & \text{при } p_y < 0, \end{cases} \quad t = \begin{cases} 0 & \text{при } |p_y| < 10, \\ 6 & \text{при } |p_y| \geq 10. \end{cases}$$

Реализация вычисления k_y в зависимости от p_y в программе 5 достаточно ясна. Значение p_y хранится на индекс-регистре в дополнительном коде. Отрицательным значениям p_y соответствуют большие значения I , выделяемые сравнением I с числом 4000_8 . Заслуживает внимания команда $q_5 + 1$, в результате выполнения которой код операции $06 (+AP)$ в команде q_6 заменяется кодом $46 (-AP)$ при $p_y < 0$. После сравнения p_y с нулем в индекс-регистр помещается значение $I = p_y + 9 \pmod{2^{12}}$. Благодаря этому одной командой удастся проверить соблюдение неравенства $|p_y| < 10$.

Программа 5

q_0	0	$\wedge$	x	z_1	y	$\pi_y := \pi_x, s_y := s_x, k_y := 0,$ $m_y := 0$
	0	$\neq$	x	y	x	$\pi_x := 0, s_x := 0 \ (x := x)$
	0	$\Pi 1$	z_2	q_7	q_6	$s := 1$; если $x = 0$, то выход
	0	$\vee$	y	z_3	y	$k_y := 32, m_y := 0, 1_{10}$
	0	AI	0	0	0	$I := 0$
	0	$+PP$	z_4	x	0	Если $x \geq 1$, то
	0	$\Pi 1$	0	q_2	0	перейти на q_2
q_1	0	$\times$	x	$\langle 10 \rangle$	x	$x := x \times 10$
	1	$0I \geq A$	7756	q_1	7777	$I := I - 1$; если $I \geq -19 \wedge x < 1$, то на q_1
q_2	0	$\times$	x	$\langle 1/10 \rangle$	x	$x := x/10$
	1	$1I \geq A$	0	q_2	1	$I := I + 1$; если $x \geq 1$, то на q_2
	0	$\leftarrow MP$	x	x	x	Денормализация x
	0	$+M$	z_4	x	x	$x := x + 0,5 \cdot 10^{-9}$
	1	$1I \geq A$	0	q_5	1	$I := I + 1$; если $x \geq 1$, то на q_5
	4	AI	7777	1	q_4	$p_y := I - 1$; $I := 1$
q_3	0	$\times a$	x	z_6	r	$[10x]$ в младших разрядах
	0	$\times b$	0	0	x	$x := \{10x\}$
	0	$\leftarrow MP$	$\langle 10 \rangle$	y	y	Сдвиг m_y влево на 4 разряда
	0	$+M$	y	r	y	Приписывание очередной цифры
	1	$I < A$	11	q_3	1	$I := I + 1$; если $I \leq 9$, то на q_3

q_4	0	АН	0	p_y	0	$I := p_y$
q_5	1	$I < A$	4000	q_6	11	$I := I + 9$; если $p_y > 0$, то на q_6
	0	+АП	140	q_6	q_6	$s := -1$
q_6	4	$\pm$ АП	27	y	y	$k_y := k_y + (p_y - 32) \cdot s$ ($I = p_y + 9$)
	0	$I < A$	23	q_7	0	Если $I < 19_{10}$ ($ p_y < 10$), то выход
	0	+АП	106	y	y	$k_y := k_y + 6$
q_7						Выход

Программа использует следующие константы:

z_1	6	00	0000	0000	0000	
z_2	4	+АП	27	y	y	
z_3	0	40	0400	0000	0000	
z_4	1	00	0000	0000	0043	$2^0 \cdot (0,5 \cdot 10^{-9})$
z_5	1	00	0000	0000	0012	$2^0 \cdot (10 \cdot 2^{-30})$

В качестве следующего примера приведем программу перевода чисел из двоично-десятичной системы в двоичную (программа 6), используемую на многих машинах типа М-20 [31, 32]. На рис. 14.5.1 изображена блок-схема программы 6.

Программа 6

p	0	АН	0	0	0	$I := 0$
	0	ПЧ	0	0	y	$y := 0$
p_1	0	$\wedge$	x	z_1	r	Выделение очередной десятичной цифры из числа x ($r := \sigma_q \cdot 2^{-64} \cdot \alpha_i / 16$)
	0	,+	y	r	y	Формирование числа
	0	$\times$	y	$\langle 10 \rangle$	y	$2^{-68} \sigma_q (\alpha_1 \cdot 10^i + \dots + \alpha_i \cdot 10)$
	0	$\leftarrow$ МП	$\langle 10 \rangle$	x	x	
	1	$I < A$	10	p_1	1	$i := i + 1$; если $i \leq 8$, то на p_1 ($I = i - 1$)
	0	$\times$	y	c	y	$y := y \times 2^{68} \cdot 10^{-10}$
	0	$\wedge$	x	z_2	0	Выделение знака порядка; если $s_p = 0$, то $\omega := 1$
	0	П1	$\langle 10 \rangle$	p_2	r	$r := 10$; на p_2 при $\omega = 1$
	0	/	$\langle 1 \rangle$	$\langle 10 \rangle$	r	$r := 1/10$
p_2	0	$\leftarrow$ СА	56	x	x	Сдвиг $\beta_2 \beta_1$ в старшие разряды II адреса
	0	КН	0	x	0	$I := (16\beta_2 + \beta_1) \cdot 100_8$
	1	$I < A$	1777	p_4	0	Если $\beta_2 = 0$, то на p_4
	2	АН	0	7100	0	$I := (16\beta_2 + \beta_1) \cdot 100_8 - 7 \cdot 100_8 =$ $= (9 + \beta_1) \cdot 100_8$
p_3	0	$\times$	y	r	y	$y := yr$
p_4	1	$I \geq A$	100	p_3	7700	$C := I$; $I := I - 100_8$; если $C \geq 100_8$, то на p_3

где

z_1	6	00	7400	0000	0000
z_2	1	00	0000	0000	0000

Перевод мантиссы осуществляется командами $p_1 \div p_1 + 4$. Первая из этих команд выделяет старшую цифру мантиссы, знак мантиссы и признак числа. При первом выполнении в ячейке r

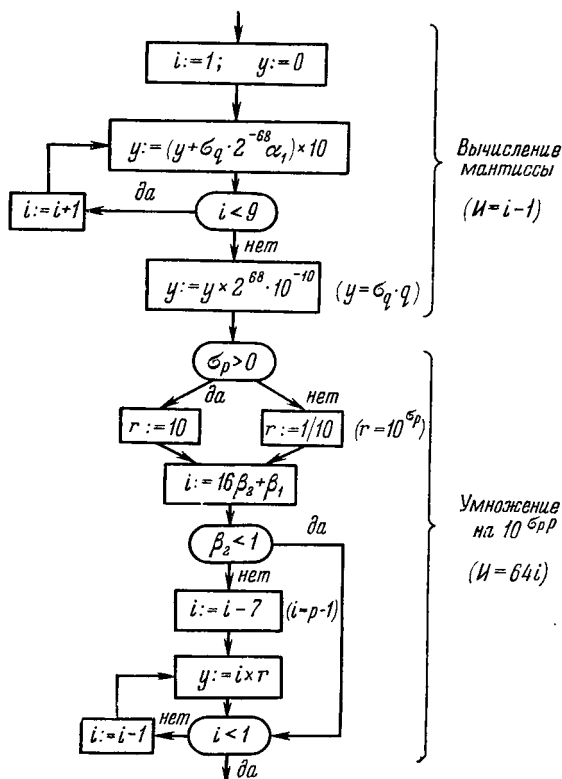


Рис. 14.5.1. Блок-схема программы 14.5.6.

окажется следующий код (если в разряде признака записан 0):

$r \ 0s_00 \ 00 \ \alpha_1 00000000$

Этот код используется в качестве операнта арифметической операции. Прочтем его как число, записанное в двоичной системе. Код порядка 000 соответствует двоичному порядку -64 . Число α_1 записано в двоичной системе в четырех старших разрядах мантиссы полученного числа, т. е. мантисса равна $\alpha_1/16$. Весь код в ячейке r изображает число $\sigma_q \cdot 2^{-64} \alpha_1 \cdot 2^{-4} = \sigma_q \cdot 2^{-68} \alpha_1$ ($\sigma_q = 1 - 2s_q$ — единица со знаком, совпадающим со знаком переводимого числа.)

Следующая команда прибавляет полученное число к результату, накопленному в ячейке y . В этой команде для ускорения вычислений используется операция сложения с блокировкой нормализации. Полученный результат умножаем на десять (команда $p_1 + 2$). В ячейке y получаем число $\sigma_q \cdot 2^{-68} (\alpha_1 \cdot 10^i + \dots + \alpha_i \cdot 10)$. При выполнении следующей операции ($p_1 + 3$) все цифры мантиссы сдвигаются на четыре разряда влево. Обработанная цифра сдвигается за пределы 36-разрядной сетки, на ее место попадает следующая цифра мантиссы. Так обрабатываются все цифры мантиссы (окончание обработки проверяется командой $p_1 + 4$).

После окончания этого цикла в ячейке y получим в двоичной системе число $\sigma_q \cdot 2^{-68} (\alpha_1 \cdot 10^9 + \dots + \alpha_9 \cdot 10)$. Чтобы получить мантиссу с нулевым порядком, это число умножаем на константу $2^{88} \cdot 10^{-10}$ (константа хранится в ячейке c) — это делает команда $p_1 + 5$.

Полученную мантиссу нужно умножить на $10^{(1-2\beta_p)(10\beta_2+\beta_1)}$, т. е. в зависимости от знака порядка умножить $(10\beta_2 + \beta_1)$ раз на 10 или $1/10$. Команды $p_2 - 3 \div p_2 - 1$ помещают в ячейку r число 10^{σ_p} , где σ_p — единица со знаком порядка переводимого числа.

В результате выполнения команды p_2 код в ячейке x сдвинется вправо на $22_8 = 18_{10}$ разрядов, т. е. цифры $\beta_2\beta_1$ окажутся в старших шести разрядах второго адреса, а значения признака числа и знаков мантиссы и порядка — в трех младших разрядах первого адреса. Операция *КИ* (команда $p_2 + 1$) зашлет в индекс-регистр содержимое второго адреса ячейки x ; в двоичной системе этот код обозначает число $(16\beta_2 + \beta_1) \cdot 100_8$. Наибольшее представимое в машине М-20 число имеет десятичный порядок, равный 19, т. е. цифра β_2 может быть равна 0 или 1. Если $\beta_2 = 1$, наименьшее значение I равно $010\,000\,000\,000_2 = 2000_8$; если $\beta_2 = 0$, наибольшее возможное значение I (при $\beta_1 = 9$): $001001000000_2 = 1100_8$. Командой $p_2 + 2$ мы и хотим различить эти два случая. Для этого достаточно сравнить значение I с любым числом, меньшим, чем 2000_8 , но большим, чем 1100_8 . В программе в качестве такого числа выбрано число 1777_8 . Если $\beta_2 = 0$ ($I < 1777_8$), то управление будет передано команде p_4 , если $\beta_2 = 1$, — команде $p_2 + 3$. Цикл, состоящий из команд p_3 и p_4 , осуществляет p -кратное умножение y на 10^{σ_p} , иначе говоря, умножение на $10^{\sigma_{pp}}$. Когда мы приходим на команду p_4 , значение I должно быть равно $(p \text{ минус число выполненных умножений на } 10^{\sigma_p}) \times 100_8$.

Если сравнение $p_2 + 2$ пропустило на выполнение команды $p_2 + 3$, то одно умножение на 10^{σ_p} заведомо выполнится до прихода на команду p_4 . Поэтому команда $p_2 + 3$ прибавляет к содержимому индекс-регистра, равному $(16 + \beta_1) \cdot 100_8 = (p + 6) \cdot 100_8$, число $(-7) \cdot 100_8$ (в дополнительном коде по модулю 2^{12}), и содержимое индекс-регистра оказывается равным $(p - 1) \cdot 100_8$.

Глава 15. ПОДПРОГРАММЫ

§ 1. Подпрограммы и обращения к ним

Подпрограмма — это часть программы, описывающая более или менее самостоятельный этап вычислительного процесса. Самостоятельность этапа означает, что потребность выполнить этот этап может возникнуть в различных местах данной программы или даже в программах решения других задач. Типичные примеры таких этапов — вычисление элементарных функций, вывод результатов вычислений, решение уравнений и т. п. Многие из примеров программ, составлявшихся в предыдущих разделах, могли бы служить подпрограммами в программах решения реальных задач.

Самостоятельность подразумевает разнообразие условий применения подпрограммы. Пусть, например, в программе возникла потребность решить систему линейных алгебраических уравнений. Если соответствующий участок программы составлен в расчете лишь на определенный порядок системы, на одно определенное расположение в памяти массива коэффициентов и свободных членов системы и массива ячеек, в которых должно быть получено решение, то такой участок еще не заслуживает названия подпрограммы. Если же он позволяет решить систему любого порядка (в определенных пределах, конечно), при любом расположении массивов коэффициентов и решений, то мы вправе говорить, что этот участок образует подпрограмму решения систем линейных алгебраических уравнений.

Программу, по отношению к которой некоторая совокупность команд является подпрограммой, будем называть *основной программой*. Основная программа определяет конкретные условия, в которых должна выполняться подпрограмма, и объекты, с которыми она должна работать. Величины, характеризующие эти конкретные условия и объекты, называются *параметрами обращения* к подпрограмме. В рассмотренном примере параметры обращения — это порядок системы (определяющий также размер расширенной матрицы системы и массива решения) и базовые адреса матрицы и решения системы. Встречаются две основные разновидности параметров. Числовые параметры — это просто значения некоторых исходных величин, с которыми имеет дело программа, например, значение аргумента некоторой элементарной функции.

В немного более общем случае параметр такого типа может быть и не числом, а произвольным кодом, занимающим одну ячейку памяти, например, двоично-кодированной записью десятичного числа, подлежащего переводу в двоичную систему, заготовкой команды, логическим вектором не слишком большого размера и т. п. Подобные параметры (включая числовые) будем называть *параметрами-значениями*. В ряде случаев полезен иной тип параметра — *параметр-адрес*. Это может быть, например, начальный или конечный адрес массива ячеек, или (в случае алгольских массивов) адрес элемента массива с нулевыми значениями всех индексов. В подпрограмме вычисления определенного интеграла могут оказаться необходимыми параметры-адреса начала и конца программы вычисления подинтегральной функции.

В некоторых случаях удобны параметры промежуточного типа, представляющие собой числовое значение, записанное в виде адреса. В таком виде, например, естественно задать порядок системы линейных алгебраических уравнений в упоминавшейся выше подпрограмме. Хотя этот параметр и является обычным целым числом, но используется он, как можно усмотреть из ряда разобранных выше примеров, лишь для формирования адресов. Связано это с особенностями конкретной машины и ее системы команд, однако задание параметра, о котором идет речь, в виде адреса удобно не только для машины М-20 (вернее, для подпрограмм, составленных для этой машины), но и для большинства других машин. Конструкция машины может предписывать и еще более специфическую форму задания параметров. На машине М-20, например, числовой параметр, определяющий количество разрядов сдвига, целесообразно задать в разрядах кода порядка числа. Мы ограничимся лишь этим упоминанием о подобных нетипичных способах задания параметров и в дальнейшем будем рассматривать лишь два способа задания параметров подпрограмм — в виде содержимого целой ячейки и в виде одного из адресов кода команды, представленной содержимым ячейки. Этого будет достаточно, чтобы познакомиться со всеми важнейшими приемами задания и использования параметров в подпрограммах.

Чтобы сообщить подпрограмме информацию, заключенную в параметрах обращения, и передать ей управление, нужно осуществить так называемое *обращение* к подпрограмме. Как правило, для этого основная программа должна содержать группу *команд обращения*, хотя знакомство с методами обращения к подпрограммам мы начнем с метода, в котором таких команд нет.

Ясно, что понятие подпрограммы в машинном коде тесно связано с понятием процедуры в немашинном языке программирования, например, в АЛГОЛе. Поэтому все, что будет говориться о составлении подпрограмм, является описанием приемов программирования процедур.

§ 2. Программирование по образцу

Если некоторая совокупность команд, которая по своему характеру может быть использована как подпрограмма (т. е. если она в достаточной мере самостоятельна и закончена), написана в символических обозначениях, то такую совокупность уже можно рассматривать как готовую подпрограмму. Для обращения к ней нужно лишь заменить символические обозначения, фигурирующие в подпрограмме, истинными (числовыми) адресами, соответствующими конкретным условиям обращения, и включить полученный текст в основную программу.

Иллюстрируем этот прием примером, который будет использован и в некоторых последующих параграфах. Пусть требуется вычислить сумму

$$y = \sum_{i=1}^n c_i f(x, i). \quad (1)$$

Хотя этот пример выглядит несколько надуманным (вероятно, ни в одной библиотеке стандартных подпрограмм нет подпрограммы решения именно такой задачи), но он хорош своей простотой при достаточном разнообразии форм параметров. Здесь есть строго числовой параметр x , числовой параметр n , одновременно определяющий размер массива c , сам массив c , параметр-функция f . Наконец, переменную y , значение которой требуется вычислить, тоже можно считать параметром.

Начнем с описания этой задачи в виде процедуры на АЛГОЛе. Это можно сделать, например, так:

```

procedure  $P(x, n, c, f, y)$ ; value  $x, n$ ;
real  $x, y$ ; integer  $n$ ; array  $c$ ; real procedure  $f$ ;
begin integer  $i$ ;
     $y := 0$ ;
    for  $i := 1$  step 1 until  $n$  do
         $y := y + c[i] \times f(x, i)$ 
end

```

(2)

Возможны и другие описания, например, в виде процедуры-функции, но мы остановимся на этом варианте.

Напишем машинную программу для операторной части тела процедуры P :

Программа 1

$p_{\text{ц}}$	0	ПЧ	0	0	y	$y := 0$
	0	ПЧ	$\langle 1 \rangle$	0	i	$i := 1$
	4	АН	0	1	$p_{\text{н}} + 10$	Запоминание $H_{\text{осп}}$, $H := 1$
	0	ПВ	$p_{\text{н}} + 4$	$f_{\text{ц}}$	$f_{\text{к}}$	Переход на f с возвратом

$p_n + 4$	4	$\times$	c	f	r	$c[i] \times f(x, i) \ (H=i)$
	0	$+$	y	r	y	$y := y + c[i] \times f(x, i)$
	0	$+$	i	$\langle 1 \rangle$	i	$i := i + 1;$
	1	$H < A$	$+n$	f_n	1	если $i \leq n$, то на f_n
$p_n + 10_8$	0	АИ	0	$\sim$	0	Восстановление $H_{осн}$

Здесь, как обычно, y — адрес ячейки, в которую требуется поместить значение суммы, f_n и f_k — адреса входа и выхода участка программы, вычисляющего в ячейке f значение функции $f(x, i)$. Предполагается, что этот участок берет аргументы для вычисления f из ячеек с адресами x и i и не портит содержимого этих ячеек. Существенной частью программы 1 являются команды $p_n + 2$ и $p_n + 10$, которые запоминают, а потом восстанавливают значение индекса-регистра основной программы.

Когда в какой-нибудь программе возникнет потребность вычислить сумму вида (1), например, сумму

$$u = \sum_{k=1}^{12} a_k \varphi(z, k), \quad (3)$$

то можно составить требуемый участок программы, взяв за образец программу 1. Для этого следует заменить в ней символические адреса $p_n, y, \dots$ числовыми адресами, соответствующими принятому в основной программе распределению памяти. Нужно также предусмотреть, чтобы к началу работы этого участка программы в ячейку, отведенную под переменную z , было заслано требуемое значение.

Пусть в основной программе для переменных u, z и k выделены ячейки с адресами 5013, 5021 и 5007, а массив $a_1, \dots, a_{12}$ размещен в ячейках $6731 \div 6744$ (так что базовый адрес a равен 6730 и $n = 12_{10} = 14_{18}$). Пусть адрес первой команды участка программы, вычисляющего значение функции φ , равен 2201, а адрес ячейки выхода из этого участка равен 2354. Этот участок должен быть составлен, разумеется, в согласии с принятым распределением памяти, т. е. значения аргументов z и k функции φ он должен брать из ячеек 5021 и 5007. Предположим, что значение функции φ он помещает в ячейку 5110. Предположим также, что для участка программы, составляемого по образцу, выделено место в памяти, начиная с ячейки 3663, а в качестве рабочей ячейки r этого участка можно взять ячейку с адресом 4522. Пусть, наконец, число 1 хранится в ячейке с адресом 7761.

При всех этих допущениях, характеризующих распределение памяти в основной программе, мы должны принять следующее распределение памяти при программировании по образцу:

p_n	3663	f_k	2354
y	5013	c	6730
$\langle 1 \rangle$	7761	f	5110
i	5007	r	4522
f_n	2201	$+n$	0014

Это распределение можно назвать *локальным*, так как оно действует только во время составления по образцу одного участка программы. При таком распределении участок, составленный по образцу, будет иметь вид:

Программа 2

3663	000	0000	0000	5013	$u := 0$
4	000	7761	0000	5007	$k := 1$
5	452	0000	0001	3673	$H := 1$, запоминание $H_{осн}$
6	016	3667	2201	2354	Переход на Φ_n с возвратом
7	405	6730	5110	4522	$a_k \times \varphi(z, k) (H = k)$
3670	001	5013	4522	5013	$u := u + a_k \varphi(z, k)$
1	001	5007	7761	5007	$\left. \begin{array}{l} k := k + 1; \text{ если } k \leq 12_{10}, \\ \text{то на } \Phi_n \end{array} \right\}$
2	112	0014	2201	0001	
3673	052	0000	0000	0000	Восстановление $H_{осн}$

Кроме того, в основной программе этому участку должна предшествовать команда, помещающая в ячейку 5021 надлежащее значение переменной z .

Если в той же или в другой программе потребуется выполнить вычисления по другой формуле, аналогичной формуле (1), то изменится лишь локальное распределение памяти, а программа 1 по-прежнему сможет служить образцом. Участок же программы, составленный по этому образцу, внешне будет значительно отличаться от программы 2. Таким образом, использование образцов сокращает затраты труда на программирование в символических обозначениях. Объем окончательной программы и затраты на перекодирование из символьных в восьмеричные обозначения не уменьшаются.

Мы рассмотрели лишь один вариант метода программирования по образцу, когда образец написан в символических обозначениях и используется для составления программы сразу в истинных адресах. Возможны и другие варианты. Можно составить образец в так называемых *условных* адресах, заменив каждый адрес в подпрограмме 1 произвольно выбранным восьмеричным адресом, не связанным с конкретным распределением памяти. При этом каждый символический адрес должен всюду быть заменен одним и тем же условным адресом, а разным символическим адресам должны соответствовать разные условные адреса. Условные адреса должны быть легко отличимы от истинных адресов, не подлежащих замене при составлении программы по образцу. Для этого их выбирают из одного сравнительно узкого диапазона, в который не могут попасть истинные адреса.

Если мы, например, хотим пересоставить в условных адресах программу 1, то истинными будут, разумеется, адреса 0000 и 0001. Возможно также выбрать неизменный стандартный адрес для кон-

станты 1, допустим, тот же адрес 7761. Остальные адреса следует заменить условными, например, в соответствии с таблицей

y	3000	f_k	3005
i	3001	c	3006
$p_n + 10$	3002	f	3007
$p_n + 4$	3003	r	3010
f_n	3004	$+n$	3011

Образец примет вид:

Программа 3

000	0000	0000	3000
000	7761	0000	3001
452	0000	0001	3002
016	3003	3004	3005
405	3006	3007	3010
001	3000	3010	3000
001	3001	7761	3001
112	3011	3004	0001
052	0000	0000	0000

Если этот образец предполагается использовать для составления программы вычисления суммы (3) в описанных выше условиях, то локальная таблица распределения памяти будет следующей:

3000	5013	3005	2354
3001	5007	3006	6730
3002	3673	3007	5110
3003	3667	3010	4522
3004	2201	3011	0014

В таком виде образец в условных адресах (программа 3) не очень удобен — не говоря уже о потере наглядности, в нем утрачена информация о том, что истинные адреса, соответствующие условным адресам 3002 и 3003, должны находиться в определенном соотношении как между собой, так и с начальным адресом участка программы, составляемого по этому образцу. От этого недостатка можно избавиться, но мы больше не будем останавливаться на программировании в условных адресах. Ясно, что этот метод практически неприемлем при ручном программировании. Программировать сразу в условных адресах трудно (достаточно сравнить обозначения y и 3000, чтобы убедиться в этом), а если предварительно составить программу в символических адресах, то она может служить образцом ничуть не хуже, чем программа в условных адресах. Поэтому программы в условных адресах применяются в наши дни лишь как промежуточный этап в некоторых системах автоматического программирования.

В еще одном варианте метода программирования по образцам как сам образец, так и программа, которая по нему составляется, записываются в символических обозначениях. Например, в условиях примера, который мы рассматриваем в этом разделе, можно было бы от программы 1, пользуясь таблицей соответствия

p_n	p_{1n}	f_k	φ_k
y	u	c	a
i	k	f	φ
f_n	φ_n	$+n$	14_8

перейти к программе:

Программа 4

p_{1n}	0	ПЧ	0	0	u	$u := 0$
	0	ПЧ	$\langle 1 \rangle$	0	k	$k := 1$
	4	АИ	0	1	$p_{1n} + 10$	$I := 1$, запоминание $I_{осн}$
	0	ПВ	$p_{1n} + 4$	φ_n	φ_k	Переход на φ_n с возвратом
$p_{1n} + 4$	4	$\times$	a	φ	r	$a_k \times \varphi(z, k) (I = k)$
	0	$+$	u	r	u	$u := u + a_k \varphi(z, k)$
	0	$+$	k	$\langle 1 \rangle$	k	} $k := k + 1$; если $k \leq 14_8$, то на φ_n
	1	$I < A$	14	φ_n	1	
	0	АИ	0	0	0	Восстановление $I_{осн}$

Так составленный участок программы может быть впоследствии еще раз преобразован, уже к истинным адресам. При этом нужно будет пользоваться не локальным, а общим распределением памяти. Подобный переход от одних символических обозначений к другим целесообразен, в частности, если ранее составленную подпрограмму (образец) мы хотим включить в другую, более обширную подпрограмму, образец для которой мы сейчас составляем, причем в этой новой подпрограмме все или часть символических обозначений отличаются от обозначений в прежнем образце.

К только что рассмотренному варианту метода программирования по образцу примыкает метод *макроопераций* или, как их еще называют, *макросов*. Если к программе добавить заголовок, снабжающий ее каким-либо наименованием и содержащий список символических адресов, которые при обращении к ней должны быть заменены конкретными параметрами, то мы получим *определение макрооперации*. В случае программы 1 заголовок может иметь вид

Макро $P(y, x, i, c, f_n, f_k, f, n)$

Здесь наименование P является как бы кодом макрооперации, а обозначения $y, x, \dots, n$ — ее обобщенными адресами. Программа 1 (тело макрооперации) задает способ выполнения этой макрооперации. После того как макрооперация определена таким образом, в основной программе можно написать *макрокоманду*, указываю-

щую, что в данном месте основной программы должна быть выполнена эта макрооперация с конкретными значениями параметров. Так, для вычисления суммы (3) макрокоманда, соответствующая приведенному выше заголовку макрооперации, должна иметь вид:

$$P(u, z, k, a, \varphi_n, \varphi_k, \varphi, 14). \quad (4)$$

Макрокоманда вместе с заголовком макрооперации определяют таблицу соответствия, по которой, пользуясь телом макрооперации как образцом, можно составить группу обычных команд (в символических адресах), заменяющую макрокоманду при окончательной компоновке основной программы. Такую замену можно осуществить вручную, но существуют системы автоматического программирования, в которых эту замену выполняет сама машина.

Ясно, что метод макроопераций — это прообраз аппарата процедур в АЛГОЛе (и других языках программирования). Определение макрооперации аналогично описанию процедуры, а макрокоманда — оператору процедуры. Например, если в какой-нибудь АЛГОЛ-программе содержится описание процедуры (2), то для вычисления суммы (3) к этой процедуре следует обратиться посредством оператора процедуры

$$P(z, 12, a, fi, u),$$

имеющего большое сходство с макрокомандой (4).

§ 3. Неизменяемые подпрограммы

В предыдущем параграфе мы выбирали распределение памяти, исходя только из условий обращения к подпрограмме. Это распределение оказывалось различным для каждого обращения. Нельзя ли поступить наоборот — принять неизменное распределение памяти для всех величин, фигурирующих в подпрограмме? Тогда вся подпрограмма уже в истинных, а не только в символических адресах окажется неизменной, но зато каждое обращение придется записывать, считаясь с принятым распределением.

Очевидно, в некоторых случаях такой прием не только возможен, но и довольно удобен. Пусть, например, требуется составить подпрограмму вычисления функции одной переменной, для определенности, функции

$$s = f(n) = \sum_{i=1}^n \frac{1}{i}. \quad (1)$$

Такая программа (программа 7.2.7) уже была составлена. Дополним ее еще одной ячейкой для команды возврата из подпрограммы

и выберем распределение памяти, например,

$f_n = p$	3000
n	0001
s	0002
t	0003
r	0004
$f_k = p + 7$	3007
$\langle 1 \rangle$	3010

Подпрограмма примет вид:

Программа 1									
3000	000	0000	0000	0002	$s := 0$				
1	000	3010	0000	0003	$t := 1$				
2	004	3010	0003	0004	$1/i$				
3	001	0002	0004	0002	$s := s + 1/i$				
4	001	0003	3010	0003	$t := t + 1$				
5	002	0001	0003	0000	Если $i \leq n$, то				
6	076	0000	3002	0000	перейти на 3002				
7	016	0000	0000	0000	ячейка возврата				
3010	101	4000	0000	0000	1 (число в коде M-20)				

Пусть теперь требуется вычислить в ячейке t значение $f(m)$. Это можно сделать, обратившись к подпрограмме 1 командами

k	0	ПЧ	m	0	n	$n := m$
$k+1$	0	ПВ	$k+2$	f_n	f_k	Переход на f_n с возвратом
$k+2$	0	ПЧ	s	0	t	$t := f(m)$

где адреса m и t зависят от условий обращения, а n , s , f_n и f_k — неизменные адреса, равные соответственно 0001, 0002, 3000 и 3007.

Из этих трех команд обращения к подпрограмме первая и последняя часто оказываются излишними. Так, например, команду $k+2$ можно не включать в программу, если не заводить специальной ячейки t для хранения $f(m)$, а пользоваться ячейкой s (0002), где подпрограмма 1 вычисляет это значение. Аналогично, можно не выделять специальной ячейки и для m , а использовать ячейку n (0001), куда значение m все равно должно быть заслано. Однако такая экономия ячеек памяти и команд пересылки возможна не всегда. Пусть требуется вычислить выражение

$$y = \frac{f(m_1)f(m_2)}{m_1 + m_2}. \quad (2)$$

Во время повторного обращения к подпрограмме 1 для вычисления $f(m_2)$ содержимое ячеек n и s , сохранившееся от вычисления $f(m)$, будет испорчено, а значения m_1 и $f(m_1)$ потребуются в последующих вычислениях. Поэтому участок программы, вычисляющий значение y по формуле (2), может быть, например, таким:

Вычисление значения m_1 в
выделенной для него ячейке

k	0	ПЧ	m_1	0	n	$n := m_1$
$k+1$	0	ПВ	$k+2$	f_n	f_k	Переход на f_n с возвратом
$k+2$	0	ПЧ	s	0	y	$y := f(m_1)$

Вычисление значения
 m_2 в ячейке n

k_1	0	ПВ	k_1+1	f_n	f_k	Переход на f_n с возвратом
k_1+1	0	×	y	s	y	$f(m_1) \times f(m_2)$
k_1+2	0	+	m_1	n	n	$m_1 + m_2$
k_1+3	0	/	y	n	y	$y := f(m_1) \times f(m_2) / (m_1 + m_2)$

У п р а ж н е н и е 1. Составить возможно более экономную программу для вычисления значения $f(n) + f(2n)$, используя подпрограмму 1 для функции f .

Неизменяемые подпрограммы могут быть составлены и в более сложных случаях. Покажем это на примере подпрограммы для приближенного вычисления интеграла по формуле трапеций с заданным числом шагов на заданном промежутке. Формула трапеций имеет вид

$$I = \int_a^b f(x) dx \approx \left[\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a + ih) \right] \cdot h,$$

где $h = (b - a)/n$. Для вычислений по этой формуле можно воспользоваться следующей подпрограммой:

Программа 2

p_n	0	—	b	a	h	$b - a$
	0	/	h	n	h	$h := (b - a)/n$
	0	ПЧ	b	0	x	$x := b$
	0	ПВ	p_1	f_n	f_k	Переход на f_n с возвратом
p_1	0	ПЧ	y	0	I	$f(b)$
	0	ПЧ	a	0	x	$x := a$
	0	ПВ	p_2	f_n	f_k	Переход на f_n с возвратом
p_2	0	+	I	y	I	$f(b) + f(a)$
	0	+АП	0077	I	I	$I := (f(b) + f(a))/2$
	0	ПЧ	$\langle 1 \rangle$	0	i	$i := 1$
p_3	0	+	x	h	x	$x := a + ih$
	0	ПВ	p_4	f_n	f_k	Переход на f_n с возвратом
p_4	0	+	I	y	I	$I := I + f(a + ih)$
	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
	0	—	i	n	0	Если $i < n$, то ($\omega := 1$)
	0	П1	0	p_3	0	перейти на p_3
	0	×	I	h	I	$I := I \times h$
p_k	0	0	0	0	0	Ячейка возврата

Эта подпрограмма сама обращается к подпрограмме вычисления подынтегральной функции, которую для краткости назовем под-

программой f . Адрес входа в подпрограмму f обозначим f_n , адрес ячейки возврата из этой подпрограммы — f_k . Подпрограмма f берет свой аргумент из ячейки x и засылает результат в ячейку y . В подпрограмме 2 содержатся три обращения к подпрограмме f , точнее, есть три места, из которых производится такое обращение. В первом случае (команды $p_1 - 2 \div p_1$), кроме команды перехода на подпрограмму f с возвратом ($p_1 - 1$), потребовались еще две команды для пересылки аргумента b в ячейку x и результата y в ячейку I . Во втором случае (команды $p_2 - 2, p_2 - 1$) понадобилась лишь пересылка аргумента a . В третьем случае обращение свелось лишь к одной команде перехода с возвратом ($p_4 - 1$), а засылка аргумента и извлечение результата совмещены с арифметическими операциями (команды p_3 и p_4) и не требуют отдельных команд. Заметим, что это третье (по месту записи) обращение выполняется многократно ($n - 1$ раз) и для каждого следующего (по времени) обращения значение аргумента x вычисляется добавлением шага h к прежнему значению x . Поэтому подпрограмма f не должна портить содержимого ячейки x . Ясно, что когда подпрограмма 2 уже составлена, можно составить подпрограмму f так, чтобы она удовлетворяла предъявленным к ней требованиям, т. е. чтобы адреса f_n , f_k , x и y имели заданные значения и чтобы эта подпрограмма во время своей работы не меняла содержимого ячейки x . В рассматриваемом примере такой подход оправдан, так как по отношению к подпрограмме вычисления интеграла подпрограмма вычисления подынтегральной функции является подчиненной, и поэтому естественно, чтобы вторая составлялась с учетом требований, исходящих от первой. Если это так, то для обращения к подпрограмме 2 достаточно заслать исходные значения параметров в ячейки a, b, n . После выхода из подпрограммы приближенное значение интеграла будет находиться в ячейке I . Предположим, что в конкретной задаче требуется вычислить интеграл от функции $y = f(x)$, подпрограмма вычисления которой составлена с учетом перечисленных выше требований. Пусть границы промежутка интегрирования помещаются в ячейках a_1 и b_1 , а число шагов — в ячейке n_1 . Пусть, наконец, вычисленное значение интеграла требуется поместить в ячейку I_1 . В этих условиях для обращения к подпрограмме 2 нужны команды:

k	0	ПЧ	a_1	0	a	$a := a_1$
$k+1$	0	ПЧ	b_1	0	b	$b := b_1$
$k+2$	0	ПЧ	n_1	0	n	$n := n_1$
$k+3$	0	ПВ	$k+4$	p_n	p_k	Переход на p_n с возвратом
$k+4$	0	ПЧ	I	0	I_1	$I_1 := I$

Как в предыдущих примерах, здесь можно обойтись без некоторых (или даже без всех) команд $k, k+1, k+2, k+4$, если исходные значения удастся вычислить сразу в нужных ячейках, а результат

достаточно иметь лишь в ячейке 1. Экономия команд обращения (а также используемых ячеек памяти) достигается благодаря тому, что в основной программе участок вычисления исходных данных для обращения к подпрограмме и участок, на котором используются результаты работы подпрограммы, совмещаются с участком собственно обращения к подпрограмме. Однако величины, используемые в качестве параметров обращения (например, границы промежутка интегрирования), могут также иметь в основной программе назначение, вовсе не связанное с обращением к подпрограмме. Поэтому ясно, что экономия команд приводит в данном случае к менее четкой логической схеме программы, к тому, что способ составления одних частей программы существенно зависит от вида других ее частей.

§ 4. Элементы модульного программирования

Пока число частей, на которые распадается программа, невелико, а сами части не очень громоздки и устроены просто, зависимость одних частей от других не приводит к особым неприятностям. Взаимные связи и взаимное влияние частей легко прослеживаются и могут без труда быть учтены при составлении каждой части. Но чем больше и сложнее программа, тем больше осложнений в составлении и отладку программы вносит взаимозависимость ее частей. Разумеется, от такой взаимной зависимости полностью избавиться невозможно. Однако можно попытаться ограничить эту зависимость определенными рамками.

Постараемся удовлетворить следующим требованиям.

1. Части программы должны быть неизменяемы. Как минимум, это означает, что любая часть программы не должна меняться при изменении значений параметров обращения к ней. Но можно понимать это требование и в более широком смысле — отдельная часть программы не должна требовать переделок при изменении или даже полной замене других частей, лишь бы при этом не менялся состав параметров обращения.

2. Распределение памяти в каждой части не должно зависеть от распределения памяти, принятого в других частях (в предположении, что участки памяти, выделенные для каждой части, не перекрываются).

3. Схема связи между частями (передачи информации от одной части к другой) должна быть простой и единообразной.

При таком подходе независимые (до известной степени) части, на которые распадается программа, принято называть *модулями*, а сам подход — *модульным программированием*. Этому подходу будут посвящены все оставшиеся параграфы. Но и при этом нам не удастся разобрать все особенности модульного программирования. Ограничимся лишь основными понятиями.

Продолжим исследование предыдущего примера. С точки зрения, на которую мы стали, в нем следует выделить, по меньшей мере, три модуля: основной модуль, в котором возникает потребность вычислить интеграл и вырабатываются исходные данные для этого вычисления, подпрограмму 15.3.2 (модуль вычисления интеграла по формуле трапеций), подпрограмму f (модуль вычисления подынтегральной функции). Кроме этих модулей, программа примера должна содержать *связующие части*, организующие совместную работу модулей.

Каждый модуль обладает своими рабочими ячейками. Так, мы уже условились, что основной модуль готовит исходные данные в ячейках a_1, b_1, n_1 и ждет, что значение интеграла будет заслано в ячейку I_1 . Подпрограмма 15.3.2 использует ряд рабочих ячеек, в частности, ячейки a, b, n, x, y, I . В духе модульного подхода следует считать, что подпрограмма f составляется независимо от подпрограммы 15.3.2, что она берет значение аргумента, скажем, из ячейки x' и помещает значение подынтегральной функции в ячейку y' , что адрес входа в эту подпрограмму (ее начальной команды) равен f_n , а адрес выхода (ячейки, куда попадает управление по завершении вычисления функции) равен f_k . Следовательно, ячейки f_n и f_k уже не относятся к подпрограмме f , их (наряду с ячейками x, y и др.) следует считать рабочими ячейками подпрограммы 15.3.2.

Как же должна выглядеть часть, связывающая подпрограммы 15.3.2 и f ? Очевидно, в нее должны входить команды

q	0	ПЧ	x	0	x'	$x' := x$
$q+1$	0	ПВ	$q+2$	f'_n	f'_k	Переход на f'_n с возвратом
$q+2$	0	ПЧ	y'	0	y	$y := y'$

Но этого не достаточно. В ячейку f_n надо поместить команду

0 ПБ 0 q 0

передающую управление связующей части, а к командам $q \div q+2$ присоединить команду

$q+3$ 0 ПБ 0 f_k 0

обеспечивающую возврат в подпрограмму 15.3.2. Особенности системы команд машины М-20 позволяют поступить экономнее: в ячейку f_n заслать команду

z 0 ПБ x $q+1$ x' (1)

а связующую часть составить из двух команд

$q+1$	0	ПВ	$q+2$	f'_n	f'_k
$q+2$	0	ПБ	y'	f_k	y

Засылка команды (1) в ячейку f_n — это одна из подготовительных операций для обращения из основного модуля к подпрограмме

15.3.2. Эта операция должна входить в часть, связывающую эти два модуля, которая может быть такой:

k	0	ПЧ	a_1	0	a
$k+1$	0	ПЧ	b_1	0	b
$k+2$	0	ПЧ	n_1	0	n
$k+3$	0	ПЧ	z	0	f_n
$k+4$	0	ПВ	$k+5$	p_n	p_k
$k+5$	0	ПЧ	I	0	I_1

Но и это еще не все. Если основной модуль составляется независимо от подпрограммы 15.3.2, то в нем команда обращения к модулю вычисления интеграла имеет вид

0 ПВ ... p_n p_k ,

где p_n и p_k — это адреса рабочих ячеек основного модуля, а многооточие обозначает адрес возврата в основной модуль (адрес, не известный нам, но известный составителю основного модуля). Тогда команду $k+5$ следует заменить командой

$k+5$ 0 ПБ I p_k I_1 ,

а также предусмотреть засылку заготовки

z_1 0 ПБ 0 k 0

в ячейку p_n (усложняя вид этой заготовки, можно сэкономить команду k , подобно тому, как команда q была исключена из части, связывающей подпрограмму 15.3.2 с подпрограммой f). Команда

0 ПЧ z_1 0 f_n

аналогичная команде $k+3$, должна войти в состав группы команд обращения к основному модулю.

Схему взаимодействия модулей можно проследить на рис. 15.4.1, где двойные стрелки изображают передачу управления, одинарные — пересылку информации, пунктирные — засылку команд.

Такая схема организации совместной работы модулей может показаться чересчур сложной. Однако ее несомненное достоинство в том, что она позволила обеспечить неизменяемость модулей и независимое распределение памяти в них. Неизменяемость модулей в нашем примере проявляется, в частности, в постоянстве адресов p_n и p_k в основном модуле независимо от того, какой модуль используется для вычисления интеграла и где он расположен, и в независимости адресов f_n и f_k в программе 15.3.2 от вида подынтегральной функции и от места расположения модуля f .

Независимое распределение памяти позволяет в нашем примере писать основной модуль, не заботясь о том, как составлены и как работают обе подпрограммы. Например, разделение ячеек a_1 , b_1 , n_1 и a , b , n позволяет программисту, составляющему основной

модуль, не интересоваться тем, портит или сохраняет подпрограмма 15.3.2 содержимое ячеек a , b и n , куда засылаются ее аргументы. Даже если это содержимое портится, значения аргументов сохраняются в ячейках a_1 , b_1 и n_1 и могут быть использованы

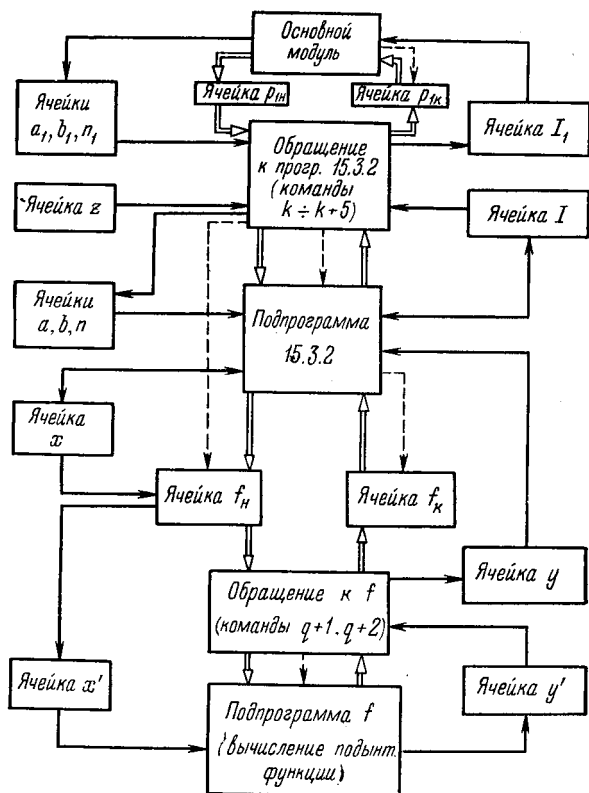


Рис. 15.4.1. Схема связей между модулями.

в вычислениях, выполняемых после выхода из подпрограммы. Любая из подпрограмм также может быть составлена без учета того, как написаны другие модули. Это особенно важно по отношению к подпрограмме 15.3.2. Действительно, подпрограмма вычисления интеграла, по-видимому, может оказаться полезной в очень многих программах, которые будут составляться, как правило, уже после того, как составлена и отлажена эта подпрограмма. Поэтому автор подпрограммы и не может ничего знать об устройстве и работе этих программ. Ему важно написать свою подпрограмму так, чтобы она работала надежно и правильно, как бы впоследствии ни изощря-

лись программисты, составляющие другие программы, которые будут обращаться к его подпрограмме.

Разумеется, свобода действий этих программистов должна быть поставлена в какие-то рамки. В частности, автор основного модуля должен знать, что для вычисления интеграла в его модуле следует предусмотреть вычисление пределов интегрирования и числа шагов, но бесполезно, например, вычислять шаг интегрирования, поскольку подпрограмма 15.3.2 вычислит его сама.

В общем случае рассматриваемая схема связи между модулями требует, чтобы составитель модуля M_i , обращающегося к модулю M_j , знал количество k (и, разумеется, смысл) переменных $x_1, \dots, x_k$, значения которых являются исходными для работы модуля M_j , а также количество s переменных $y_1, \dots, y_s$, значения которых модуль M_j вырабатывает в качестве результата. В каждом из этих модулей как для переменных $x_1, \dots, x_k$, так и для переменных $y_1, \dots, y_s$ должны быть выделены в силу основных принципов модульного программирования свои рабочие ячейки. Пусть обозначения $x_1^i, \dots, x_k^i$ заменяют адреса ячеек, выделенных для хранения значений переменных $x_1, \dots, x_k$ в модуле M_i . Аналогичный смысл имеют обозначения $x_1^j, \dots, x_k^j$; $y_1^i, \dots, y_s^i$ и $y_1^j, \dots, y_s^j$. Тогда часть модуля M_i , связанная с обращением к модулю M_j , должна быть такой:

$$\begin{array}{l}
 \left[\begin{array}{l} \text{Команды, вычисляющие значения} \\ \text{переменных } x_1, \dots, x_k \text{ в} \\ \text{ячейках } x_1^i, \dots, x_k^i \end{array} \right] \\
 p \quad 0 \quad ПВ \quad p+1 \quad p_{nj}^i \quad p_{kj}^i \quad (2) \\
 \left[\begin{array}{l} \text{Команды, использующие значения} \\ \text{переменных } y_1, \dots, y_s \text{ из} \\ \text{ячеек } y_1^j, \dots, y_s^j \end{array} \right]
 \end{array}$$

Здесь p_{nj}^i и p_{kj}^i — адреса рабочих ячеек модуля M_i , используемых для передачи управления модулю M_j . Однако этот переход управления осуществляется не прямо, а через специальный блок связи, имеющий совершенно стандартный вид:

$$\begin{array}{l}
 \left. \begin{array}{lllll} r_{ij} & 0 & ПЧ & x_1^i & 0 & x_1^j \\ \dots & \dots & \dots & \dots & \dots & \dots \\ r_{ij+k-1} & 0 & ПЧ & x_k^i & 0 & x_k^j \end{array} \right\} \begin{array}{l} \text{Пересылка} \\ \text{аргументов} \\ \text{обращения} \end{array} \\
 \begin{array}{lllll} r_{ij+k} & 0 & ПВ & r_{ij+k+1} & p_{nj} & p_{kj} \\ r_{ij+k+1} & 0 & ПЧ & y_1^j & 0 & y_1^i \\ \dots & \dots & \dots & \dots & \dots & \dots \\ r_{ij+k+s-1} & 0 & ПЧ & y_{s-1}^j & 0 & y_{s-1}^i \\ r_{ij+k+s} & 0 & ПВ & y_s^j & p_{kj}^i & y_s^i \end{array} \left. \right\} \begin{array}{l} \text{Переход на } M_j \text{ с возвратом} \\ \text{Пересылка} \\ \text{результатов} \\ \text{обращения} \end{array}
 \end{array}$$

В этом блоке p_{nj} и p_{kj} — уже истинные адреса входа и выхода модуля M_j . Команда p засылает в ячейку p_{kj}^i команду

$$0 \text{ ПВ } 0 \text{ } p+1 \text{ } 0$$

чем обеспечивается возврат в модуль M_i (через блок связи). Осталось лишь потребовать, чтобы ячейка p_{nj}^i содержала команду

$$0 \text{ ПБ } 0 \text{ } r_{ij} \text{ } 0 \\ (\text{ПВ})$$

Для этого среди команд пересылки аргументов обращения к модулю M_i (аналогичных командам $r_{ij} \div r_{ij} + k - 1$) должна содержаться команда

$$q \quad 0 \text{ ПЧ } z_{ij} \text{ } 0 \text{ } p_{nj}^i \quad (3)$$

где z_{ij} — адрес заготовки

$$z_{ij} \quad 0 \text{ ПБ } 0 \text{ } r_{ij} \text{ } 0$$

Команда q однотипна со всеми другими командами пересылки аргументов. Можно записать вместо нее команду

$$q \quad 0 \text{ ПВ } r_{ij} \text{ } q+1 \text{ } p_{nj}^i$$

Нарушение стиля команд пересылки аргументов обращения окунается тем, что не нужно запастись заготовку z_{ij} .

После такой стандартизации команд связи между модулями для общего случая, становится ясно, что схема связи совсем не сложна. Следующее сомнение, которое может возбудить эта схема, — не слишком ли много команд занимает каждое обращение из одного модуля к другому? Если обращение к модулю возникает лишь в одном месте программы, то ради экономии памяти и сокращения потерь времени можно, используя модуль как образец, переписать его на этом месте, учитывая то распределение памяти, которое принято для параметров обращения. Но такое переписывание более трудоемко, чем составление шаблонных команд обращения, и сопряжено со значительно большей возможностью появления ошибок. Если же обращаться к модулю нужно из разных мест программы с разными параметрами, то описанные выше приемы организации совместной работы модулей позволяют иметь каждый модуль в одном экземпляре, выписывая для каждого обращения лишь команды связи. При этом, как правило, экономится даже память по сравнению с методом программирования по образцам.

При составлении модулей удобно (хотя и не обязательно) объединять все рабочие ячейки одного модуля (например, ячейки a , b , n , l , f_n , f_k , x , y , h , i для подпрограммы 15.3.2) в один массив и располагать этот массив рядом с массивом команд модуля (непосредственно перед ячейкой p_n или вслед за ячейкой p_k). Такое компактное размещение в памяти всех ячеек, связанных с работой одного

модуля, облегчает компоновку программы, распределение памяти и даже отладку программы. Если почему-либо модуль при переработке программы окажется ненужным, то занятая им память освободится одним куском — это тоже удобно.

Далее, независимость модулей друг от друга делает возможным проводить экономии ячеек в пределах модуля, не считаясь с тем, что делается вне него. Так, в подпрограмме 15.3.2 используются ячейки a, b, n, x, y, l, h, i . Содержимое ячейки a перестает быть нужным подпрограмме после выполнения команды $p_1 + 1$. Необходимость в ячейке i впервые возникает позднее — при выполнении команды $p_2 + 2$. Правильность работы подпрограммы не нарушится, если при распределении памяти ячейки a и i будут совмещены, т. е. если символическим адресам a и i будет поставлен в соответствии один и тот же истинный адрес. Но при этом подпрограмма не будет сохранять значение переменной a в этой совмещенной ячейке. Как мы уже говорили, для основного модуля и других подпрограмм безразлично, сохранит или испортит подпрограмма содержимое ячейки a . Поэтому совмещение ячеек a и i , допустимое с точки зрения правильности работы подпрограммы 15.3.2, приемлемо и с точки зрения правильной работы всех других модулей программы. Из тех же соображений допустимо, чтобы символические адреса b и l заменяли один и тот же истинный адрес.

§ 5. Настройка модулей по параметрам

Является ли схема организации модулей, описанная в предыдущем параграфе, достаточно универсальной? Чтобы ответить на этот вопрос, рассмотрим модуль, представленный подпрограммой 15.2.1. В таком виде, в каком он написан, он не удовлетворяет требованиям описанной выше схемы. Во-первых, он не может оставаться неизменным, если изменяется значение параметра n при обращении к этому модулю *), потому что это значение фигурирует в качестве адреса $+n$ в команде $p_{11} + 7$ и, следовательно, при изменении n должен измениться сам этот адрес. Во-вторых, одним из параметров обращения к подпрограмме 15.2.1 является массив c , а схема § 4 предусматривает передачу информации от модуля к модулю только в виде содержимого отдельных ячеек.

Прямолинейное обобщение этой схемы — разрешение передавать информацию массивами — нельзя считать приемлемым. В са-

*) Здесь мы допускаем некоторую вольность в терминологии. Вместо «параметр n » следовало бы сказать «параметр обращения, соответствующий величине, фигурирующей в модуле под обозначением n ». В примере, рассмотренном в § 2, таким параметром было число 12₁₀, при других обращениях этот параметр может принимать совсем другие значения. В дальнейшем мы будем постоянно прибегать к такой сокращенной терминологии, если это не может привести к недоразумениям.

мом деле, образование в памяти копии массива только для того, чтобы модуль мог работать с этой копией, а не с оригиналом, — это слишком расточительно с точки зрения экономии как времени работы программы, так и места в памяти. Далее, память, занятую такой копией массива, придется считать частью рабочей памяти модуля. Но тогда размер этой рабочей памяти будет зависеть от параметров обращения к модулю. Нетрудно понять, что это заметно осложнит распределение памяти под модули — придется либо распределять память с учетом всех предстоящих обращений к модулю, либо отказаться от предварительного (во время составления программы) распределения памяти и производить такое распределение во время работы программы. Такое *динамическое*, как его называют, распределение памяти уже не может произвести программист, его должна делать сама программа. С этой задачей можно справиться, но без особой нужды не следует идти по этому пути. Главное, в нашей задаче нет никакой надобности копировать массив c — подпрограмма 15.2.1 только использует, но не изменяет значения элементов этого массива. Поэтому нет оснований опасаться, что во время работы подпрограммы массив будет испорчен, а только это могло бы быть серьезной причиной для создания копии массива, с которой работала бы подпрограмма.

Но если подпрограмма 15.2.1 должна работать с тем массивом c , который задается в обращении к ней, то адрес c в команде $p_n + 4$ должен меняться от обращения к обращению.

Из рассмотренных примеров видно, что схема с неизменяемыми модулями хороша, пока все параметры обращения к модулю являются параметрами-значениями (см. § 1). Параметры-адреса, такие, как параметры n и c в подпрограмме 15.2.1, не укладываются в эту схему и требуют, чтобы вид команд модуля изменялся в зависимости от значений этих параметров. Преобразование модуля, заключающееся в замене некоторых его адресов значениями соответствующих параметров-адресов в обращении, называется *настройкой модуля по параметрам*.

Задача настройки модуля по параметрам может решаться разными способами. Одно из простейших, на первый взгляд, решений сводится к тому, что вся команда, содержащая адрес, зависящий от параметра, рассматривается как дополнительный параметр обращения. Тогда при обращении к подпрограмме надо заслать в ячейку, отведенную под такую команду, соответствующую заготовку.

Составим обращение к подпрограмме 15.2.1 в духе предыдущего параграфа, но с учетом зависимости команд $p_n + 4$ и $p_n + 7$ от параметров. Пусть из модуля M_0 производится обращение к подпрограмме 15.2.1 для вычисления суммы (15.2.3). Тогда переменным x и y в подпрограмме 15.2.1 соответствуют переменные z и u в модуле M_0 , адресу $+n$ — значение $12_{10} = 14_8$, адресу c — базовый адрес a массива, размещающегося в ячейках $a + 1, \dots, a + 14$.

Само обращение осуществляется командой

$$0 \text{ ПВ } \dots p_{n1}^0 p_{k1}^0$$

Для связи модуля M_0 с подпрограммой 15.2.1 необходимо в этих условиях написать блок:

Программа 1

$k+1$	0	ПЧ	z	0	x
$k+2$	0	ПЧ	z_1	0	p_n+4
$k+3$	0	ПЧ	z_2	0	p_n+7
$k+4$	0	ПВ	$q+1$	$k+5$	f_n
$k+5$	0	ПВ	$k+6$	p_n	p_n+11
$k+6$	0	ПВ	y	p_{k1}^0	u

где заготовки z_1 и z_2 имеют вид

$$\begin{array}{lcl} z_1 & 4 & \times \quad a \quad f \quad r \\ z_2 & 1 & H < A \quad 14 \quad f_n \quad 1 \end{array}$$

$q+1$ — начальный адрес блока связи подпрограммы 15.2.1 с модулем вычисления функции f , а f , r и f_n — рабочие ячейки подпрограммы 15.2.1. Необходимо также при обращении к модулю M_0 предусмотреть засылку в ячейку p_{n1}^0 заготовки

$$z \quad 0 \text{ ПВ } 0 \quad k+1 \quad 0 \\ (\text{ПБ})$$

Казалось бы, мы полностью уложились в схему § 4. Однако заготовки z_1 и z_2 портят дело. Они имеют слишком нестандартный вид, и если при составлении блоков связи придется запастись много таких заготовок, то простота и стройность схемы полностью рушатся. А ведь программа 15.2.1 очень проста. Обратимся к немного более сложному примеру — подпрограмме 9.2.4. В обращении к ней участвуют только два параметра: a и n , оба — параметры-адреса. Для работы подпрограммы нужны 9 констант ($z_2 \div z_8$ и $z_8 \div z_{11}$), вид которых зависит от этих параметров. Если следовать только что избранному пути, то при каждом обращении к подпрограмме 9.2.4 (с новыми параметрами) придется запастись все эти заготовки и писать девять команд их пересылки в рабочие ячейки подпрограммы. (Какие рабочие ячейки подпрограммы здесь имеются в виду? Для заготовок z_9 , z_{10} и z_{11} это могут быть прямо ячейки r_3 , r_4 и r_5 , тогда в самой подпрограмме команды пересылки этих заготовок в эти ячейки будут уже не нужны, а команду

$$0 \text{ ПЧ } z_9 \quad 0 \quad r_2$$

можно заменить командой

$$0 \text{ ПЧ } r_3 \quad 0 \quad r_2$$

Но для остальных констант ячейки с адресами $z_2 \div z_8$ и z_8 следует рассматривать как рабочие и засылать в них нужное содержимое при обращении к подпрограмме.) Поэтому попытаемся организовать работу с модулями так, чтобы параметр-адрес сообщался модулю в более или менее чистом виде, а использование этого адреса внутри модуля было бы делом самого модуля.

В программах для машин типа М-20 параметр-адрес a проще всего передавать в составе машинного слова (заготовки)

$$0 \text{ АИ } 0 \text{ а } 0 \quad (1)$$

Пусть в модуле M_1 , к которому мы обращаемся, под такое слово выделена рабочая ячейка a_1 . Тогда в блоке связи, через который происходит обращение к модулю M_1 (в группе команд засылки аргументов), для посылки слова (1) в эту ячейку можно воспользоваться командой

$$2 \text{ АИ } a \text{ } 0 \text{ а}_1$$

но можно, конечно, применить и команду привычного вида

$$0 \text{ ПЧ } a_0 \text{ } 0 \text{ а}_1$$

если в модуле M_0 , из которого производится обращение, уже существует ячейка a_0 , в которой хранится заготовка (1).

Таким образом, мы сводим случай параметра-адреса a к случаю параметра-значения, представленного словом (1). Для использования информации, переданной модулю с помощью слов такого вида, к модулю добавляется *формирующая часть*. Ее назначение — сформировать все команды, вид которых зависит от параметров-адресов, т. е. команды типа команд $p_n + 4$ и $p_n + 7$ в подпрограмме 15.2.1. Покажем на ее примере, как составляется эта часть. Воспользуемся приемами, известными нам по § 4 гл. 12.

Пусть в рабочие ячейки c_1 и n_1 засланы заготовки

$$c_1 \quad 0 \text{ АИ } 0 \text{ с } 0$$

$$n_1 \quad 0 \text{ АИ } 0 \text{ +n } 0$$

Тогда формирующая часть может состоять из четырех команд

$$\begin{array}{llllll} 0 & \leftarrow \text{МА} & 114 & c_1 & r & \text{«0 АИ с 0 0»} \\ 0 & +\text{М} & z_1 & r & p_n + 4 & \text{Формирование команды } p_n + 4 \\ 0 & \leftarrow \text{МА} & 114 & n_1 & r & \text{«0 АИ +n 0 0»} \\ 0 & +\text{М} & z_2 & r & p_n + 7 & \text{Формирование команды } p_n + 7 \end{array}$$

Используемые здесь заготовки имеют вид

$$\begin{array}{llll} z_1 & 4 & \times & 0 \text{ f } r \\ z_2 & 1 & \text{И} < \text{А} & 0 \text{ f}_n \text{ } 1 \end{array}$$

Они являются составной частью формируемого модуля. Весь модуль оказывается таким:

Программа 2

p_n	0	$\leftarrow MA$	114	c_1	r	} Формирующая часть
	0	$+M$	z_1	r	p_1	
	0	$\leftarrow MA$	114	n_1	r	
	0	$+M$	z_2	r	p_2	
	0	$ПЧ$	0	0	y	$y := 0$
	0	$ПЧ$	$\langle 1 \rangle$	0	i	$i := 1$
	4	$АИ$	0	1	p_3	Запоминание $I_{ст}$, $I := 1$
	0	$ПВ$	p_1	f_n	f_k	Переход на f_n с возвратом
p_1	4	$\times$	c	f	r	$c[i] \times f(x, i) (H = i)$
	0	$+$	y	r	y	$y := y + c[i] \times f(x, i)$
	0	$+$	i	$\langle 1 \rangle$	i	} $i := i + 1$,
p_2	1	$H < A$	$+n$	f_n	1	
p_3	0	$АИ$	0	$I_{ст}$	0	Восстановление $I_{ст}$
p_k	0	$ПВ$	0	$\sim$	0	Ячейка возврата

Программа 2 длиннее на 4 команды (не считая ячеек возврата), чем программа 15.2.1, и соответственно больше время ее работы. Кроме того, для нее необходимы две заготовки z_1 и z_2 . Но зато обращение к ней можно написать в виде

$k+1$	0	$ПЧ$	z	0	x
	2	$АИ$	a	0	c_1
	2	$АИ$	14	0	n_1
	0	$ПВ$	$q+1$	$k+5$	f_n
	0	$ПВ$	$k+6$	p_n	p_k
$k+6$	0	$ПВ$	y	p_{k1}^0	u

(обозначения те же, что в программе 1). Здесь уже заготовки не нужны, тогда как в программе 1 потребовались две заготовки, к тому же зависящие от параметров обращения. Следует учесть, что подпрограмма составляется один раз, а обращения к ней пишутся после этого многократно. Поэтому небольшое усложнение подпрограммы и еще менее существенное, на первый взгляд, упрощение команд обращения на деле оборачиваются большой экономией общего труда программистов — тем большей, чем чаще применяется данная подпрограмма в разных программах.

На тех машинах семейства М-20, на которых есть операции $H < K$ и $H \geq K$, программа 2 может быть сокращена. Надобность в командах $p_n + 2$ и $p_n + 3$, формирующих команду p_2 , а также в заготовке z_2 отпадает, так как команду p_2 можно записать в виде

p_2	1	$H < K$	n_1	f_n	1

Итак, мы несколько обогатили чересчур примитивную схему связи модулей, описанную в § 4. Теперь в ячейки $x_1^i, \dots, x_k^i$ модуля M_j , выделенные для засылки в них параметров обращения, эти параметры могут засылаться в одном из трех видов. В случае

параметра-значения вид его произволен и для его засылки по-прежнему применяется команда

$$0 \text{ ПЧ } x_q^i \ 0 \ x_q^j \quad (2)$$

Параметр-адрес засылается в ячейку x_q^j , как правило, в виде

$$0 \text{ АИ } 0 \ x_q \ 0 \quad (3)$$

Для этого используется команда вида

$$2 \text{ АИ } x_q \ 0 \ x_q^j$$

Из этого общего правила пересылки параметров-адресов целесообразно сделать исключение, если параметр x_q является адресом передачи управления. Такой параметр, как мы отмечали уже в § 4, естественно передавать в виде кода команды

$$0 \text{ ПВ } 0 \ x_q \ 0 \quad (4)$$

используя для этого команду засылки

$$r \quad 0 \text{ ПВ } x_q \ r+1 \ x_q^j$$

Предполагается, что значение параметра-адреса x_q известно во время написания блока связи между модулями M_i и M_j . Если же значение x_q вычисляется модулем M_i во время его работы, то такой параметр следует трактовать как параметр-значение. Модуль M_i должен сформировать это значение в виде (3) или (4) в ячейке x_q^j и для его пересылки в ячейку x_q^j может быть применена команда (2).

Модуль, среди параметров обращения к которому есть параметры-адреса, состоит из двух частей: формирующей и формируемой, или рабочей части. Рабочая часть модуля уже не является неизменяемой. Но при наличии формирующей части эта изменяемость рабочей части становится как бы внутренним делом модуля, она осложняет задачу лишь автора данного модуля, которому нужно составить, кроме рабочей, еще и формирующую часть. Программисту же, пишущему другие модули и обращения к данному модулю, изменяемость рабочей части никаких забот не доставляет — модуль всегда готов к работе, как бы ни менялись значения параметров обращения к нему. В этом отношении мы добились такого же эффекта, как если бы модуль был неизменяемым.

Составим формирующую часть к программе 9.2.4. Мы снова останавливаемся на этой программе лишь в учебных целях, т. к. для этой задачи существуют более компактные программы (см. упражнение 10.2.1). Итак, пусть в ячейках a_1 и n_1 находятся коды

$$\begin{array}{lllll} a_1 & 0 & \text{АИ} & 0 & a & 0 \\ n_1 & 0 & \text{АИ} & 0 & +n & 0 \end{array}$$

Сформировать требуется (см. выше) содержимое ячеек $r_2 \div r_5$, $z_2 \div z_6$, z_8 . Будем предполагать также, что в нашем распоряжении есть константы

z_1	0	0	1	0	0
z_7	0	0	0	0	1
z_9	0	ПЧ	0	0	r_1
z_{10}	0	ПЧ	r_1	0	0

не зависящие от параметров обращения. Учтем, что код операции ПЧ равен 00₈. Это упрощает формирование команд с таким кодом операции. Формирующая часть программы 9.2.4 может быть такой:

0	← МА	64	n_1	z_3	«0	АИ	0	0	+n»
0	+М	z_1	z_3	z_6	«0	0	1	0	+n»
0	+М	z_{10}	z_3	z_8	«0	ПЧ	r_1	0	+n»
0	← СА	114	n_1	z_5	«0	0	+n	0	0 »
0	+М	z_5	z_7	z_2	«0	0	+n	0	1 »
0	+М	z_5	z_1	z_1	«0	0	$n+1$	0	0 »
0	← МА	64	a_1	r_3	«0	АИ	0	0	a »
0	+М	z_8	r_2	z_8	«0	ПЧ	r_1	0	a+n»
0	+М	z_7	r_2	r_2	«0	0	0	0	a+1»
0	+М	z_{10}	r_2	r_5	«0	ПЧ	r_1	0	a+1»
0	← МА	130	r_5	r_3	«0	ПЧ	a+1	0	0 »
0	+М	r_3	r_2	r_4	«0	ПЧ	a+1	0	a+1»
0	+М	z_9	r_3	r_3	«0	ПЧ	a+1	0	r_1 »
0	ПЧ	r_3	0	r_2	«0	ПЧ	a+1	0	r_1 »

Заметим, что если в качестве рабочей ячейки r_1 использовать ячейку с адресом 0001 (в системах программирования, принятых для машин типа М-20, эта ячейка, как правило, и используется как рабочая ячейка для большинства подпрограмм), то заготовки z_9 и z_{10} становятся не нужны, а формирование заготовки z_8 упрощается.

У п р а ж н е н и е 1. Составить подпрограмму вычисления произведения C двух прямоугольных матриц: A , размером $m \times p$, и B , размером $p \times n$ (таким образом, матрица C имеет размер $m \times n$). Матрицы располагаются в памяти по строкам, так что адрес $a[i, j]$ равен $a + (i-1)p + j$, адрес $b[j, k]$ равен $b + (j-1)n + k$, адрес $c[i, k]$ равен $c + (i-1)n + k$. Снабдить подпрограмму формирующей частью, исходя из предположения, что к началу работы подпрограммы в ее рабочих ячейках содержатся следующие коды:

a_1	0	АИ	0	a	0
b_1	0	АИ	0	b	0
c_1	0	АИ	0	c	0
m_1	0	АИ	0	+m	0
n_1	0	АИ	0	+n	0
p_1	0	АИ	0	+p	0

Предусмотреть сохранение (т. е. запоминание и восстановление) содержимого индекс-регистра во время работы подпрограммы.

§ 6. Настройка модулей по месту

Организация подпрограмм (модулей), описанная в предыдущих разделах, выглядит довольно стройной. Но у нее есть существенный недостаток — каждый модуль рассчитан на то, что он занимает определенное место в памяти. Современные библиотеки стандартных подпрограмм (не считая подпрограмм, разрабатываемых лишь для включения в состав одной или немногих программ) насчитывают, как правило, сотни модулей. Их общий объем намного превышает объем оперативной памяти. Если даже учесть, что каждая программа использует во время своей работы лишь небольшую часть библиотеки и что использование некоторых комбинаций модулей в одной программе очень маловероятно, то и тогда оказывается практически невозможным выбрать для каждого модуля свое неизменное место в памяти, на которое не претендовал бы ни один из совместно работающих с ним модулей.

Отсюда возникает задача — научиться располагать модули в произвольном месте памяти. Такая постановка задачи еще слишком расплывчата. Будем для определенности считать, что модуль вместе со своими константами, заготовками и рабочими ячейками (такими, которые он не делит с другими модулями) составлен, исходя из предположения, что он занимает в памяти массив ячеек с начальным (наименьшим) адресом a_n и конечным (наибольшим) адресом a_k . Здесь a_n и a_k не обязательно обозначают адрес входа в модуль и выхода из него. Пусть этот модуль требуется фактически разместить, начиная с ячейки с адресом b_n . Ясно, что при этом любой адрес ячейки, входящей в состав модуля (такой адрес называется *внутренним адресом* модуля), должен получить приращение $d = b_n - a_n$. Как распознать такие адреса? Можно просмотреть все слова, из которых состоит модуль, прочесть их как коды команд, выделить по очереди каждый из трех адресов команды, и если этот адрес A удовлетворяет неравенству $a_n \leq A \leq a_k$, считать, что он относится к интересующей нас категории, и соответственно изменить его. В действительности при этом можно пропустить некоторые адреса, которые следовало бы откорректировать, и испортить некоторые коды, которые должны остаться неизменными. Например, может оказаться, что адрес $A < a_n$, но он снабжен признаком модификации и во время выполнения команды, содержащей этот адрес, значение индекс-регистра удовлетворяет неравенствам $a_n - A \leq I \leq a_k - A$ (иначе, $a_n \leq A' = A + I \leq a_k$). Такой адрес A может иметь отношение к нашему массиву и его следовало бы откорректировать. Еще более вероятен другой случай, например, когда среди слов модуля содержится код некоторой числовой константы и группа разрядов этого кода, занимающая место одного из адресов, изображает число из диапазона $a_n \div a_k$. Таким образом, модули, которые мы хотим перемещать из одного места памяти

в другое, не могут быть совершенно произвольными. Потребуем, чтобы все внутренние адреса модуля обязательно удовлетворяли неравенствам $a_n \leq A \leq a_k$ и $a_n \leq A' \leq a_k$. Потребуем также, чтобы коды команд и заготовок, содержащие такие адреса, располагались внутри модуля в ячейках с адресами от c_n до c_k (так что $a_n \leq c_n \leq c_k \leq a_k$). Удовлетворить этим требованиям при составлении модуля нетрудно. Адреса c_n и c_k — тоже внутренние. Если модуль фактически размещен, начиная с адреса b_n , то корректировке подлежат лишь коды команд (или заготовок), расположенные в ячейках с адресами от $c'_n = c_n + b_n - a_n$ до $c'_k = c_k + b_n - a_n$.

Итак, нам надо составить подпрограмму, которая просматривает все слова с адресами i из диапазона $c'_n \leq i \leq c'_k$, читает их как коды команд, исследует адреса этих команд и к каждому адресу A , удовлетворяющему неравенствам $a_n \leq A \leq a_k$, добавляет приращение d . Это подпрограмма представляет собой цикл, в котором i пробегает значения от c'_n до c'_k . При составлении рабочей части этого цикла будем считать, что код исследуемой команды (из ячейки с адресом i), имеющий вид

$r \quad \pi \quad \text{КОП} \quad A_1 \quad A_2 \quad A_3$

помещен (на время его обработки) в ячейку r , а в ячейке q размещается код

$q \quad \pi \quad \text{КОП} \quad A_j \quad A_{j+1} \quad A_{j+2}, \quad j \geq 1,$

где $A_s = 0$ при $s > 3$, а при $s \leq 3$ величина A_s в кодах r и q имеет одно и то же значение. Независимо от значения адресов A_{j+1} и A_{j+2} операция

$0 \quad -M \quad q \quad s_1 \quad 0$

где s_1 — адрес константы

$\sim \sim a_n \quad 0 \quad 0$

вырабатывает значение $\omega = 0$, если $A_j \geq a_n$, и $\omega = 1$, если $A_j < a_n$. Неравенство $A_j \leq a_k$ эквивалентно неравенству $A_j < a_k + 1$, поэтому для его проверки достаточно выполнить операцию

$0 \quad -M \quad q \quad s_2 \quad 0$

где в ячейке s_2 хранится константа

$s_2 \quad \sim \sim a_k + 1 \quad 0 \quad 0$

В зависимости от результата этих двух проверок надо выполнить или не выполнять операцию

$0 \quad +M \quad r \quad d+j \quad r$

где константы $d+1$ и $d+3$ имеют вид

$d+1 \quad \sim \sim +d \quad 0 \quad 0$

$d+2 \quad \sim \sim 0 \quad +d \quad 0$

$d+3 \quad \sim \sim 0 \quad 0 \quad +d$

а $d = b_n - a_n$.

Для перехода к следующему значению j адресную часть кода в ячейке q надо сдвинуть на 12_{10} разрядов влево. Если после сдвига адресная часть равна нулю ($\omega = 1$), то корректировка команд закончена. Это заведомо случится, если $j > 3$. Поэтому для выхода из цикла по j можно воспользоваться командой $0И \geq A$ с $A_1 = 0$. Итак, внутренний цикл подпрограммы может быть таким:

p_1	0	$-M$	q	s_1	0	Если $A_j < a_n$, то ($\omega := 1$)
	0	$П1$	0	p_2	0	перейти на p_2
	0	$-M$	q	s_2	0	Если $A_j > a_k$, то ($\omega := 0$)
	0	$П0$	0	p_2	0	перейти на p_2
	2	$+M$	r	d	r	$A_j := A_j + d$ (в ячейке r) ($И = j$)
p_2	0	$\leftarrow MA$	114	q	q	$j := j + 1$, если $A_j A_{j+1} A_{j+2} > 0$, то на p_1
p_3	1	$0И \geq A$	0	p_1	1	

Ранее рассмотренные примеры показывают, что вероятность встретить в настраиваемом модуле команду с нулевой адресной частью довольно высока. Поэтому желательно начать выполнение этого цикла с проверки, не равна ли нулю адресная часть кода команды, помещенной в ячейку r . Для этого во внешнем цикле следует перед командами $p_1 \div p_3$ поставить команды

p_0	2	$\leftarrow MA$	0100	0	r	$r := \text{«л КОП } A_1 A_2 A_3\text{»}$, если $A_1 A_2 A_3 > 0$, то $\omega := 0$ ($И = i$)
	4	$АИ$	0	0	p_4	Запоминание i , $И := 0$
	0	$ПБ$	r	p_3	q	$j := 1$, переход на p_3

Операции $АИ$ и $ПБ$ не меняют значения ω , поэтому значение ω , выработанное командой p_0 , будет использовано командой p_3 . Для завершения внешнего цикла и всей подпрограммы вслед за командой p_3 следует поместить команды

p_4	0	$АИ$	0	i	0	Восстановление i
	1	$ПЧ$	r	0	0	Засылка i -й команды модуля на свое место
p_5	1	$И < A$	c_k	p_0	1	$i := i + 1$, если $i \leq c_k$, то на p_0
	0	0	0	0	0	Восстановление $И$
p_k	0	0	0	0	0	Ячейка возврата

Осталось составить формирующую часть этой подпрограммы. Предположим, что при обращении к подпрограмме в ячейки a_1, a_2, b_1, c_1, c_2 были посланы коды

a_1	0	$АИ$	0	a_n	0
a_2	0	$АИ$	0	a_k	0
b_1	0	$АИ$	0	b_n	0
c_1	0	$АИ$	0	c_n	0
c_2	0	$АИ$	0	c_k	0

Формирующая часть должна сформировать команду p_5 , заготовки s_1 ,

$s_2, d + 1 \div d + 3$ и заслать в индекс-регистр начальное значение i , равное c'_n . Небольшое осложнение возникает при формировании константы $d + 3$ при $b_n < a_n$, т. е. при $d < 0$. Было бы ошибкой попытаться получить ее из константы $d + 2$, сдвигая адресную часть на 12 разрядов вправо. Адресные части этих констант изображаются в дополнительном коде, так что, например, при $a_n = 2000, b_n = 0531$ константы должны иметь вид

$d+1$	~	~	6531	0000	0000
$d+2$	~	~	7777	6531	0000
$d+3$	~	~	7777	7777	6531

Сдвигая вправо константу $d + 2$, мы получили бы неверный код

~ ~ 0000 7777 6531

Но при любом знаке разности $b_n - a_n$ константы $d + 2$ и $d + 1$ можно получить, сдвигая адресную часть константы $d + 3$ влево. Поэтому начинать следует с формирования константы $d + 3$. Формирующая часть может быть такой:

p_n	0	←MA	64	a_1	$d+3$	«0	AI	0	0	a_n »
	0	←MA	64	b_1	b_1	«0	AI	0	0	b_n »
	0	—M	b_1	$d+3$	$d+3$	«0	AI	0	0	+ d »
	0	←MA	114	$d+3$	$d+2$	«0	AI	0	+ d	0»
	0	←MA	114	$d+2$	$d+1$	«0	AI	+ d	0	0»
	0	←MA	114	a_1	s_1	«0	AI	a_n	0	0»
	0	←MA	114	a_2	s_2	«0	AI	a_n	0	0»
	0	+M	s_2	z_1	s_2	«0	AI	$a_n + 1$	0	0»
	0	+M	c_2	$d+2$	c_2	«0	AI	0	c'_n	0»
	0	←MA	114	c_2	c_2	«0	AI	c'_n	0	0»
	0	+M	z_2	c_2	p_5	«1	$H < A$	c'_n	p	1»
	0	+M	c_1	$d+2$	c_1	«0	AI	0	c'_n	0»
	0	KI	0	c_1	$p_n - 1$	Запоминание $H, i := c'_n$				

с константами

z_1	0	0	1	0	0
z_2	1	$H < A$	0	p	1

У п р а ж н е н и е 1. Предполагая, что ячейки a_1, a_2, b_1, c_1, c_2 расположены в памяти подряд (но в произвольном порядке), составить формирующую часть так, чтобы можно было совместить ячейки $s_1, s_2, d + 1 \div d + 3$ с этими ячейками. Число команд формирующей части не должно возрасти.

ЗАКЛЮЧЕНИЕ

В этой книге изложены лишь самые первоначальные, простейшие сведения о программировании на ЭВМ. За ее пределами остались такие вопросы, как структура современных ЭВМ, состав, назначение и принципы работы их математического обеспечения, методы хранения в памяти больших массивов данных и организации быстрого поиска нужной информации в этих массивах, возможности современных языков программирования и методы автоматического перевода с этих языков на машинный язык и многое другое, без чего эффективное решение сложных задач оказывается невозможным. Однако, каким бы из этих вопросов ни пришлось заниматься программисту, он неизменно будет сталкиваться с теми простейшими понятиями, о которых шла речь в этой книге. Вспомним важнейшие из них.

1. Одни устройства вычислительной машины ответственны за хранение данных, другие — за обработку данных. В то же время полностью разграничить хранение данных от их обработки невозможно. Мы имели дело почти исключительно с простейшей системой хранения данных — в оперативной памяти машины со сквозной нумерацией ячеек. Но и при этом мы столкнулись с необходимостью вычисления адресов при обращении к памяти в случаях, когда данные объединялись в массивы. Вычисление адресов как раз и является обработкой данных, которую необходимо выполнять в связи с хранением данных. Другие примеры — подсчет контрольных сумм при операциях ввода, вывода и обмена с внешней памятью, сравнение номера очередной зоны на ленте с заданным номером зоны при обмене между лентой и оперативной памятью. Чем сложнее система хранения данных, чем больше ее возможности, тем больше операций приходится выполнять при посылке данных на хранение и при их извлечении, тем сложнее эти операции. В современных машинах для выполнения подобных операций существуют специальные процессоры, работающие параллельно с центральным процессором, который выполняет основные операции по переработке данных, предписанные программой. В свою очередь, выполнение этих операций невозможно без запоминания, хотя бы кратковременного, некоторых данных — адреса выполняемой команды,

кода операции, значений операндов и т. п. В машинах семейства М-20 для этого служат регистры центрального устройства, в частности, индекс-регистр. В других машинах количество таких вспомогательных регистров центрального процессора во много раз больше. Часто создаются специальные запоминающие устройства, называемые буферными, целиком находящиеся в ведении центрального процессора. Занесение данных в буферную память и выборка их оттуда не предписываются программистом явно в составленной им программе, а планируются и осуществляются самим центральным процессором, в соответствии с потребностями выполняемой им в данный момент операции и с историей работы программы. Буферные запоминающие устройства допускают более быстрое обращение, чем оперативная память, благодаря чему выполнение программы значительно ускоряется.

2. Существует вполне определенная связь между адресом ячейки и ее содержимым (машинным словом). В то же время эти понятия четко различаются. Недостаточное понимание этой связи и этого различия — главная беда начинающего программиста. Отношение между адресом и содержимым ячейки имеет множество аналогий. Ближайшая из них — это отношение между обозначением переменной (в АЛГОЛе или ином языке программирования) и ее текущим значением. Более отдаленная — отношение между словом естественного языка и обозначаемым им предметом или понятием. Некоторым утешением для программистов, продолжающих испытывать трудности в уяснении этого и подобных ему отношений, может служить то, что оно продолжает оставаться предметом внимательного изучения и непрекращающихся дискуссий для многих математиков, логиков, филологов и философов. Наградой программисту, твердо усвоившему характер связи между адресом ячейки и ее содержимым, послужит легкость, с которой он сможет воспринимать гораздо более тонкие свойства аналогичных понятий в теории алгоритмических языков, в логике и других разделах математики.

3. В машинной команде адрес выступает в роли наименования (обозначения) операнта или результата выполнения этой команды. С другой стороны, сам адрес может быть задан тем или иным способом — представлен восьмеричным числом, обозначен идентификатором (при записи команды в символических обозначениях), получен как значение исполнительного адреса или вычислен как значение некоторого выражения. Всеми этими способами представления адресов мы неоднократно пользовались в книге. Таким образом, адрес может поочередно то обозначать объекты вычисления (при исполнении команды), то быть таким объектом (при формировании команды, при образовании исполнительного адреса, при переводе символических обозначений в восьмеричные). Такой двойственной природой обладает не только отдельный адрес, но и вся команда, участки программ и программы в целом. Это открывает

путь к самым разнообразным возможностям формирования программ в целом и по частям. В этих возможностях — сила ЭВМ, но в них же кроется и источник многочисленных трудностей в программировании. В самом деле, если программа выполняется совсем не в том виде, в каком ее составил программист, если предвзвешенно она подвергается многократным разнообразным преобразованиям, то трудно быть уверенным, что окончательный результат работы программы будет именно тем, какого хотел ее автор.

4. Уже при рассмотрении простейших задач — в гл. 2, когда речь шла о переводе чисел из одной системы счисления в другую, в гл. 4 в связи с решением уравнения $ax^2 + bx + c = 0$ — обращалось внимание на важность обработки текстовой или символьной информации. Роль такой информации постоянно возрастает по мере расширения круга решаемых задач, обогащения средств их описания. Так, если мы хотим избавить человека от необходимости пользоваться в программах числовыми адресами, то необходимо оснастить машину программой, способной переводить команды из символических в восьмеричные обозначения (см. § 2 гл. 15). Гораздо более сложной является задача перевода с АЛГОЛа на машинный язык. Такие задачи постоянно возникают в практике программирования. От программирования вычислений по формулам происходит постепенный переход к автоматическому получению самих формул. Как заметил один известный программист, не за горами время, когда машина, получив задание обратиться матрицу, будет в ответ печатать разъяснение, почему в данной задаче можно обойтись без обращения матриц. Пусть это только шутка, но программисты уже давно задумываются над тем, как научить машину анализировать смысловое содержание поставленной перед ней задачи, с тем чтобы она самостоятельно находила способ решения задачи. Ясно, что при этом языки, подобные АЛГОЛу, становятся мало пригодными для формулировки задания машине. Шагом вперед являются языки, целиком ориентированные на обработку символьной информации, или дополненные средствами описания процессов такой обработки. В перспективе стоит задача научить машину понимать естественные языки.

5. При решении задачи главная трудность заключается вовсе не в том, чтобы написать программу. Главное — это выработать систему понятий, которая четко отражает существо решаемой задачи и открывает естественный путь к составлению программы. У нас было мало возможностей показать, как, развивая систему понятий, связанных с решаемой задачей, можно прийти к более простой и эффективной программе ее решения. Нашей целью было изучение важнейших методов программирования, поэтому примеры задач выбирались из тех областей, где круг основных понятий давно сложился и должен быть известен читателю. Отметим все же задачу о транспонировании матрицы, где идея о возможности

выборки переставляемых элементов матрицы не только вдоль строк или столбцов, но и вдоль диагоналей позволила заметно упростить машинную программу (см. упражнения 9.2.2 и 10.2.3.). Введение новых понятий позволило прийти к быстро работающей программе перевода чисел из десятичной системы в двоичную (программа 14.5.3). Общие принципы подхода к решению задач на машине были продемонстрированы также в гл. 4 и § 11 гл. 8.

Формирование в той или иной области знания понятий, облегчающих решение на машинах возникающих в ней задач, относится, конечно, к самой этой области, а не к программированию. Этим должны заниматься, как правило, специалисты в этой области. Важно, однако, чтобы эти специалисты хорошо знали возможности и особенности машин, их сильные и слабые стороны, знали не только на уровне понятий, вошедших в такие языки, как АЛГОЛ, но и значительно глубже. Нельзя сознательно овладеть АЛГОЛОм или иным языком программирования, не зная, на базе каких понятий возникли и развивались эти языки, не представляя себе, во что превращаются программы, написанные на этих языках, после трансляции. В еще большей степени это относится к пользованию другими элементами математического обеспечения ЭВМ. Наоборот, хорошее владение основными понятиями программирования позволяет вести проектирование алгоритмов систематично и целенаправленно, последовательно продвигаясь от общей постановки задачи через серию промежуточных этапов к такому уровню частных задач, что способ их решения на машине становится очевидным.

Составление программ во многом родственно проектированию машин. Но если конструктор в известной степени вправе надеяться, что его ошибки и недоработки в чертежах будут обнаружены и исправлены расчетчиками, технологами, даже рабочими, воплощающими его конструкцию в металле, то программист обязан знать, что малейшая его погрешность либо (в лучшем случае) сделает выполнение программы невозможным, либо будет воспроизведена машиной и приведет к неверным результатам работы программы. Поэтому программист обязан быть предельно логичным, методичным, аккуратным и предусмотрительным. Он должен выработать у себя стиль работы, отвечающий этим требованиям. Однако его задача будет тем легче, чем большим числом приемов программирования он владеет, чем больше ситуаций, которые могут возникнуть в ходе составления программы, окажутся для него привычными.

Приемы программирования, разработанные в книге, не охватывают всех ситуаций, с которыми может встретиться программист. Эти приемы, продемонстрированные на машине с ограниченными возможностями, не могут быть механически использованы на других, более современных машинах. Но идеи, на которых основаны эти приемы, остаются в силе, и усвоение их позволит программисту быстро овладеть техникой программирования на любой машине.

РЕШЕНИЯ УПРАЖНЕНИЙ

4.3.1.

Q''. Даны коэффициенты a , b и c уравнения $ax^2 + bx + c = 0$. Требуется определить тип и число корней уравнения, присвоив соответствующее значение текстовой переменной s , и значения корней, присвоив их переменным b и c в случае вещественных различных корней, переменной b — в случае одного вещественного корня (простого или кратного), наконец, в случае комплексных корней присвоив переменной b значение вещественной части корней, а переменной c — мнимой части.

Q''1. Если $a = 0$, то перейти на Q''5, в противном случае $b := -b/2a$, $a := b^2 - c/a$.

Q''2. Если $a > 0$, то $s := \langle 2в \rangle$, $a := \sqrt{a}$, $c := b - a$, $b := b + a$, выход.

Q''3. Если $a < 0$, то $s := \langle 2к \rangle$, $c := \sqrt{-a}$, выход.

Q''4. $s := \langle 2р \rangle$, выход.

Q''5. Если $b \neq 0$, то $s := \langle 1 \rangle$, $b := -c/b$, выход.

Q''6. Если $c \neq 0$, то $s := \langle 0 \rangle$, выход.

Q''7. $s := \langle \infty \rangle$.

4.3.2.

A. Найти неотрицательное число x , представленное его записью z в системе счисления с основанием q . Предполагается, что запись z содержит по крайней мере одну цифру, если число x — целое, и не менее одной цифры как до, так и после запятой, если x — не целое.

A1. $x := n(\lfloor z \rfloor)$, $z := \lfloor z$, $t := 0$.

A2. Если $z = \langle \rangle$, то перейти на A6.

A3. $w := \lfloor z$, $z := \lfloor z$.

A4. Если $w = \langle \rangle$, то $t := 1$, перейти на A3.

A5. $x := qx + n(w)$, $t := qt$, перейти на A2.

A6. Если $t > 0$, то $x := x/t$.

4.3.3.

B. Найти запись z неотрицательного числа x с s знаками после запятой в системе счисления с основанием q .

B1. Если $s > 0$, то $x := xq^s$.

B2. Округлить x до ближайшего целого, $z := \langle \rangle$.

B3. Найти частное u и остаток v от деления x на q , $z := d(v)^{\wedge} z$, $x := u$, $s := s - 1$.

B4. Если $s = 0$, то $z := \langle \rangle^{\wedge} z$, перейти на B3.

B5. Если $x \neq 0$, то перейти на B3.

4.4.1.

R. Вычисление остатка r от деления $m \geq 0$ на $n > 0$.

R1. $r := m$.

R2. Пока $r \geq n$ выполнять $r := r - n$.

4.4.2.

S. Вычисление суммы $s = \sum_{i=1}^n f(i)$.

S1. $i := 0, s := 0$.

S2. Пока $i < n$ выполнять $i := i + 1, s := s + f(i)$.

4.4.3.

НОД (m, n) = если $n = 0$, то m , в противном случае НОД (n , ОСТ (m, n)).
ОСТ (m, n) = если $m < n$, то m , в противном случае ОСТ ($m - n, n$).

4.5.1.

R'. Найти остаток r от деления целого m на целое $n > 0$.

R'1. $k := 0$.

R'2. Если $m \geq n$, то $n := 2n, k := k + 1$, перейти на R'2.

R'3. Если $k = 0$, то $r := m$, выход.

R'4. $n := n/2, k := k - 1$.

R'5. Если $m \geq n$, то $m := m - n$.

R'6. Перейти на R'3.

4.5.2.

E'. Даны два целых числа m и $n, m > n \geq 0$. Найти их наибольший общий делитель d .

E'1. $d := m$.

E'2. Если $n = 0$, то выход.

E'3. Если $d < n$, то $m := n, n := d$, перейти на E'1.

E'4. $d := d - n$, перейти на E'3.

4.6.1.

S'. Дан многочлен M . Построить многочлен M_1 , тождественно равный многочлену M , но не содержащий подобных членов.

S'1. $M_1 :=$ пустой многочлен.

S'2. $Z :=$ начальный знак $M, O :=$ первый одночлен $M, M :=$ остаток M .

S'3. Если $Z = «+»$, то перейти на S'5.

S'4. $K :=$ коэффициент $O, T :=$ терм $O, K := -K, O := K \wedge T$.

S'5. $M_1 := M_1 \wedge «+» \wedge O$.

S'6. Если M не пуст, то перейти на S'2.

S'7. $M := M_1, M_1 :=$ пустой многочлен.

S'8. $O :=$ первый одночлен $M, M :=$ остаток $M, K :=$ коэффициент O .
 $T :=$ терм $O, M_2 :=$ пустой многочлен.

S'9. Если M пуст, то $M := M_2$, перейти на S'13.

S'10. $O_1 :=$ первый одночлен $M, M :=$ остаток M .

S'11. Если терм $O_1 = T$, то $K := K +$ коэффициент O_1 , в противном случае $M_2 := M_2 \wedge O_1$.

S'12. Перейти на S'9.

S'13. Если $K = 0$, то перейти на S'15.

S'14. $M_1 := M_1 \wedge «+» \wedge K \wedge T$.

S'15. Если M не пуст, то перейти на S'8.

4.6.2.

В а р и а н т 1.

R'1. Если T_1 пуст, то перейти на R'3.

R'2. $D :=$ первый множитель $T_1, T_1 :=$ остаток T_1 , изменить знак показателя D на обратный, $T := T \wedge D$, перейти на R'1.

R'3. Если T пуст, то ответ := «да», выход.

R'4. $D :=$ первый множитель $T, T :=$ остаток $T, V :=$ переменная D ,
 $E :=$ показатель $D, T_1 :=$ пустой терм, перейти на R'7.

R'5. $D :=$ первый множитель $T, T :=$ остаток T .

R'6. Если переменная $D = V$, то $E := E + \text{показатель } D$, в противном случае $T_1 := T_1 \setminus D$.

R'7. Если T не пуст, то перейти на R'5.

R'8. Если $E = 0$, то $T := T_1$, перейти на R'3.

R'9. Ответ := «нет».

В а р и а н т 2. Дополнить алгоритм R новым начальным шагом:

R0. Привести подобные множители в терме T , результат обозначить снова через T , привести подобные множители в терме T_1 , результат обозначить T_1 .

Алгоритм приведения подобных множителей в терме вполне аналогичен описанному выше алгоритму приведения подобных членов в многочлене. Его можно получить из алгоритма S' , заменив слова «многочлен», «одночлен», «терм» и «коэффициент» соответственно словами «терм», «множитель», «переменная» и «показатель». С помощью преобразованного алгоритма можно детализировать шаг R0.

5.1.1. 2^{-100} (BK: 000 0000 0000 0001).

5.1.2. $2^{63}(1 - 2^{-36}) = 2^{63} - 2^{27}$ (BK: 177 7777 7777 7777).

5.1.3. $-2^5 \left(\frac{4}{8} + \frac{2}{8^2} \right) = -17$.

5.1.4. 076 4444 4444 4445 (с округлением).

7.3.1.

p	0	ПЧ	0	0	s	$s := 0$
$p+1$	0	ПЧ	$\langle 0,01 \rangle$	0	x	$x := 0,01$
$p+2$	0	$\sqrt{}$	x	0	r	$\sqrt{x}$
$p+3$	0	+	s	r	s	$s := s + \sqrt{x}$
$p+4$	0	+	x	$\langle 0,01 \rangle$	x	$x := x + 0,01$
$p+5$	0	-	$\langle 1,005 \rangle$	x	0	Если $x \leq 1,005$, то
$p+6$	0	П0	0	$p+2$	0	перейти на $p+2$

7.5.1. Алгоритм:

A. Вычисление $s = \arctg x$ с точностью до ϵ .

A1. $k := 1$, $u := x$, $s := x$, $v := -x^2$.

A2. $k := k + 2$, $u := uv$, $s := s + u/k$.

A3. Если $|u/k| > \epsilon$, то на A2.

Машинная программа:

A1	0	ПЧ	$\langle 1 \rangle$	0	k	$k := 1$
	0	ПЧ	x	0	u	$u := x$
	0	ПЧ	x	0	s	$s := x$
	0	$\times$	x	x	v	x^2
A2	0	-	0	v	v	$v := -x^2$
	0	+	k	$\langle 2 \rangle$	k	$k := k + 2$
	0	$\times$	u	v	u	$u := uv$
	0	/	u	k	r	u/k
	0	+	s	r	s	$s := s + u/k$
A3	0	-	ϵ	r	0	Если $ u/k > \epsilon$, то
	0	П1	0	A2	0	перейти на A2

7.5.2. Алгоритм:

E. Вычисление $y = e^x$.

E1. $i := 0$, $u := 1$, $s := 1$, $y := 1$.

E2. Если $x > 0,5$, то $x := x - 1$, $y := y \times e$, перейти на E2.

Е3. Если $x < -0,5$, то $x := x + 1$, $y := y/e$, перейти на Е3.

Е4. $i := i + 1$, $u := ux/i$, $s := s + u$.

Е5. Если $|u| > e$, то перейти на Е4.

Е6. $y := y \times s$.

Машинная программа:

Е1	0	ПЧ	0	0	i	$i := 0$
	0	ПЧ	$\langle 1 \rangle$	0	u	$u := 1$
	0	ПЧ	$\langle 1 \rangle$	0	s	$s := 1$
	0	ПЧ	$\langle 1 \rangle$	0	y	$y := 1$
Е2	0	—	$\langle 0,5 \rangle$	x	0	Если $x \leq 0,5$, то
	0	ПО	0	Е3	0	перейти на Е3
	0	—	x	$\langle 1 \rangle$	x	$x := x - 1$
	0	$\times$	y	$\langle e \rangle$	y	$y := ye$
	0	ПБ	0	Е2	0	Перейти на Е2
Е3	0	+	x	$\langle 0,5 \rangle$	0	Если $x \geq -0,5$, то
	0	ПО	0	Е4	0	перейти на Е4
	0	+	x	$\langle 1 \rangle$	x	$x := x + 1$
	0	/	y	$\langle e \rangle$	y	$y := y/e$
	0	ПБ	0	Е3	0	Перейти на Е3
Е4	0	+	i	$\langle 1 \rangle$	i	$i := i + 1$
	0	$\times$	u	x	u	ux
	0	/	u	i	u	$u := ux/i$
	0	+	s	u	s	$s := s + u$
Е5	0	$ - $	e	u	0	Если $ u > e$, то
	0	П1	0	Е4	0	перейти на Е4
Е6	0	$\times$	y	s	y	$y := ys$

7.5.3. Обозначим $b_i = i^2 + 1$, $c_i = (-1)^i (2i)!$, $d_i = b_i - b_{i-1} = 2i - 1$, $e_i = c_i/c_{i-1} = -2i(2i-1)$, $f_i = e_{i-1} - e_i = 8i - 6$.

Алгоритм (начальные значения d и f шаг Q1 вырабатывает для $i = 1$, а не для $i = 0$).

Q. Вычислить значение полинома

$$q = \sum_{i=0}^n a_i x^{n-i}, \quad a_i = (-1)^i \frac{i^2 + 1}{(2i)!} \quad \text{при } n > 0.$$

Q1. $q := 1$, $b := 1$, $c := 1$, $d := 1$, $e := 0$, $f := 2$.

Q2. $b := b + d$, $d := d + 2$, $e := e - f$, $f := f + 8$, $c := ce$, $q := qx + b/c$.

Q3. Если $d < 2n$, то перейти на Q2.

Программа:

Q1	0	ПЧ	$\langle 1 \rangle$	0	q	$q := 1$
	0	ПЧ	$\langle 1 \rangle$	0	b	$b := 1$
	0	ПЧ	$\langle 1 \rangle$	0	c	$c := 1$
	0	ПЧ	$\langle 1 \rangle$	0	d	$d := 1$
	0	ПЧ	0	0	e	$e := 0$
	0	ПЧ	$\langle 2 \rangle$	0	f	$f := 2$
	0	+	n	n	N	$N := 2n$

Q2	0	+	b	d	b	$b := b + d$
	0	+	d	<2>	d	$d := d + 2$
	0	-	e	f	e	$e := e - f$
	0	+	f	<8>	f	$f := f + 8$
	0	×	c	e	c	$c := ce$
	0	/	b	c	a	$a := b/c$
	0	×	q	x	q	qx
Q3	0	+	q	a	q	$q := qx + a$
	0	-	d	N	0	Если $d < N$, то
	0	П1	0	Q2	0	перейти на Q2

7.7.1. Программа вычисляет в ячейке 0001 значение выражения

$$\sum_{i=1}^{32} a_i b_i$$

(проще — скалярное произведение векторов a и b), где a_i и b_i — числа, хранящиеся в ячейках 4001, ..., 4040 и 4041, ..., 4100 соответственно.

7.7.2.

C1	0	ПЧ	z	0	p	} $t := 0$
	0	ПЧ	z ₁	0	p+2	
C2	0	+M	p	z ₃	p	} $i := i + 1$
	0	+M	p+2	z ₃	p+2	
p	0	ПЧ	x+i	0	u	$u := x_i$
	0	ПВ	p+2	f _n	f _k	Переход на вычисление $f(u)$ с возвратом
C3	0	ПЧ	v	0	y+i	$y_i := v$
	0	≠	p	z ₄	0	Если $i \neq n$, то
	0	П0	0	C2	0	перейти на C2

Константы:

z	0	ПЧ	x	0	u
z ₁	0	ПЧ	v	0	y
z ₂	0	0	1	0	0
z ₃	0	0	0	0	1
z ₄	0	ПЧ	x+n	0	u

7.7.3. Константа переадресации не меняется, константа формирования для $i = 1$:

z_ф 0 КОП $a_1 + b_1$ $a_2 + b_2$ $a_3 + b_3$

константа сравнения для $i = n + 1$

z_c 0 КОП $a_1 + b_1(n+1)$ $a_2 + b_2(n+1)$ $a_3 + b_3(n+1)$

7.7.4. Все константы имеют тот же вид, что для алгоритма L.

7.7.5.

L'''. Алгоритм простого структурного цикла.

L'''1. Присвоить начальные значения переменным из X, сформировать переменные команды для $i = 1$.

L'''2. P(i).

L'''3. Если $i = n$, то выход.

L'''4. Переадресация для $i := i + 1$, перейти на L'''2.

7.7.6. Программа:

A1	0	ПЧ	0	0	s	$s := 0$
	0	ПЧ	z_Φ	0	p	Формирование для $i := 1$
A2	0	$\neq$	p	z_c	0	Если $i = n + 1$, то
	0	П1	0	$p + 3$	0	выход
p	0	+	s	$x + i$	s	$s := s + x_i$
	0	+ M	p	z_n	p	$i := i + 1$
	0	ПБ	0	A2	0	Перейти на A2

Константы:

z_Φ	0	+	s	$x + 1$	s
z_c	0	+	s	$x + n + 1$	s
z_n	0	0	0	1	0

Более экономный вариант программы:

A1	0	ПЧ	0	0	s	$s := 0$
	0	ПБ	z_Φ	A2	p	$i := 1$, перейти на A2
p	0	+	s	$x + i$	s	$s := s + x_i$
	0	+ M	p	z_n	p	$i := i + 1$
A2	0	$\neq$	p	z_c	0	Если $i \neq n + 1$, то
	0	П0	0	p	0	перейти на p

Константы остаются прежними.

7.7.7. Формирование команд в ячейках $14 + 4$ и $14 + 2$ и переадресация команды $14 + 4$ выполняются так же, как в программе 7.7.7. К моменту выполнения команды $12 + 1$ в ячейке $14 + 4$ находится уже сформированная для текущего i команда (в том виде, в каком она написана в программе). Легко видеть, что операция $+M$, выполняемая по команде $12 + 1$, зашлет в ячейку $14 - 2$ нужное значение команды.

К моменту выполнения команды $12 + 2$ ячейка $14 + 2$ содержит команду, сформированную для предыдущего значения i , т. е. команду

$$0 - a + 2i \quad a + 2i - 2 \quad r$$

В результате выполнения операции $+M$ над этим кодом и кодом z_5 в ячейку 14 зашлет код

$$0 - a + 2i + 1 \quad a + 2i - 1 \quad r$$

следовательно, содержимое ячейки 14 будет правильно сформировано для текущего i . После этого команда $12 + 3$ сформирует в ячейке $14 + 2$ код, также соответствующий текущему значению i . Команды $14 - 2$, 14 , $14 + 2$ и $14 + 4$ несколько отличаются от соответствующих команд программы 7.7.7, но видно, что в ячейку y они засылают то же числовое значение.

9.1.1. $a + in - j + 1$, $a + (m - i)n + j$, $a + (m - i + 1)n - j + 1$, $a + jm - i + 1$, $a + (n - j)m + i$, $a + (n - j + 1)m - i + 1$.

9.1.2. $a + d_1(i_1 - l_1) + \dots + d_k(i_k - l_k)$, где $d_1 = 1$, $d_j = d_{j-1}w_{j-1}$ для $j = 2, \dots, k$; w_j имеет прежний смысл.

9.1.3. $a + 2i + j$.

9.1.4. $a + \frac{(2m - i + 2)(i - 1)}{2} + j - i$.

9.1.5. $a + ni^2 - \frac{1}{3}(i-1)i(2i-1) + c + k$, где

$$c = \begin{cases} i(j+1) & \text{при } j \leq i \text{ и } j \leq n-i, \\ (i+2j-n)(n-i+1) & \text{при } n-i < j \leq i, \\ i(2j-i+1) & \text{при } i < j \leq n-i, \\ 2i(n-i+1) - (n-j)(n-j+1) & \text{при } i < j \text{ и } n-i < j. \end{cases}$$

9.2.1.

	0	ПЧ	z_5	0	p	} $i := 1, j := 1$
	0	ПЧ	z_6	0	$p+1$	
	0	ПЧ	z_7	0	$p+2$	
	0	ПЧ	z_8	0	r_2	
	0	-M	z_2	z_6	r_3	} $i := 1$
	0	+M	z_1	z_3	r_4	
	0	ПЧ	z_8	0	r_5	
	0	+M	p	r_3	p	
p_1	0	+M	$p+1$	r_4	$p+1$	} $j := j-i+1$
	0	+M	$p+2$	r_5	$p+2$	
	0	+M	r_2	z_4	r_2	} $i := i+1$
	0	-M	r_3	z_1	r_3	
	0	-M	r_4	z_2	r_4	
	0	-M	r_5	z_3	r_5	
p	0	ПЧ	$a+(i-1)n+i$	0	r_1	$r_1 := a_{ij}$
	0	ПЧ	$a+(j-1)n+i$	0	$a+(i-1)n+j$	$a_{ij} := a_{ji}$
	0	ПЧ	r_1	0	$a+(j-1)n+i$	$a_{ji} := r_1$
	0	+M	p	z_1	p	} $j := j+1$
	0	+M	$p+1$	z_2	$p+1$	
	0	+M	$p+2$	z_3	$p+2$	
	0	$\neq$	p	r_2	0	Если $j \neq i$, то
	0	П0	0	p	0	
	0	$\neq$	r_3	z_1	0	Если $i \neq n$, то
	0	П0	0	p_1	0	

Переменные заготовки:

r_2	0	ПЧ	$a+(i-1)n+i$	0	r_1
r_3	0	0	$n-i+1$	0	0
r_4	0	0	$-n(i-1)+1$	0	$n-i+1$
r_5	0	0	0	0	$-n(i-1)+1$

Константы:

z_1	0	0	1	0	0
z_2	0	0	$+n$	0	1
z_3	0	0	0	0	$+n$
z_4	0	0	$n+1$	0	0
z_5	0	ПЧ	$a+1$	0	r_1
z_6	0	ПЧ	$a+1$	0	$a+1$
z_7	0	ПЧ	r_1	0	$a+1$
z_8	0	0	0	0	1

9.2.2. Оператор цикла на АЛГОЛе:

```

for k := 1 step 1 until n-1 do
  for i := 1 step 1 until n-k do
    begin r := a[i, i+k]; a[i, i+k] := a[i+k, i];
          a[i+k, i] := r end

```

Пусть a означает адрес $a[1,1]$. Тогда программа может быть такой:

	0	ПЧ	z_1	0	r_1	} $k := 0$
	0	ПЧ	z_2	0	r_2	
	0	ПЧ	z_3	0	r_3	
	0	ПЧ	z_4	0	r_4	
p_1	0	+M	r_1	z_5	r_1	} $k := k+1$
	0	+M	r_2	z_6	r_2	
	0	+M	r_3	z_7	r_3	
	0	ПЧ	r_1	0	p	} $t := 1$
	0	ПЧ	r_2	0	$p+1$	
	0	ПЧ	r_3	0	$p+2$	
p	0	ПЧ	$a+(i-1)n+$ $+i+k-1$	0	r	$r :=$ $a[i, i+k]$
	0	ПЧ	$a+(i+k-1) \times$ $\times n+i-1$	0	$a+(i-1)n+$ $+i+k-1$	$a[i, i+k]$ $:= a[i+k, i]$
	0	ПЧ	r	0	$a+(i+k-1)n+$ $+i-1$	$a[i+k, i]$ $:= r$
	0	+M	p	z_8	p	} $i := i+1$
	0	+M	$p+1$	z_9	$p+1$	
	0	+M	$p+2$	z_{10}	$p+2$	
	0	$\neq$	p	r_4	0	Если $t \neq n-k+1$,
	0	ПО	0	p	0	
	0	-M	r_4	z_{11}	r_4	то перейти на p
	0	$\neq$	r_4	z_{12}	0	$k := k+1$
	0	ПО	0	p_1	0	Если $k \neq n$, то перейти на p_1

где переменные заготовки имеют вид:

r_1	0	ПЧ	$a+k$	0	r
r_2	0	ПЧ	$a+kn$	0	$a+k$
r_3	0	ПЧ	r	0	$a+kn$
r_4	0	ПЧ	$a+(n-k+1)n$	0	r

Константы, не зависящие от параметров циклов:

z_1	0	ПЧ	a	0	r
z_2	0	ПЧ	a	0	a
z_3	0	ПЧ	r	0	a
z_4	0	ПЧ	$a+n^2$	0	r
z_5	0	0	1	0	0
z_6	0	0	$+n$	0	1
z_7	0	0	0	0	$+n$

z_8	0	0	$n+1$	0	0
z_9	0	0	$n+1$	0	$n+1$
z_{10}	0	0	0	0	$n+1$
z_{11}	0	0	$+n$	0	0
z_{12}	0	ПЧ	$a+n$	0	r

9.2.3. Из таблички видно, что нужно провести вычисления для $j = 1, 2, \dots, n$, а при каждом j для $i = 0, 1, \dots, n - j$. Очевиден также порядок вычисления — снизу вверх (от $i = n - j$ до $i = 0$), так как иначе вычисленную разность потребовалось бы засылать в ячейку, где хранится значение функции или ранее вычисленной разности, нужное для последующих вычислений. С учетом этих замечаний запишем алгоритм вычисления на АЛГОЛе:

```

for j := 1 step 1 until n do
  for i := n - j step -1 until 0 do
    f[i+j] := (f[i+j] - f[i+j-1]) / (x[i+j] - x[i])

```

Выпишем соответствующую ему программу:

	0	ПЧ	z_8	0	r_2	} Формирование для $j := 0$
	0	ПЧ	z_9	0	r_3	
p_1	0	-M	r_2	z_5	r_2	} $j := j + 1$
	0	+M	r_3	z_6	r_3	
p	0	ПЧ	z_3	0	p	} $i := n - j$
	0	ПЧ	r_2	0	$p + 1$	
	0	ПЧ	z_4	0	$p + 2$	
	0	-	$f + j + i$	$f + j + i - 1$	r	$r := f[i+j] - f[i+j-1]$
	0	-	$x + j + i$	$x + i$	r_1	$r_1 := x[i+j] - x[i]$
	0	/	r	r_1	$f + i + j$	$f[i+j] := r/r_1$
	0	-M	p	z_1	p	} $i := i - 1$
	0	-M	$p + 1$	z_1	$p + 1$	
	0	-M	$p + 2$	z_2	$p + 2$	
	0	$\neq$	$p + 1$	r_3	0	Если $i \neq -1$, то
	0	ИТО	0	p	0	перейти на p
	0	$\neq$	r_2	z_7	0	Если $j \neq n$, то
	0	ИТО	0	p_1	0	перейти на p_1

Общий вид переменных команд:

r_2	0	-	$x + n$	$x + n - j$	r_1
r_3	0	-	$x + j - 1$	$x - 1$	r_1

Константы:

z_1	0	0	1	1	0
z_2	0	0	0	0	1
z_3	0	-	$f + n$	$f + n - 1$	r
z_4	0	/	r	r_1	$f + n$
z_5	0	0	0	1	0
z_6	0	0	1	0	0
z_7	0	-	$x + n$	x	r_1
z_8	0	-	$x + n$	$x + n$	r_1
z_9	0	-	$x - 1$	$x - 1$	r_1

10.2.1. Так же, как в примере 7.7.5, примем, что адрес x_i равен $a + 2i + 2$, адрес y_i равен $a + 2i + 1$. Считаем, что x заведомо удовлетворяет неравенству $x_0 \leq x \leq x_n$. Программа на АЛГОЛе (ее операторная часть) имеет вид:

```

for i := 1 step 1 until n do
begin r := a [2 × i + 2] - x;
  if r ≥ 0 then
begin y := a [2 × i + 1] + (a [2 × i - 1] - a [2 × i + 1])
  × r / (a [2 × i + 2] - a [2 × i]);
  go to L end
end; L :

```

Машинная программа:

	0	AI	0	a	0	H := a
p	4	—	4	x	r	r := x _i - x (H = a + 2i - 2)
	1	1H ≥ A	0	p	2	H := H + 2; если r < 0, то на p
	6	—	7777	1	y	y _{i-1} - y _i (H = a + 2i)
	0	×	y	r	y	(y _{i-1} - y _i) × r
	6	—	2	0	r	x _i - x _{i-1}
	0	/	y	r	y	(y _{i-1} - y _i) × r / (x _i - x _{i-1})
	4	+	1	y	y	y := y _i + (y _{i-1} - y _i) × r / (x _i - x _{i-1})

Здесь в операции $1H \geq A$ условие $H \geq A$ всегда выполняется, переход по адресу p происходит только в зависимости от значения ω .

10.2.2. Во внутреннем цикле (см. решение упражнения 9.2.3) выгодно иметь на индекс-регистре значение $k = i + j$. С учетом этого АЛГОЛ-программу перепишем в виде

```

for j := 1 step 1 until n do
  for k := n step -1 until j do
    f[k] := (f[k] - f[k-1]) / (x[k] - x[k-j])

```

Соответствующая машинная программа может быть такой:

	0	ПЧ	z ₁	0	p+1	} j := 1
	0	ПЧ	z ₂	0	p+3	
	0	AI	0	+n	0	H := n
p	6	—	f	f-1	r	r := f _k - f _{k-1}
	6	—	x	x-j	r ₁	r ₁ := x _k - x _{k-j}
	1	/	r	r ₁	f	f _k := r / r ₁
	1	H ≥ A	j+1	p	7777	H := H - 1; если H ≥ j, то на p
	0	-M	p+1	z ₃	p+1	} j := j+1
	0	+M	p+3	z ₄	p+3	
	0	H < A	n-1	p	+n	H := n; если j-1 < n-1, то на p

Константы:

z ₁	6	—	x	x-1	r ₁
z ₂	1	H ≥ A	2	p	7777
z ₃	0	0	0	1	0
z ₄	0	0	1	0	0

В этой программе использовано то обстоятельство, что из внутреннего цикла мы выходим со значением $H = j - 1$.

10.2.3. Из двух использованных ранее алгоритмов транспонирования (см. пример 9.2.2 и упражнение 9.2.2) сейчас, когда мы собираемся прибегнуть к помощи индекс-регистра, гораздо выгоднее взять за основу второй вариант, так как в нем все переменные адреса одинаково зависят от параметра внутреннего цикла. Пусть во время выполнения рабочей части внутреннего цикла содержимое индекс-регистра равно $(i-1)(n+1)$. Это приводит к программе:

	0	ПЧ	z_1	0	p	} $k := 0$
	0	ПЧ	z_2	0	$p+1$	
	0	ПЧ	z_3	0	$p+2$	
	0	ПЧ	z_4	0	$p+3$	
p_1	0	+M	p	z_5	p	} $k := k+1$
	0	+M	$p+1$	z_6	$p+1$	
	0	+M	$p+2$	z_7	$p+2$	
	0	-M	$p+3$	z_8	$p+3$	
	0	АИ	0	0	0	$i := 1$
p	4	ПЧ	$a+k$	0	r	$r := a[i, i+k]$
	5	ПЧ	$a+kn$	0	$a+k$	$a[i, i+k] := a[i+k, i]$
	1	ПЧ	r	0	$a+kn$	$a[i+k, i] := r$
	1	$I < A$	$(n-k-1)(n+1)$	p	$n+1$	$i := i+1$, если $i \leq n-k$, то на p
	0	$\neq$	$p+3$	z_9	0	Если $k \neq n-1$, то
	0	П0	0	p_1	0	перейти на p_1

с константами:

z_1	4	ПЧ	a	0	r
z_2	5	ПЧ	a	0	a
z_3	1	ПЧ	r	0	a
z_4	1	$I < A$	n^2-1	p	$n+1$
z_5	0	0	1	0	0
z_6	0	0	+n	0	1
z_7	0	0	0	0	+n
z_8	0	0	$n+1$	0	0
z_9	1	$I < A$	0	p	$n+1$

Однако эту программу можно сократить, если во время выполнения рабочей части внутреннего цикла на индекс-регистр помещать значение $(i-1)(n+1) + kn$, а перед входом в цикл — значение kn . Программа примет вид:

	0	ПЧ	z_1	0	p	} $k := 0$
	0	ПЧ	z_2	0	$p+1$	
	0	АИ	0	+n	0	$k := 1$ ($I = kn$)
p_1	0	-M	p	z_3	p	} $k := k+1$
	0	-M	$p+1$	z_4	$p+1$	
	6	АИ	0	0	p_2	$i := 1$ ($I = (i-1)(n+1) + kn$)
p	4	ПЧ	$a-k(n-1)$	0	r	$r := a[i, i+k]$
	5	ПЧ	a	0	$a-k(n-1)$	$a[i, i+k] := a[i+k, i]$
	1	ПЧ	r	0	a	$a[i+k, i] := r$
	1	$I < A$	$n(n-1)$	p	$n+1$	$i := i+1$, если $i \leq n-k$, то на p
p_2	0	АИ	0	kn	0	$I := kn$
	1	$I < A$	$n(n-1)$	p_1	+n	$k := k+1$, если $k \leq n-1$, то на p_1

при константах:

z_1	4	ПЧ	a	0	r
z_2	5	ПЧ	a	0	a
z_3	0	0	$n-1$	0	0
z_4	0	0	0	0	$n-1$

11.1. Один из возможных вариантов программы:

	0	ПЧ	z_1	0	r_2	$si := 0$
	0	ПЧ	0	0	i_1	} $i := 0$
	0	ПЧ	0	0	r_3	
	0	ПЧ	z_2	0	p_6	$ij := 0$
	4	АИ	0	0	p_7	$I := 0$, запоминание I
p_1	0	+M	r_2	i_1	r_2	$si := si + i$
	0	+M	z_3	i_1	$p_4 + 2$	} Формирование для текущего i
	0	+M	z_4	i_1	$p_6 + 2$	
	0	ПЧ	r_2	0	r_4	$sj := 0$
	0	ПЧ	z_5	0	$p_3 + 2$	$j := 0$
p_2	4	АИ	0	0	j_1	$k := 0, j := I$
	0	+M	r_4	j_1	r_4	$sj := sj + j$
	0	ПЧ	r_4	0	p_3	$ik := si, kj := sj$
	0	ПЧ	0	0	s	$s := 0$
	0	+M	p_3	z_6	p_3	$ik := ik + 1, kj := kj + 1$
p_3	0	×	$a + ik$	$a + kj$	r	$a_{ik} \times a_{kj}$
	0	+	s	r	s	$s := s + a_{ik} \times a_{kj}$
	1	$I < A$	+j	$p_3 - 1$	1	$k := k + 1$, если $k \leq j$, то на $p_3 - 1$
	0	+M	z_7	j_1	r_1	$k := j$
	0	ПЧ	p_3	0	p_4	Формирование
	1	$I \geq A$	0	$p_4 + 2$	-1	$k := k - 1$, перейти на $p_4 + 2$
	0	+M	r_1	z_8	r_1	$k := k + 1$
	0	+M	p_4	r_1	p_4	$ik := ik + 1, kj := kj + k$
p_4	0	×	$a + ik$	$a + kj$	r	$a_{ik} \times a_{kj}$
	0	+	s	r	s	$s := s + a_{ik} \times a_{kj}$
	1	$I < A$	+i	$p_4 - 2$	1	$k := k + 1$, если $k \leq i$, то на $p_4 - 2$
	0	ПЧ	r_3	0	r_1	$k := i$
	0	ПЧ	p_4	0	p_5	Формирование
	1	$I \geq A$	0	$p_5 + 2$	-1	$k := k - 1$, перейти на $p_5 + 2$
	0	+M	r_1	z_6	r_1	$k := k + 1$
	0	+M	p_5	r_1	p_5	$ik := ik + k, kj := kj + k$
p_5	0	×	$a + ik$	$a + kj$	r	$a_{ik} \times a_{kj}$
	0	+	s	r	s	$s := s + a_{ik} \times a_{kj}$
	1	$I < A$	$n - 1$	$p_5 - 2$	1	$k := k + 1$, если $k \leq n - 1$, то на $p_5 - 2$
	0	+M	$p_3 + 2$	z_7	$p_3 + 2$	$j := j + 1$
	0	+M	p_6	z_8	p_6	$ij := ij + 1$
p_6	0	ПЧ	s	0	$b + ij$	$b_{ij} := s$
i_1	0	АИ	0	+j	0	$I := j$
	1	$I < A$	+i	p_2	1	$I := I + 1$, если $I \leq i$, то на p_2
	0	+M	i_1	z_7	i_1	} $i := i + 1$
	0	+M	r_3	z_6	r_3	
	0	$I < A$	+n	p_1	0	$j := 0$, если $I \leq n$, то на p_1
p_7	0	АИ	0	~	0	Восстановление I

Во время работы этой программы ее рабочие ячейки содержат следующие коды:

i_1	0	0	+i	0	0	
r_1	{	0	0	1	+k	0 (в цикле $p_1 - 2 \div p_4 + 2$)
		0	0	+k	+k	0 (в цикле $p_5 - 2 \div p_5 + 2$)
r_2		0	×	$a + si$	a	r
r_3		0	0	+i	+i	0
r_4		0	×	$a + si$	$a + sj$	r

Константы

z_1	0	$\times$	a	a	r
z_2	0	$\Pi\mathcal{C}$	s	0	b
z_3	1	$H < A$	0	$p_4 - 2$	1
z_4	1	$H < A$	0	p_2	1
z_5	1	$H < A$	0	$p_3 - 1$	1
z_6	0	0	1	1	0
z_7	0	0	1	0	0
z_8	0	0	0	1	0
z_9	0	0	0	0	1

12.1.1.

а)	0	—	$\langle 1 \rangle$	x	y	$y := \neg x$	$(= 1 - x)$
б)	0	$\times$	x	y	z	$z := x \wedge y$	$(= xy)$
в)	0	+	x	y	z		
	0	$\times$	x	y	r		
	0	—	z	r	z	$z := x \vee y$	$(= x + y - xy)$

или

	0	—	0	x	0	if x then	
p	0	$\Pi 1$	$\langle 1 \rangle$	$p + 2$	z	$z := \text{true}$	
	0	$\Pi\mathcal{C}$	y	0	z	else $z := y$	
г)	0	—	y	x	0		
p	0	$\Pi 0$	$\langle 1 \rangle$	$p + 2$	z	} $z := x \supset y$ $(= y \geq x)$	
	0	$\Pi\mathcal{C}$	0	0	z		
д)	0	—	0	x	0	if x then	
p	0	$\Pi 1$	y	$p + 2$	z	$z := y$	
	0	—	$\langle 1 \rangle$	y	z	else $z := \neg y$	
е)	0	—	a	b	0	if $a < b$ then	
p	0	$\Pi 1$	$\langle 1 \rangle$	$p + 2$	z	$z := \text{true}$	
	0	$\Pi\mathcal{C}$	0	0	z	else $z := \text{false}$	
ж)	0	—	0	x	0	if $\neg x$ then	
p	0	$\Pi 0$	y	$p + 4$	z	$z := y$	
	0	$\neq$	a	b	0	else if $a = b$ then	
	0	$\Pi 0$	$\langle 1 \rangle$	$p + 4$	z	$z := \text{true}$	
	0	$\Pi\mathcal{C}$	0	0	z	else $z := \text{false}$	

12.1.2. Под массив *lambda* отводятся 5 ячеек с адресами $lambda + 1 \div lambda + 5$. Значение элемента $lambda[i, j, k]$ размещается в разряде с номером $i + 4k$ ячейки с адресом $lambda + j + 3$.

12.2.1.

$\wedge$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1
2	0	0	2	2	0	0	2	2
3	0	1	2	3	0	1	2	3
4	0	0	0	0	4	4	4	4
5	0	1	0	1	4	5	4	5
6	0	0	2	2	4	4	6	6
7	0	1	2	3	4	5	6	7

$\vee$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	1	3	3	5	5	7	7
2	2	3	2	3	6	7	6	7
3	3	3	3	3	7	7	7	7
4	4	5	6	7	4	5	6	7
5	5	5	7	7	5	5	7	7
6	6	7	6	7	6	7	6	7
7	7	7	7	7	7	7	7	7

$\equiv$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	1	0	3	2	5	4	7
2	2	3	0	1	6	7	4	5
3	3	3	2	1	0	7	6	5
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

12.2.2. $\wedge$ — дизъюнкция, $\vee$ — конъюнкция, $\equiv$ — эквивалентность.

12.2.3. Пусть x, y — любые множества рассматриваемого вида, u — универсальное множество, изображаемое машинным словом 777 7777 7777 7777. Тогда пересечение $x \cap y$ может быть образовано с помощью машинной операции $x \wedge y$, объединение $x \cup y$ — с помощью операции $x \vee y$, дополнение $\bar{x}$ множества x — с помощью операции $x \neq u$. Разность $x \setminus y$ множеств x и y находится с помощью двух машинных операций $x \wedge (y \neq u)$. Машинной операции $x \neq y$ соответствует так называемая симметрическая разность множеств x и y : $x \Delta y = (x \setminus y) \cup (y \setminus x)$.

12.3.1. Число выполняемых команд в программах дается следующей таблицей:

Номер программы	Область определяется неравенствами							
	$x^2 + y^2 > 1$				$x^2 + y^2 \leq 1$			
	$x > 0$		$x \leq 0$		$x > 0$		$x \leq 0$	
	$y > 0$	$y \leq 0$	$y > 0$	$y \leq 0$	$y > 0$	$y \leq 0$	$y > 0$	$y \leq 0$
12.3.1	14	13	13	12	13	12	12	11
12.3.2	6	6	6	6	10	10	8	8
12.3.3	5	5	5	5	10	9	7	7

12.4.1. Достаточно выполнить две команды:

$$\begin{array}{l} 0 \neq p \quad q \quad r \\ 0 \wedge \quad r \quad z \quad 0 \end{array}$$

где константа z имеет вид:

$$0 \quad 77 \quad 0 \quad 0 \quad 0$$

Если коды операций команд p и q совпадают, то признак ω получит значение 1, в противном случае — 0.

12.4.2. Возможен такой вариант программы:

	0	АН	0	1	s	$H := 1, s := 0$
p_1	4	ПЧ	a	0	r	Выборка очередной ячейки
p_2	0	$\wedge$	r	z_1	r_1	Выделение очередной литеры
	0	$\neq$	r_1	z_2	0	Если код литеры $\neq 022$,
	0	ПО	0	p_3	0	то перейти на p_3 ,
	0	+M	s	z	s	иначе $s := s + 1$
p_3	0	$\neq$	r_1	z_3	0	Если код литеры $\neq 023$,
	0	ПО	0	p_4	0	то перейти на p_4 ,
	0	-M	s	z	s	иначе $s := s - 1$
	0	П1	0	L	0	Если $s < 0$, то на L
p_4	0	$\leftarrow CA$	107	r	r	} Если не все литеры исследованы, то перейти на p_2
	0	$\wedge$	r	z_1	r	
	0	ПО	0	p_2	0	
	1	$H < A$	+n	p_1	1	$H := H + 1$, если $H \leq n$, то на p_1
	0	-	0	s	0	Если $s > 0$, то
	0	П1	0	M	0	перейти на M

Константы:

z	0	0000	...	1
z_1	0	7740	...	0
z_2	0	1100	...	0
z_3	0	1140	...	0
z_4	0	7777	...	7

Переход на метку L означает, что для некоторой закрывающей скобки нет парной ей открывающей скобки. Переход на метку M означает, что для некоторой открывающей скобки нет парной ей закрывающей скобки.

13.1.1. Мы уже видели, что для получения хорошо округленного результата в основных разрядах к сумме мантисс надо прибавить число, изображаемое единицей в младшем дополнительном разряде, и это число уже было прибавлено. При нормализации вправо место младшего дополнительного разряда займет младший основной разряд, и единицу следовало бы добавлять в этом разряде. Проще всего достичь этой цели, прибавив еще одну единицу к младшему дополнительному разряду перед сдвигом мантиссы вправо. Две единицы младшего дополнительного разряда как раз и составят единицу младшего основного разряда. Новой нормализации вправо не потребуется. Действительно, мантиссы операндов m_1 и m_2 удовлетворяют неравенствам $m_1 \leq 1 - 2^{-36}$, $m_2 \leq 1 - 2^{-36}$. Сложив эти мантиссы и добавив к их сумме число 2^{-36} для округления, получим число

$$m_1 + m_2 + 2^{-36} \leq 2 - 2^{-36} < 2.$$

Переноса в разряд, следующий за старшим дополнительным, не произойдет.

13.1.2. Пусть операнды равны $x_1 = 2^{p_1} m_1$, $x_2 = 2^{p_2} m_2$ (случай отрицательных операндов не вносит никакого существенного отличия). Если $|p_1 - p_2| \leq 1$, то сдвиг при выравнивании порядков производится не более чем на один разряд. Благодаря наличию младшего дополнительного разряда все цифры разности мантиссы сохраняются на сумматоре. Если потребуется нормализация влево, то цифра младшего дополнительного разряда сдвинется в основные разряды и результат окажется точным. Если нормализации влево не произойдет, то эта цифра будет отброшена и, так как весь хвост сдвинутой мантиссы состоит из одной этой цифры, погрешность результата не превысит $2^p \cdot 2^{-37}$, где p — большее из чисел p_1 и p_2 , т. е. приближение окажется хорошим.

Пусть теперь $|p_1 - p_2| > 1$. Примем для определенности, что $p_1 > p_2$ (случай $p_1 < p_2$ рассматривается аналогично). После выравнивания порядков из мантиссы $m_1 \geq 1/2$ вычитается число m'_2 , представленное сдвинутой не менее чем на два разряда мантиссы m_2 . Ясно, что $m'_2 < 1/4$, поэтому $m_1 - m'_2 > 1/4$. Следовательно, при нормализации влево сдвиг произойдет не более чем на один разряд. Если сдвиг не потребуется, то, как мы уже знаем, результат будет хорошим приближением к точной разности операндов. Осталось разобрать случай, когда при нормализации влево произойдет сдвиг на один разряд, и порядок p заменится порядком $p' = p - 1$. Погрешность от отбрасывания всех цифр хвоста сдвинутой мантиссы, кроме старшей (сохранившейся в младшем дополнительном разряде), не превзойдет $2^p \cdot 2^{-37} = 2^{p'} \cdot 2^{-36}$, т. е. результат окажется приемлемым приближением к точной разности.

13.3.1 и 13.3.2. Для одновременного вычисления $y = [x]$ и $z = \{x\}$ можно воспользоваться следующей программой:

	0	+ a	x	c	y	y := [x] или [x] + 1
	0	-	x	y	z	z := {x} или {x} - 1
p	0	ПО	0	$p+3$	0	Если $z \geq 0$, то на $p+3$
	0	-	y	<1>	y	y := [x] }
	0	+	z	<1>	z	z := {x} } при $x < 0$,

где c — адрес константы 145 0000 0000 0000. Если нужно получить только значение $[x]$ или только $\{x\}$, то команду $p + 2$ или $p + 1$ можно исключить, соответственно изменив второй адрес команды p на $p + 2$.

14.1.1. В ячейки 0021—0026 будут введены следующие слова:

0021	0	10	0100	0000	0000
0022	0	50	0015	0000	7777
0023	0	70	0100	0000	0000
0024	0	50	4411	0000	7777
0025	0	70	0100	0022	0000
0026	0	56	0000	0100	0000

Содержимое других ячеек ОЗУ не изменится.

14.3.1. Если хотя бы одна из двух групп разрядов первого операнта: с 1-го по 36-й и с 37-го по 45-й — состоит сплошь из нулей, то при циклическом сложении со словом 777 7777 7777 7777 соответствующая группа разрядов результата будет заполнена единицами. Ненулевая группа разрядов при сложении с таким словом переходит в результат без изменения.

14.3.2. Коммутативность и ассоциативность операции циклического сложения являются простым следствием коммутативности и ассоциативности сложения натуральных чисел и определения операции циклического сложения.

14.4.1. При выполнении команды $p + 1$ произойдет останов, во время которого следует вновь (в третий раз) установить вводимые перфокарты на читающее устройство на случай несовпадения контрольных сумм, полученных при первом и втором вводе. Однако, если суммы совпадут, то программа начнет выполняться дальше. Если в ней предусмотрен ввод еще одного массива, то читающее устройство может заработать раньше, чем оператор машины успеет снять уже не нужную первую колоду карт.

14.4.2. Программа может состоять всего из одной команды

p 0 BB a p 0

Если при выполнении этой команды подсчитанная контрольная сумма не совпадет с суммой, введенной с карт, то машина остановится. Оператор должен вновь установить карты на читающее устройство и пустить машину. Управление автоматически будет передано команде p и начнется повторный ввод. При совпадении сумм программа станет выполняться дальше с команды $p + 1$.

14.4.3. На команду РЗ мы приходим со значениями I вида $511_{10} \cdot k$, $k = 1, 2, \dots, 8$. Среди этих значений должно найтись хотя бы одно, удовлетворяющее неравенству $I \geq n$. При $n > 511 \cdot 8 = 4088$ этого не случится. Но этот дефект программы нельзя считать серьезным, так как в реальных программах накопить такой массив чисел в оперативной памяти практически невозможно (не остается места для программы). Кроме того, программа 14.4.5 (как и программа 14.4.4) не будет работать при $n = 0$. Исполнительные адреса, определяющие начало и конец выводимого массива, при $n = 0$ равны $a + 1$ и a . Будет сделана попытка записать на буфер и отпечатать все содержимое оперативной памяти, начиная с ячейки $a + 1$ и кончая ячейкой a (с переходом к ячейке 0000 после ячейки 7777). Но такая попытка, естественно, ни к чему хорошему не приведет, так как буфер состоит всего из 512 ячеек.

14.4.4. В конце шага C1 и в начале шага C2 (перед командой C2 программы 14.4.7) следует вставить команды:

R1 + 2	4	AI	0	0012	k	$k := I, I := 10$
C2'	0	Ma	0014	0	z	} Запись массива $a \div z$ на нулевой барабан
	0	Mб	a	0	r	
	0	$\neq$	r	s0	0	Если $r \neq s0$, то
	0	П0	0	C2'	0	перейти на C2'

0	Ma	4410	0	z	}	Фиктивное чтение с барабана
0	Mб	a	C2'	r		
0	$\neq$	r	s0	0	Если $r \neq s0$, то	
0	П0	0	C2'	0	перейти на C2'	
1	$I < A$	0012	C3—1	0001	$I := I + 1$, если $I \leq 10$, то на C3—1	
0	KИ	0	k	0	$I := k$	

Команду C3—1 программы 14.4.7 следует заменить двумя командами:

C3—2 4 АИ 1 1 k $k := I + 1$, $I := 1$

C3—1 4 КИ 0 i j $j := I$, $I := I_{\text{осн}}$

Блок C9 будет состоять лишь из двух команд вместо трех в программе 14.4.7:

C9 0 КИ 0 j 0 $I := j$

0 ПБ s C2' s0 $s0 := s$, перейти на C2'

Наконец, блок В следует переделать так:

В	0	Ma	0410	0	z	}	Чтение с барабана
	0	Mб	a	В	r		
	0	$\neq$	r	s0	0	Если $r \neq s0$, то	
	0	П0	0	В	0	перейти на В	
q	0	АИ	0	a	0	}	Переход на вычисление контрольной суммой с возвратом
	0	ПВ	q	C1+1	R1		
	0	$\neq$	s	s0	0	Если $s \neq s0$, то	
	0	П0	0	В	0	перейти на В	
	0	КИ	0	i	0	$I := I_{\text{осн}}$	
	0	ПБ	R2	R	R1	Перейти на А с выходом по адресу из ячейки R2	

14.4.5. Программа, составленная без особых ухищрений, такова:

p_0	0	АИ	0	1	0	$I := 1$
	1	ПЧ	0	0	a	} Заполнение нулями массива
	1	$I < A$	1750	p_0	1	
	1	ПЧ	w_1	0	a	} Заполнение единицами массива
	1	$I < A$	3720	$p_0 + 2$	1	
p_1	0	Ма	0041	0	1750	} Разметка на первом магнитофоне
	0	Мб	1	0	0	
	0	АИ	0	2	0	$i := 2$
	0	ПЧ	z_1	0	p_4	$\Delta := -2$
	2	Ма	0041	0	$a + 1750$	} Разметка i -й зоны нулями
0	Мб	$a + 1$	0	0		
p_2	2	Ма	5421	0	1750	} Фиктивное чтение i -той зоны
	0	Мб	1	$p_2 + 2$	r	
	0	$\neq$	r	0	0	} Если сумма верна, то
	0	П1	0	p_6	0	
	2	Ма	1025	0	$a + 1750$	} Запись в i -ю зону массива нулей
0	Мб	$a + 1$	0	0		

p_3	2	Ma	5421	0	1750	}	Фиктивное чтение
	0	$M6$	1	p_4	r		
	0	$\neq$	r	0	0		Если сумма верна, то
p_4	0	$\Pi 1$	0	p_6	0	}	перейти на p_6
	2	Ma	7021	Δ	1750		
	0	$M6$	1	0	0		мером $i + \Delta$
p_5	2	Ma	0041	7777	1750	}	Разметка зоны с номером $i - 1$
	0	$M6$	1	0	0		
	0	$\Pi Б$	z_2	$p_1 + 1$	p_4		$\Delta := -1$
p_6	2	Ma	1025	0	$a + 3720$	}	Запись в i -ю зону массива единиц
	0	$M6$	$a + 1751$	0	0		
	2	Ma	5421	0	1750	}	Фиктивное чтение
p_7	0	$M6$	1	$p_7 + 2$	r		
	0	$\neq$	r	w_1	0		Если сумма верна, то
	0	$\Pi 1$	0	p_9	0		перейти на p_9
p_8	2	Ma	1025	0	$a + 3720$	}	Повторная запись массива единиц
	0	$M6$	$a + 1751$	0	0		
	2	Ma	5421	0	1750	}	Фиктивное чтение
p_9	0	$M6$	1	p_4	0		
	0	$\neq$	r	w_1	0		Если сумма неверна, то
	0	$\Pi 0$	0	p_4	0		перейти на p_4
p_9	1	$И < A$	0764	p_1	2		$i := i + 2$, если $i \leq 500_{10}$, то на p_1
	0	$СТОП$	0	0	0		

В программе используются константы:

w_1	7	77	7777	7777	7777
z_1	2	Ma	7021	7776	1750 ($\Delta = -2$)
z_2	2	Ma	7021	7777	1750 ($\Delta = -1$)

14.5.1. Сначала число n следует денормализовать, так, чтобы цифра единиц сдвинулась в 1-й разряд ячейки, а код порядка стал бы равен 144 (в дальнейшем этот код интереса не представляет). Затем целое двоичное число $m_0 = n$, представленное младшими 36 разрядами ячейки, следует преобразовать в число m'_6 , двоичная запись которого совпадает с двоично-десятичной записью числа n . Это можно сделать последовательными преобразованиями, обратными преобразованию (14.5.1):

$$m_i = m_{i-1} + 6k_i \cdot 2^{4(i-1)}, \quad i = 1, \dots, 8,$$

причем

$$k_i = [k_{i-1}/10], \\ k_{i-1} = [m_{i-1}/2^{4(i-1)}],$$

где $[x]$ обозначает целую часть числа x . Из этих соотношений вытекает формула

$$m_i = m_{i-1} + 6 [m_{i-1}/(10 \cdot 2^{4(i-1)})] \cdot 2^{4(i-1)}.$$

Разобраться в деталях приводимой ниже программы предоставляем читателю.

	0	, +	n	z_1	n	Денормализация
	0	$\leftarrow MA$	103	z_2	r_1	$i := 1$
p	0	, ×	n	z_3	r	$m_{i-1} \cdot 8/10$
	0	$\wedge$	r	r_1	r	$[m_{i-1}/(10 \cdot 2^{4(i-1)})] \cdot 2^{4i-1}$

0	П1	0	p_1	0	Если $m_i = m_{i-1}$, то на выход
0	,X	r	z_4	r	$6 [m_{i-1}/(10 \cdot 2^{4(i-1)})] \cdot 2^{4(i-1)}$
0	+M	n	r	n	m_i
0	←МП	$\langle 10 \rangle$	r_1	r_1	$i := i + 1$, если $i \leq 8$, то
0	П0	0	p	0	перейти на p
p_1	0	$\wedge$	n	z_2	n Очистка порядка

Константы:

z_1	1 44	0000	0000	0000	
z_2	6 00	7777	7777	7777	
z_3	1 00	6314	6314	6315	$2^0 \cdot (8/10)$
z_4	1 00	6000	0000	0000	$2^0 \cdot (3/4)$

14.5.2. Программа последовательно переводит число n из системы счисления с основанием 10^i в систему с основанием 10^{2i} для $i = 1, 2, 4, 8$. Под одну цифру (кроме старшей) записи числа в системе с основанием 10^{2i} отводится $4 \cdot 2i$ двоичных разрядов. В этих разрядах цифра изображается своим двоичным представлением. Перед очередным повторением цикла каждая группа на $4 \cdot 2i$ разрядов содержит число вида $d_1 \cdot 16^i + d_0$, где d_1 и d_0 — цифры в системе с основанием 10^i . Чтобы получить из этого числа изображение цифры в системе с основанием 10^{2i} , нужно выделить старшие $2 \cdot 2i$ разрядов, умножить представленное ими число на $16^i - 10^i$ и вычесть это произведение из исходного числа. Именно эти операции и выполняются в программе.

14.5.3. Требуемую программу можно получить, заменив первые две команды программы 14.5.4 командами:

0	/	n	z_0	x	$x := n/10^9$
0	П1	z_2	q	r_1	$i := 0$, если $x \geq 1$, то на q
0	,+a	x	z_1	x	

Здесь константа z_0 имеет вид

$$z_0 \quad 1 \ 36 \ 7346 \ 5450 \ 0000 \quad 10^9$$

константы z_1 и z_2 — такие же, как в программе 14.5.4. Переход на метку q означает, что заданное число n больше или равно 10^9 и его точный перевод в десятичное представление рассматриваемого вида невозможен.

14.5.4. Такое «упрощение» алгоритма Д2 привело бы к тому, что при $|x| > 2^{63} (1 - 2^{-36})/10$ на шаге Д2 $\cdot 2$ возникал бы «авост» из-за невозможности представить результат в виде машинного числа.

15.3.1.

Вычисление значения n
в ячейке 0001

k	0	ПВ	$k+1$	f_n	f_k	Переход на f_n с возвратом
$k+1$	0	+	s	s	s	$s := 2s$
$k+2$	0	+	n	n	n	$n := 2n$
$k+3$	0	ПВ	$k+4$	f_n+2	f_k	Переход на f_n+2 с возвратом

Требуемое значение вычисляется в ячейке s . Здесь использовано тождество

$$f(n) + f(2n) = 2f(n) + \sum_{i=n+1}^{2n} 1/i$$

и то обстоятельство, что после вычисления $f(n)$ значение i остается равным $n+1$. Однако на подобные ухищрения можно идти лишь в случае очень простых под-

программ, структура и схема работы которых досконально известны программисту.

15.5.1. Наиболее естественным представляется следующий алгоритм вычислений

```

for  $i := 1$  step 1 until  $m$  do
  for  $k := 1$  step 1 until  $n$  do
    begin  $c[i, k] := 0$ ;
          for  $j := 1$  step 1 until  $p$  do
             $c[i, k] := c[i, k] + a[i, j] \times b[j, k]$ 
          end
    end
end

```

Однако класть этот алгоритм в основу машинной программы невыгодно. Дело в том, что в самом внутреннем цикле этого алгоритма (цикле с параметром j) переменным $a[i, j]$ и $b[j, k]$ соответствуют ячейки, адреса которых по-разному зависят от параметра цикла. Это затрудняет использование индекс-регистра во внутреннем цикле и увеличивает число команд в рабочей части цикла. С этой точки зрения выгоднее перестроить алгоритм так, чтобы параметром самого внутреннего цикла была переменная k . Это можно сделать, например, так:

```

for  $i := 1$  step 1 until  $m$  do
  for  $k := 1$  step 1 until  $n$  do
     $c[i, k] := 0$ ;
    for  $j := 1$  step 1 until  $p$  do
      for  $k := 1$  step 1 until  $n$  do
         $c[i, k] := c[i, k] + a[i, j] \times b[j, k]$ 
      end
    end
  end
end

```

Соответствующая программа такова:

	0	АИ	0	$m \times n$	0	$H := m \times n$
q	1	ПЧ	0	0	c	$c[i, k] := 0$ ($H :=$
	1	$H \geq A$	2	q	7777	$= (i-1)n + k$)
						$H := H - 1$, если
						$H \geq 1$, то на q
	0	ПЧ	z_1	0	q_1	$i := 1, j := 1$
	0	ПЧ	z_2	0	$q_1 + 1$	$i := 1$
	4	АИ	0	1	q_2	Запоминание i ,
						$j := 1$ ($H = i - 1$)
	4	АИ	0	1	q_2	Запоминание j ,
						$k := 1$ ($H = j$)
q_1	2	$\times$	$a + (i-1)p + j$	$b + (j-1)n$	r	$a_{ij} \times b_{jk}$ ($H = k$)
	5	$+$	$c + (i-1)n$	r	$c + (i-1)n$	$c_{ik} := c_{ik} +$
						$+ a_{ij} \times b_{jk}$
	1	$H < A$	$+n$	q_1	1	$k := k + 1$, если
						$k \leq n$, то на q_1
	0	$+M$	q_1	z_3	q_1	$j := j + 1$
q_2	0	АИ	0	$+j$	0	$H := j$
	1	$H < A$	$+p$	$q_1 - 1$	1	$j := j + 1$, если
						$j \leq p$, то на $q_1 - 1$
	0	$-M$	q_1	z_4	q_1	$i := i + 1, j := j - p$
	0	$+M$	$q_1 + 1$	z_5	$q_1 + 1$	$i := i + 1$

q_3	0	АИ	0	$i-1$	0	$I := i-1$
	1	$I < A$	$m-1$	q_1-2	1	$I := I+1$, если $I \leq m-1$, то на q_1-2

Программа использует константы:

z_1	2	$\times$	$a+1$	b	r
z_2	5	$+$	c	r	c
z_3	0	0	1	$+n$	0
z_4	0	0	0	$p \times n$	0
z_5	0	0	$+n$	0	$+n$

Здесь переменная команда q_1 формируется сразу по i и по j . В дальнейшем переадресация этой команды по i совмещается с восстановлением по j . Кроме того, использованы следующие программистские трюки. Двойной алгольный цикл, в котором в ячейки $c[i, k]$ засылаются нули, превращен в одинарный. Это оказалось возможным благодаря тому, что в индекс-регистр засылается текущее значение выражения $(i-1)n+k$, которое должно изменяться в пределах от 1 до mn . Здесь выбрана обратная последовательность изменения значения индекс-регистра (от mn до 1). После выхода из цикла значение индекс-регистра равно нулю. В цикле по i принято, что текущее значение индекс-регистра на единицу меньше текущего значения i . Поэтому специальная команда посылки начального значения в индекс-регистр в этом цикле не понадобилась.

Перейдем теперь к составлению формирующей части. Если сохранить рабочую часть такой, как она написана, то формированию подлежат все константы ($z_1 \div z_5$), а также команды $q-1, q, q_1+2, q_2+1, q_3+1$, содержащие адреса, значения которых зависят от параметров обращения к подпрограмме. Однако заготовки z_1 и z_2 , а также команды $q+2$ и $q+3$ нужны лишь для формирования содержимого ячеек q_1 и q_1+1 . Естественно потребовать, чтобы формирующая часть взяла это формирование (которое выполняется однократно) на себя. Тогда названные заготовки и команды окажутся излишними. От формирующей части можно потребовать также, чтобы она засылала в индекс-регистр значение mn вместо того, чтобы формировать команду $q-1$, осуществляющую эту засылку.

Окончательно подпрограмма вместе с формирующей частью и константами, включенными в текст программы, имеет вид:

p_n	0	$\leftarrow$ МП	$z+3$	a_1	r	«0	АИ	a	0	0»
	0	$+$ М	z	r	r	«2	$\times$	$a+1$	0	г»
	0	$+$ М	r	b_1	q_1	«2	$\times$	$a+1$	b	г»
	0	$\leftarrow$ МА	0064	c_1	r	«0	АИ	0	0	с»
	0	$-$ ПО	r	r	q	«1	00	0	0	с»
	0	$+$ М	$z+1$	r	q_1+1	«5	$+$	0	r	с»
	0	$\leftarrow$ МП	$z+3$	c_1	r	«0	АИ	c	0	0»
	0	$+$ М	q_1+1	r	q_1+1	«5	$+$	c	r	с»
	0	$\leftarrow$ МП	$z+3$	m_1	m_1	«0	АИ	$+m$	0	0»
	0	$+$ М	$z+2$	m_1	q_3+1	«1	$I < A$	$m-1$	q_1-2	1»
	0	$+$ М	$z+3$	n_1	$z+6$	«1	14	1	$+n$	0»
	0	$\leftarrow$ МП	$z+3$	n_1	r	«0	АИ	$+n$	0	0»
	0	$+$ М	$z+4$	r	q_1+2	«1	$I < A$	$+n$	q_1	1»
	0	$\leftarrow$ МА	0064	n_1	n_1	«0	АИ	0	0	$+n$ »
	0	$+$ М	r	n_1	$z+10$	«3	АИ	$+n$	0	$+n$ »
	0	$\leftarrow$ МП	$z+3$	p_1	p_1	«0	АИ	$+p$	0	0»
	0	$+$ М	$z+5$	p_1	q_2+1	«1	$I < A$	$+p$	q_1-1	1»

	0	, $\times a$	r	p_1	$z+7$	«0	24	0	np	0»
	0	, $\times a$	m_1	r	r	«0	24	0	mn	0»
	4	KH	0	r	p_K-1	Запоминание $I_{ст}$, $I := mn$				
q	0	0	0	0	0					
	1	$H \geq A$	2	q	7777					
	4	AI	0	1	q_3					
	4	AI	0	1	q_2					
q_1	0	0	0	0	0					
	0	0	0	0	0					
	0	0	0	0	0					
	0	$+M$	q_1	$z+6$	q_1					
q_2	0	0	0	0	0					
	0	0	0	0	0					
	0	$-M$	q_1	$z+7$	q_1					
	0	$+M$	q_1+1	$z+10$	q_1+1					
q_3	0	0	0	0	0					
	0	0	0	0	0					
	0	0	0	0	0	Восстановление $I_{ст}$				
p_K	0	0	0	0	0	Выход				
z	2	$\times$	1	0	r					
	5	$+$	0	r	0					
	1	$H < A$	7777	q_1-2	1					
	1	14	1	0	0					
	1	$H < A$	0	q_1	1					
	1	$H < A$	0	q_1-1	1					
	0	0	0	0	0	} Рабочие ячейки для формирования констант				
	0	0	0	0	0					
	0	0	0	0	0					
$z+10$	0	0	0	0	0					

15.6.1. Одно из возможных решений таково. Пусть ячейки a_1, a_2, b_1, c_1, c_2 располагаются в следующей последовательности и совмещаются с ячейками $s_1, s_2, d+1, d+2, d+3$ в соответствии с таблицкой:

c_2	s_2
c_1	$d+1$
b_1	$d+2$
a_2	$d+3$
a_1	s_1

Тогда подпрограмма может иметь вид:

p_H	0	$\leftarrow MA$	0114	a_2	r	«0	AI	a_K	0	0»
	0	$\leftarrow MA$	0064	a_1	a_2	«0	AI	0	0	a_H »
	0	$\leftarrow MA$	0114	a_1	a_1	«0	AI	a_H	0	0»
	0	$\leftarrow MA$	0064	b_1	b_1	«0	AI	0	0	b_H »
	0	$-M$	b_1	a_2	a_2	«0	AI	0	0	$+d$ »
	0	$\leftarrow MA$	0114	a_2	b_1	«0	AI	0	$+d$	0»
	0	$+M$	c_1	b_1	c_1	«0	AI	0	c_H	0»
	4	KH	0	c_1	p_K-1	Запоминание $I, i := c_H$				

	0	$+M$	c_2	b_1	c_2	«0 АИ 0 c'_k 0»
	0	$\leftarrow MA$	0114	c_2	c_2	«0 АИ c'_k 0 0»
	0	$+M$	z_2	c_2	p_5	«1 $H < A$ c'_k p 1»
	0	$+M$	r	z_1	c_2	«0 АИ $a_k + 1$ 0 0»
	0	$\leftarrow MA$	0114	b_1	c_1	«0 АИ $+d$ 0 0»
p	2	$\leftarrow MA$	0100	0	r	«л КОП A_1 A_2 A_3 »
	4	АИ	0	0	p_4	Запоминание i , $I := 0$
	0	ПБ	r	p_3	q	$j := 1$, перейти на p_3
p_1	0	$-M$	q	a_1	0	Если $A_j < a_n$, то ($\omega := 1$)
	0	П1	0	p_2	0	перейти на p_2
	0	$-M$	q	c_2	0	Если $A_j > a_k$, то ($\omega := 0$)
	0	ПО	0	p_2	0	перейти на p_2
	2	$+M$	r	c_2	r	$A_j := A_j + d$ ($I = j$)
p_2	0	$\leftarrow MA$	0114	q	q	} $j := j + 1$, если $A_j A_{j+1} A_{j+2} > 0$, то
p_3	1	$0I \geq A$	0	p_1	1	
p_4	0	АИ	0	$+i$	0	Восстановление i
	1	ПЧ	r	0	0	Засылка модифицированной команды на место
p_5	0	0	0	0	0	$i := i + 1$, если $i \leq c'_k$, то на p
	0	0	0	0	0	Восстановление I
p_k	0	0	0	0	0	Выход
z_1	0	0	1	0	0	} Константы
z_2	1	$H < A$	0	p	1	
r	0	0	0	0	0	} Рабочие ячейки
q	0	0	0	0	0	

ЛИТЕРАТУРА

Ниже перечислены основные работы учебного и справочного характера в области программирования, изданные на русском языке. В работах [4, 9, 10, 12, 16, 19, 21, 27, 31, 32] излагаются общие вопросы программирования. Книги [1, 11, 14, 17, 20, 22, 25, 28, 29] описывают системы команд и приемы программирования для ряда отечественных и некоторых зарубежных машин. В работах [2, 5, 6, 7, 8, 18, 23] излагается язык АЛГОЛ 60, в [3, 13, 15, 24, 26, 30] — некоторые другие алгоритмические языки.

1. Айнберг В. Д., Гавриленко Е. Т., Сабсович Л. Л., Программирование для электронных вычислительных машин типа «Урал» («Урал-2», «Урал-3», «Урал-4»), «Наука», 1966.
2. Алгоритмический язык АЛГОЛ 60, пересмотренное сообщение, пер. с англ., «Мир», 1965.
3. Алгоритмический язык АЛГОЛ 68, под общ. ред. А. П. Ершова, «Кибернетика», 1969, № 6, 17—144, 1970, № 1, 13—160.
4. Балакин В. Б., Савинков В. М., Цальп В. Д., Сборник упражнений по программированию, М., «Высшая школа», 1970.
5. Балчев А. Н., Белых В. М., Даугавет В. А., Шидловская Н. А., Сборник упражнений по АЛГОЛ 60, Л., Изд-во Ленинградского ун-та, 1967.
6. Белявский Е. И., Деген А. Б., Этин Ю. Б., АЛГОЛ 60, Л., Гидрометеорологическое изд-во, 1966.
7. Боттенбрух Г., Структура АЛГОЛ 60 и его использование, пер. с англ., ИЛ, 1963.
8. Брудно А. Л., АЛГОЛ, «Наука», 1971.
9. Брудно А. Л., Программирование в содержательных обозначениях, «Наука», 1968.
10. Бухтияров А. М., Зикевская Л. М., Фролов Г. Д., Сборник задач по программированию, под ред. Н. А. Криницкого, «Наука», 1970.
11. Вычислительная система IBM/360, принципы работы, пер. с англ., М., «Советское радио», 1969.
12. Гутер Р. С., Резниковский П. Т., Резник С. М., Программирование и вычислительная математика, вып. 1., Основы программирования, алгоритмические языки, «Наука», 1971.
13. Дал У.-И., Мюрхауг Б., Нюгорд К., СИМУЛА 67, универсальный язык программирования, пер. с англ., «Мир», 1969.
14. Джермейн К., Программирование на IBM/360, пер. с англ., «Мир», 1971.
15. Дубовицкая Р. К. и др., Система автоматической обработки данных на базе языка КОБОЛ, М., «Статистика», 1971.
16. Жоголев Е. А., Трифионов Н. П., Курс программирования, «Наука», 1971.
17. Калачев В. М., Якубович М. М., Программирование для ЭЦВМ «Минск-2» и «Минск-22», М., «Советское радио», 1971.

18. Лавров С. С., Универсальный язык программирования (АЛГОЛ 60), изд. 3-е, «Наука», 1972.
19. Лавров С. С., Гончарова Л. И., Автоматическая обработка данных, хранение информации в памяти ЭВМ, «Наука», 1971.
20. Латышева Т. С., Программирование и решение задач на двухадресных машинах (М-3, «Минск-1»), Изд-во АН СССР, 1963.
21. Ледли Р. С., Программирование и использование цифровых вычислительных машин, пер. с англ., «Мир», 1966.
22. Ляшенко В. Ф., Программирование для цифровых вычислительных машин М-20, БЭСМ-3М, БЭСМ-4, М-220, «Советское радио», 1967.
23. Мак-Кракен Д. Д., Программирование на АЛГОЛе, пер. с англ., «Мир», 1964.
24. Мак-Кракен Д., Дорн У., Численные методы и программирование на ФОРТРАНе, пер. с англ., «Мир», 1969.
25. Марголин М. С., Скоромник М. Г., Столяров Г. К., Чупригина Л. Т., Принципы работы ЭВМ «Минск-23», М., «Статистика», 1970.
26. Меррей-Шелли Р., Программирование на ФОРТРАНе, пер. с англ., М., «Энергия», 1971.
27. Папернов А. А., Логические основы цифровых машин и программирования, изд. 2-е, перераб., «Наука», 1968.
28. Пржицкий В. В., Смирнов Г. Д., Пыхтин В. Я., Электронная вычислительная машина «Минск-32», М., «Статистика», 1972.
29. Савинков В. М., Программирование для ЭВМ «Минск-22», М., «Статистика», 1972.
30. Универсальный язык программирования PL/1, пер. с англ., «Мир», 1968.
31. Штаркман В. С., Перевод чисел из десятичной системы счисления в двоичную, программа СП-0002, в сб. «Интерпретирующая система и элементарные функции», М., Изд-во Вычислительного центра АН СССР, 1965.
32. Шура-Бура М. Р., Система интерпретации ИС-2, в сб. «Интерпретирующая система и элементарные функции», М., Изд-во Вычислительного центра АН СССР, 1965.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Автоматический останов 16, 33, 68, 246—249, 269, 270, 275
- Адрес 14, 16, 37, 65, 73, 76, 78, 81, 84, 108, 109, 118, 177, 181, 194, 238, 293, 296, 310, 316
- Адресная часть команды 38, 69
- Адресный код на перфокарте 257, 262
- Алгоритм 43, 47
- Алгоритмический язык 56, 124
- Арифметические операции 12, 33, 64, 67, 74, 90, 91, 125, 131, 132, 242
- Арифметическое выражение в АЛГОЛе 130, 132, 133, 135, 141, 146, 174
- устройство 12, 14, 243, 265
- Базовый адрес 108, 109, 177, 178, 293
- Безусловное выражение 133, 134
- Безусловный переход 70
- Блок в АЛГОЛе 136, 140, 150, 160, 162
- (программы) 92
- Блок-схема 52
- Блокировка записи в ОЗУ 266, 270
- контроля 266, 270
- нормализации 249
- округления 249, 251
- Буфер вывода 271
- Вводное устройство 12, 17, 255
- Вещественные числовые значения в АЛГОЛе 124
- Внешнее запоминающее устройство 14, 255, 262
- Внешние устройства 255
- Внутренний адрес модуля 316
- Воображаемый блок 162
- Восстановление индекс-регистра 196, 201, 203, 295
- переменных команд 183, 192
- Восьмеричный код чисел и команд 66, 73, 76, 81, 114, 258
- Вывод данных 12, 18, 261, 269, 271
- Выводное устройство 12, 18, 261
- Вызов значением 160, 170
- параметров 160, 162, 170
- по наименованию 160, 171
- Выравнивание порядков 33, 244
- Выражения в АЛГОЛе 130, 131, 135, 139, 141, 142, 146, 158, 162, 171, 172, 175
- Вычислительный блок (блок-схемы) 52
- Глобальный идентификатор 137, 160
- Границы индексов 129, 162
- Граничная пара 130
- Данные 11, 14, 17, 141, 257
- Двоично-десятичное представление чисел 258, 281
- Двойной просчет 276
- Детализация алгоритма 59
- Доокругление 246
- Дополнительные разряды регистров АУ 244—246
- Дополнительный код 34, 65, 114, 246
- Заголовок процедуры 159, 160, 172
- цикла 146
- Заготовка 112
- Закрепленный идентификатор 128, 137
- Запись программы 72, 258
- Запоминание содержимого индекс-регистра 196, 201, 203, 295
- Запоминающее устройство 12, 13, 67, 262
- Зона магнитной ленты 263, 270
- Идентификатор 125, 128, 129, 136, 137, 139, 141, 158—160, 162, 172, 174
- массива 129, 141, 160
- метки 136, 139, 160
- переключателя 174
- переменной 129
- процедуры 160, 162, 172
- Именующее выражение 139, 175
- Индекс-регистр 194, 198
- Индексы 129
- Исполнительный адрес 194
- Итерационный цикл 84, 85, 95
- Код операции 37, 38, 65, 73, 198, 238
- порядка 64, 287
- Команда 12, 37, 65
- Константа переадресации 42, 110, 112, 114, 119, 181, 185
- сравнения 112, 119
- формирования 112, 119
- Контрольная сумма 265, 266, 276
- точка 276
- Кратный цикл 84, 179
- Литера 240, 258, 261
- Логическая операция 126, 132, 219, 224
- Логический блок (блок-схемы) 52
- массива 130, 217, 231
- Логическое выражение 132, 133, 135, 139, 141, 142, 146, 224
- значение 124, 216, 219
- Локальное распределение памяти 297
- Локальный идентификатор 137, 140, 162, 164
- объект 137, 140, 164, 171
- Магнитная лента 262, 270
- Магнитный барабан 262, 269, 271
- Макрокоманда 298
- Макрооперация (макрос) 299
- Мантисса 23, 32, 34, 64, 243, 248, 259, 280
- Маркер на перфокарте 255, 262, 265
- Массив 107, 118, 129, 162, 171, 176, 262

Машинная операция 12, 37, 63, 64, 198
 Машинное слово 64, 217, 219, 238
 — число 242
 Машинный нуль 66, 69, 243, 246
 Местное управляющее устройство 16, 266
 Метка 128, 136, 139, 150, 160
 Модификации арифметических операций 249
 Модификация адресов 194
 Модуль 303

Настройка модуля по месту 316
 — по параметрам 309
 Нормализация 34, 68, 246, 247, 249
 — влево 247
 — вправо 246
 Нормализованная запись числа 23, 64, 242
 Носитель данных 11, 141

Обмен данными с внешней памятью 14, 15, 262, 269
 Обратный код 35
 Обращение к подпрограмме 72, 292, 293, 307
 — к процедуре 158, 162, 172
 Ожидаемый знак результата 244
 Округление 34, 68, 245, 249
 Операнд 12, 15, 37, 125, 126, 134, 172, 198, 243

Оперативное запоминающее устройство 14, 67
 Оператор 135, 136, 139, 140, 142—145, 150, 158, 160, 162

— ввода 141
 — вывода 141
 — «если» 144
 — перехода 139, 143, 149, 175
 — присваивания 135, 172, 224
 — процедуры 158, 162
 — цикла 145

Операции над кодами команд 39, 40, 69, 110, 219, 220, 238
 — над содержимым индекса-регистра 195
 — над текстами 51, 240
 — отношения в АЛГОЛе 126
 — сдвига 221

— управления 16, 70, 196
 Описание 128, 129, 136, 158, 160, 174
 — массива 129
 — переключателя 174
 — процедуры 158, 160, 172
 — типа 129

Основание системы счисления 21
 Основные символы АЛГОЛа 125, 127, 129, 133, 136, 139, 146, 160, 161, 174
 Отладка 73

Отношение 126, 132, 217, 225
 Очистка памяти 17, 275
 — регистров 243

Память, см. Запоминающее устройство
 Параметр ввода 141
 — вывода 141
 — обращения к подпрограмме 292, 310, 312

— цикла 117, 146, 199
 Переадресация 110, 119, 185, 199
 Перевод чисел в другую систему счисления 24, 280

Передача управления 16, 71, 85, 196, 199, 266, 267, 269, 293

Переключатель 174
 Переключательный список 174

Переменная 49, 78, 117, 129, 131, 135, 141, 146, 160, 171

— команда 110, 115, 118, 183, 184, 195, 199

Переменная с индексами 129

Переполнение 246

Перфокарта 11, 17, 72, 255, 262, 267

Перфолента 11, 17

Перфорация 17, 257, 262

Печатающее устройство 18, 261, 270

Плавающая запятая 32, 286

Подготовительная часть цикла 83, 110

Подготовка данных к вводу 72, 258

Подпрограмма 72, 99, 292

Представление команд 38, 65, 66

— чисел 20, 31, 39, 64, 90, 91, 242, 258, 280

Порядок числа 23, 33, 35, 64, 242, 244

246, 251, 259, 286, 288

Приближенные числовые значения

АЛГОЛе 124, 127

Примемое приближение 242

Признак конца массива 255, 262, 265, 270

— модификации адреса 65, 73, 194

— числа 65, 69, 246, 275

Признак ω 67, 72, 85, 89, 198, 216

Присваивание 49, 74, 135, 141, 146, 160, 172, 224

Программа 11, 12, 72, 136, 140, 151

Программный массив 107, 176, 262

Простая переменная 129

Простое выражение 133, 134

Простой цикл 84

Процедура 157, 172, 277, 293, 299

Прямой код 34

Пульт управления 12, 17, 275

Пусковая программа 18, 273

Пустой оператор 150

Рабочая позиция перфокарты 255

— часть цикла 87, 119, 179

— ячейка 74, 78, 180, 187, 307, 315

Разветвление 48, 85

Размер массива по отдельному измерению 177

Размерность массива 129, 176

Разметка магнитной ленты 263, 270

Размещение по столбцам 176

— по строкам 176

Разряд 21, 31, 64, 216, 221, 243

Распределение памяти 75, 108, 170, 176, 296, 299, 303, 309

Реализация алгоритма 47

Регистр 15, 21, 33, 194, 243

— адреса, см. Индекс-регистр

— — команды 15, 198

— команды 16, 198

— результата 15

Регистр ω 194, 198, 216

Рекурсивная функция 55

Рекурсивное обращение 173

Сбой 18, 276

Связь модулей 307, 314, 304

Сегмент массива 130

Символические обозначения 67, 76, 78, 81, 109, 181, 294, 298

Система команд 64

Скобки в АЛГОЛе 127, 129, 130, 131, 134, 136, 143, 151, 162

Сложный цикл, см. Кратный цикл

Служебное слово 125

Совокупность спецификаций 160, 161

— формальных параметров 160

Составной оператор 143, 150

Спецификатор 161

Спецификация 159, 161

Список значений 160

— цикла 146

Стандартная функция 128

Старшинство операций 131

Структурный цикл 85, 108, 109, 117

Сумматор 15, 243
 — адресов 16, 198
 Схема Горнера 26, 77, 103, 113
 Текст 24, 49, 51, 240, 258, 261
 Тело процедуры 158, 160, 172
 — цикла 145, 149, 151
 Тип объекта в АЛГОЛе 125, 129, 132, 135, 172
 Транслятор 124
 Указатель переключателя 175
 — функции 158, 172
 Управляющий код на перфокарте 255
 Условие 48, 52, 70, 88, 133, 142, 224, 227
 Условное выражение 133, 139, 143, 224, 226
 — число 269
 Условный адрес 296
 — оператор 142, 151, 224, 226
 — переход 71, 196, 266
 Устройство ввода, см. Вводное устройство
 — вывода, см. Выводное устройство
 — подготовки данных 257
 — управления 12, 15
 Фактический параметр 158, 162, 170, 172
 Фиксированная запятая 31, 280
 Формальный параметр 158, 160, 170

Формирование команд 112, 119, 179, 184, 220, 240, 311, 315
 Формирующая часть модуля 312
 Функция в АЛГОЛе 172

Хорошее приближение 242

Целые числа в АЛГОЛе 127
 — числовые значения в АЛГОЛе 124, 127
 Цикл 54, 83, 145, 179
 — с переадресацией, см. Структурный цикл
 Циклическая сумма 264, 265

Числовые значения в АЛГОЛе 124, 127
 Чистка циклов 103, 188, 208
 Читающее устройство 17, 255

Щуп 233

Экономия 52, 74, 80, 111, 116, 122, 171, 188, 189, 209, 226, 303, 309, 310
 Элементарное действие алгоритма 57
 Элементарный цикл машины 13, 198
 Элемент списка цикла 146

Язык машины 64
 Ячейка 14, 21, 32, 38, 64, 67

УКАЗАТЕЛЬ ОПЕРАЦИЙ МАШИНЫ М-20

Первая цифра кода операции	Вторая цифра кода операции							
	0	1	2	3	4	5	6	7
0	$ПЧ_{70}$	$67, +_{243}$	$67, -_{243}$	$68, - _{249}$	$68, /_{249}$	$67, \times_{248}$	$+АП_{68}$	$+Ц_{264}$
1	$ВВ_{267}$	$1И < A_{197}$	$И < A_{197}$	$+М_{69}$	$\leftarrow МА_{223}$	$70, \neq_{220}$	$ПВ_{71}$	
2		$+a_{249}$	$-a_{249}$	$ - a_{249}$	$/a_{251}$	$\times a_{249}$	$+ПП_{251}$	$-Ц_{264}$
3	$ВВЕО_{267}$	$1И \geq A_{197}$	$И \geq A_{197}$	$-М_{69}$	$\leftarrow МП_{224}$	$\neq a_{265}$	$П1_{72}$	
4	$И < K_{197}$	$+_{249}$	$-_{249}$	$, - _{249}$	$68, \vee_{249}$	$, \times_{249}$	$-АП_{68}$	$\times b_{250}$
5	$Ма_{269}$	$0И < A_{197}$	$АИ_{195}$		$\leftarrow СА_{221}$	$\wedge_{219}$	$ПБ_{71}$	
6	$И \geq K_{197}$	$, +a_{249}$	$, -a_{249}$	$, - a_{249}$	$\vee a_{251}$	$, \times a_{249}$	$-ПП_{252}$	
7	$Мб_{269}$	$0И \geq A_{197}$	$КИ_{196}$		$\leftarrow СИ_{222}$	$\vee_{220}$	$ПО_{72}$	$СТОП_{72}$

В клетках таблицы указаны: сверху — символическое обозначение кода операции, принятое в этой книге, внизу — номера страниц, где описано выполнение операции. Незаполненные клетки соответствуют операциям, не упомянутым в книге.

Святослав Сергеевич Лавров

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ

М., 1973 г., 352 стр. с илл.

Редактор *Г. Я. Пирогова*

Технический редактор *В. Н. Кондакова*

Корректоры *О. А. Бутусова, М. Л. Медведская*

Сдано в набор 15/I 1973 г. Подписано к печати 27/VII 1973 г. Бумага 60×90¹/₁₆, тип. № 2. Физ. печ. л. 22. Условн. печ. л. 22. Уч.-изд. л. 22,89.

Тираж 75 000 экз. Т-11161. Цена книги 94 коп. Заказ № 658.

Издательство «Наука»

Главная редакция физико-математической литературы

117071, Москва, В-71, Ленинский проспект, 15

Ордена Трудового Красного Знамени Ленинградская типография № 1 «Печатный Двор» имени А. М. Горького Союзполиграфпрома при Государственном комитете Совета Министров СССР по делам издательства, полиграфии и книжной торговли. Ленинград, Гатчинская ул., 26.