SYSTEMATIC PROGRAMMING. AN INTRODUCTION

NIKLAUS WIRTH

Eidgenössische Technische Hochschule
Zürich, Switzerland

PRENTICE-HALL, INC.
ENGLEWOOD CLIFFS, NEW JERSEY
1973

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Н. Вирт

СИСТЕМАТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ. ВВЕДЕНИЕ

Перевод с английского
Вик. С. Штаркмана

Под редакцией
Ю. М. Баяковского

Издательство «Мир»
Москва 1977

Посвящается памяти
Джорджа Э. Форсайта

ПРЕДИСЛОВИЕ

Мой главный замысел — представить программирование как искусство или технику конструирования и формулирования алгоритмов, и причем представить систематически как самостоятельную дисциплину. Алгоритмы в самом общем смысле — это рецепты, описывающие некоторые классы процессов обработки данных и управляющих процессов. Их следует представлять себе как некоторые жесткие структуры, состоящие из блоков, построенных логично, надежно и целесообразно.

Студентов нужно обучать конструированию алгоритмов методично и систематически, знакомя их с задачами и приемами, типичными для программирования и не зависящими от конкретных приложений. По этой причине никакая специальная область приложений в книге не выделена как самоцель, а упражнения и примеры отобраны так, чтобы иллюстрировать общезначимые проблемы и методы их решения. По той же причине особо не выделяется нотация или язык программирования: язык — наш инструмент, а не самоцель. Достижение совершенства в использовании специфических особенностей какого-либо конкретного языка не должно стать основной целью курса программирования. Скорее наоборот, язык должен отражать фундаментальные и наиболее важные концепции алгоритмов в очевидной, естественной и легко воспринимаемой форме. Следует также учитывать внутренние свойства и ограничения цифровых вычислительных машин.

Касаясь проблемы надежности программ, я попытался включить в книгу описание основных идей и методов проверки правильности (верификации) программ. Конечно, при программировании стремятся сформулировать классы вычислительных процессов в виде алгоритмов. Однако опытный программист должен уметь продемонстрировать, что его продукция соответствует требованиям во всех возможных случаях. Но в обычно используемых методах тестирования (отладки) программ исследуются только отдельные вычисления, а не весь класс вычислений, которые можно выполнить с помощью этой программы. Чтобы сформулировать общезначимые утверждения о программе, необходимы аналитические методы верификации программ.

Изучение элементарных методов верификации программ требует от студента несколько большей способности к абстракции. Поэтому были некоторые сомнения относительно целесообразности включения этой темы в наш элементарный учебник. Но я пришел к твердому убеждению, что те основные понятия, которые отобраны для этой книги (по существу только идеи утверждений и инвариантов), имеют такое фундаментальное значение, что их нельзя отнести ни к каким другим учебным курсам, кроме как к самым начальным. Я готов также оспаривать мнение, что эти понятия относятся скорее к теории, чем к практике. Вопросы надежности программы важны именно для практики, а не для теории. Концепция верификации программы по существу является краеугольным камнем для более глубокого понимания алгоритмов, без нее программист не имел бы никакого другого инструмента, кроме своей собственной сомнительной интуиции.

Эта книга написана для тех, кто рассматривает курс систематического конструирования алгоритмов как часть своего общего математического образования, а не для тех, кому изредка приходится составлять программу и решать задачи с помощью вычислительной машины.

В основе нотации, применяемой в этой книге, лежит язык программирования Алгол-60. Но я не использовал Алгол-60 буквально, поскольку теперь вычислительные машины, а вместе с ними и программирование, имеют намного большую сферу применения, чем 12 лет назад. Поэтому во вводном курсе программирования нельзя целиком ориентироваться только на какое-либо одно приложение. Алгол-60 был создан главным образом в расчете на вычислительную математику. Впоследствии его использовали для программирования задач в других областях, для которых более подходили иные принципы структуризации и в которых преобладали другие концепции, что приводило к искажению и ломке языка. При обучении всегда следует избегать применения инструментов не по назначению и, конечно, ни в коем случае не превозносить подделки.

Я стремлюсь при обучении использовать такую нотацию, которая позволяет постепенно и систематически вводить новые структуры процессов и данных. Это желание основывается на наблюдении того факта, что большинство людей навсегда привязываются к языку, который они изучали первым. Причина не только в часто упоминаемой инерции человеческого мышления, но еще и в том, что «первый» язык представляет собой удобную форму, с помощью которой абстрактная мысль получает конкретное воплощение. С первым языком не только усваивается словарь и набор грамматических правил, но также открываются ворота в новую область мысли. Поэтому выбор этого языка должен быть серьезно обоснован. К сожалению, большинство широко используемых языков программиро-

вания совершенно неудовлетворительны, если речь идет о методике и системе обучения.

При изучении этого курса потребуются лишь элементарные знания из математики. Предполагается, в частности, что студенты знакомы с элементами математической логики и понятием математической индукции. Знакомство с математическим анализом вряд ли необходимо; исключение составят некоторые примеры и упражнения, которые при желании можно опустить.

Важно, чтобы студенты активно занимались решением упражнений. Программирование по своей сути является конструктивной и синтезирующей дисциплиной, поэтому в ней нельзя достигнуть совершенства, ограничиваясь одними рассуждениями. Я предпочитаю простые задачи, которые формулируются ясно без излишних подробностей. Конечный результат должен выражаться в простой форме без сложного математического формализма. Упражнения должны помочь студенту ближе познакомиться с применением рассмотренных понятий и методов. Они не содержат хитроумных загадок, разгадывание которых требует большого времени и опыта. Упражнения в конце каждой главы играют также роль примеров, поэтому их можно произвольно изменять и усложнять.

Успех курса программирования в значительной мере зависит от организации и доступности вычислительного центра. Если же определенные минимальные требования не выполняются, то в результате курс будет иметь ничтожный эффект. Во-первых, организация работы на машине должна быть такой, чтобы малые работы выполнялись в короткое время. Работы, требующие всего лишь пару секунд процессорного времени и выдающие несколько десятков строчек на печатающее устройство, не должны выполняться более чем за четверть часа. Во-вторых, компилятор при всех обстоятельствах должен выдавать вразумительный ответ. В частности, новички редко получают в качестве ответа ожидаемые результаты вычислений; как правило, компилятор сообщает им информацию о допущенных ошибках. Система должна формулировать эти сообщения либо на естественном языке, либо на используемом языке программирования. Не должно быть никаких непонятных или случайных сообщений операционной системы или, что еще хуже, аварийных выдач в виде колонок восьмеричных или шестнадцатеричных цифр. Число команд для пользователя, обращающегося к услугам операционной системы, также должно быть сведено к минимуму.

Эта книга возникла в результате обработки конспекта лекций, прочитанных в Станфордском университете и Федеральном технологическом институте (ETH) в Цюрихе. Поэтому вряд ли удастся упомянуть всех, кто помог мне своими советами и идеями. Однако я выражаю особую благодарность моим коллегам Э. В. Дейкстре (Эйндховен), К. А. Р. Хоору (Белфаст) и П. Науру (Копенгаген), работы которых не только повлияли на содержание книги, но на

программирование вообще. С особой признательностью я вспоминаю также мои беседы с Х. Рутисхаузером, родоначальником идеи языков программирования и соавтором Алгола-60. Он, по-видимому, оказал наибольшее влияние на эту работу. Я благодарен также миссис А. Форсайт, которая внимательно прочла рукопись. Наконец, я обязан моим сотрудникам У. Амману, Э. Мармьер и Р. Шилду за их героические усилия при реализации компилятора с языка программирования Паскаль. В результате этих усилий нотация, используемая в этой книге, годится не только для выражения абстрактных алгоритмов, но также и для написания эффективных и надежных программ для реальной вычислительной машины.

Н. Вирт

1. ВВЕДЕНИЕ

За последнее десятилетие вычислительная машина стала незаменимым инструментом в управлении и экономике, в промышленности и научных исследованиях. Решение многих задач, возникающих в самых различных сферах человеческой деятельности, было бы невозможно без применения ЭВМ. Вычислительная машина — это автомат, который производит вычислительную работу в соответствии с точно предписанными правилами. Обычно она выполняет только ограниченный набор элементарных команд, которые ей «понятны» и которым она способна повиноваться, но выполняются эти команды с поразительной направленностью и надежностью. Вычислительная машина способна выполнять чрезвычайно длинные последовательности команд, содержащие почти бесконечные комбинации элементарных действий. Именно это свойство определяет ее мощь и широкую применимость. Деятельность по объединению таких цепочек команд в «рецепты», описывающие некоторые классы вычислительных процессов, называется *программированием*. Но фундаментальные идеи, лежащие в основе конструирования программ, можно объяснить и понять, вообще не упоминая при этом вычислительную машину.

Программирование — дисциплина удивительно многоплановая; во-первых, оно содержит множество нетривиальных проблем и поэтому открыто для систематических методов математического анализа, а во-вторых, само по себе оно является вызовом человеческому интеллекту.

Но методической стороне программирования уделялось очень мало внимания по той простой причине, что оно только тогда приводит к интересным исследованиям и сложным проблемам, требующим твердого теоретического фундамента и систематического подхода,

когда программы достигают определенной сложности и размеров (т. е. когда они содержат тысячи и даже миллионы команд). До появления вычислительной машины не существовало «раба», способного надежно выполнить такую длинную последовательность команд с абсолютным и беспрекословным повиновением; поэтому-то и не было стимула для создания таких программ. Только современная вычислительная машина сделала программирование объектом самого пристального и всестороннего внимания.

2. ОСНОВНЫЕ ПОНЯТИЯ

В этой главе вводятся некоторые важные понятия программирования. Они относятся к разряду фундаментальных понятий, и их нельзя определить формально с помощью других понятий. Поэтому мы их объясняем на примерах.

Самым важным понятием является понятие *действия*. Здесь под действием понимается нечто, что имеет конечную продолжительность и приводит к желаемому и совершенно определенному *результату*. Каждое действие предполагает наличие некоторого *объекта*, над которым это действие совершается и по *изменению состояния* которого можно судить о результате этого действия. Каждое действие должно быть таким, чтобы его можно было описать с помощью некоторого *языка* или системы формул. Это описание называется *инструкцией*.

Если действие можно разложить на составные части, то оно называется *процессом* или *вычислением*. Если эти части во времени следуют строго друг за другом и никакие две части не выполняются одновременно, то процесс называется *последовательным*. А это означает, что инструкцию, описывающую процесс, можно разбить на составные части; тогда она называется *программой*. Таким образом, программа состоит из набора инструкций, текстуальная упорядоченность которых, вообще говоря, не совпадает с порядком выполнения соответствующих действий.

То, что выполняет эти действия согласно заданным инструкциям, называется *процессором*. Это более или менее нейтральное слово никак не определяет, кто является исполнителем — человек или автомат. В самом деле, программы (коль скоро они записаны на языке, который точно определен) имеют смысл безотносительно к специфике процессора. Вообще говоря, программиста не интересует индивидуальность процессора. Для него важно только быть уверенным, что процессор понимает язык, на котором написаны программы, поскольку предполагается, что программы содержат правила поведения процессора. Следовательно, программисту необходимо знать типы инструкций, которые процессор способен понимать и выполнять и которые программист должен писать в точном соответствии с языком.

Каждое действие требует некоторого объема *работы*, который зависит от процессора. Объем работы можно характеризовать временем, которое затрачивает процессор на выполнение действия. Меру времени в свою очередь можно более или менее прямо перевести в меру стоимости. Опытный программист всегда учитывает возможности имеющихся в его распоряжении процессоров и затем выбирает то решение, которое минимизирует стоимость.

Поскольку данная книга главным образом касается составления программ для процессоров-автоматов (вычислительных машин), остальная часть главы посвящается некоторым основным характеристикам, общим для всех вычислительных машин. Но прежде чем приступить к изложению весьма общего взгляда на вычислительные машины, мы рассмотрим два простых примера, иллюстрирующих только что введенные понятия.

Пример: Умножение
Нам дана инструкция:
Умножить два натуральных числа x и y и полученное произведение обозначить через z.

Если наш процессор понимает эту инструкцию, т. е. он знает, что означает «натуральное число» и «умножить», то дальнейшая детализация не требуется.

В интересах дискуссии предположим, однако, что наш процессор

(a) не понимает предложений естественного языка, а воспринимает только некоторые формулы, и

(b) не умеет умножать, а умеет только складывать.

Прежде всего мы заметим, что объектами вычисления являются натуральные числа. Однако предполагается, что в программе не задаются эти числа; в ней только определяется в общем виде *характер поведения* при умножении произвольной пары натуральных чисел. Поэтому вместо чисел мы используем имена, обозначающие изменяемые объекты, которые называются *переменными*. В начале каждого процесса этим переменным должны быть присвоены конкретные значения. Это *присваивание* — одно из самых фундаментальных действий в вычислительных процессах, выполняемых вычислительной машиной.

Переменную можно сравнить с классной доской: ее значение можно «читать» столько раз, сколько пожелаешь, его можно стереть и на том же месте написать другое значение. Повторная запись приводит к тому, что прежнее значение теряется. В дальнейшем присваивание значения w переменной v мы будем обозначать следующим образом:

$$v := w \tag{2.1}$$

Символ $:=$ называется *оператором присваивания*.

Теперь инструкцию нашего примера можно записать иначе:

$$z := x*y \tag{2.2}$$

Если эта инструкция распадается на последовательность сложений, следующих во времени друг за другом, то действие умножения становится последовательным процессом, а инструкция (2.2) принимает вид программы. Пока мы ограничимся следующим неформальным описанием программы:

$$\begin{array}{ll} \text{Шаг 1:} & z := 0 \\ & u := x \\ \text{Шаг 2:} & \text{повторять инструкции} \\ & z := z+y \\ & u := u - 1 \\ & \text{до } u=0 \end{array} \tag{2.3}$$

Выполняемый этой программой процесс при условии, что переменным x и y заданы начальные значения, можно представить в наглядной форме, если записать значения, которые присваиваются переменным u и z во время счета. Для $x=5$ и $y=13$ мы получим таблицу (2.4).

Шаг	Значения переменных z	u
1	0	5
2	13	4
2	26	3
2	39	2
2	52	1
2	65	0

(2.4)

Процесс заканчивается на шаге 2, как только $u=0$. В этот момент значением z будет окончательный результат $65=5*13$. Такая таблица называется *трассировочной таблицей*. Заметим, что последовательное перечисление значений не означает, что эти значения сохраняются; наоборот, каждая переменная в любой момент времени имеет ровно *одно* значение. Это следует из того, что при присваивании прежнее значение переменной стирается.

Объектами этого вычисления являются числа. Для того чтобы можно было выполнять операции над числами, нужен специальный способ записи для представления этих чисел. Следовательно, при выполнении вычисления *выбор способа записи* неизбежен. (Однако программа, вообще говоря, имеет смысл безотносительно к способу записи и представлению ее объектов.) Существенно также различать объекты и их представления, если даже это абстрактные объек-

ты, такие, как числа. Например, в машинах числа обычно представляются состоянием элементов магнитной памяти, но хотелось бы иметь возможность формулировать процессы, которые оперируют с числами и выполняются на вычислительной машине, не в терминах состояний магнитной памяти. Для подтверждения этой мысли и демонстрации того, как тот же вычислительный процесс может быть описан с помощью другой системы обозначений, вместо (2.4) мы приводим таблицу (2.5), в основе которой лежит римская система счисления.

Шаг	Значения переменных z	u
1	0	V
2	XIII	IV
2	XXVI	III
2	XXXIX	II
2	LII	I
2	LXV	0

(2.5)

Пример: Деление
Дана инструкция:

Разделить натуральное число x на натуральное число y и обозначить целое частное через q, а остаток через r.

Говоря более точно, должны выполняться следующие соотношения:

$$x=q*y+r \text{ и } 0\leqslant r<y \qquad (2.6.1)$$

Если ввести оператор **div**, то это вычисление можно описать более формально следующей инструкцией присваивания:

$$(q,\ r) := x \ \textbf{div}\ y \qquad (2.6.2)$$

Чтобы еще раз продемонстрировать, как инструкция может быть преобразована в программу, мы предположим, что процессор, для которого предназначена эта программа, не умеет делить, т. е. не имеет команды **div**. Поэтому нужно деление разложить на последовательность вычитаний делителя y из делимого x. Тогда количество вычитаний будет искомым частным q.

Шаг 1: $q := 0$
$r := x$
Шаг 2: пока $r\geqslant y$ повторять (2.7)
$q := q+1$
$r := r-y$

Имена x и y вновь обозначают *константы*, которые представляют полученные ранее фиксированные значения, а q и r обозначают *переменные*, которым присваиваются целочисленные значения. Процесс, описанный с помощью программы (2.7), для значений $x=100$ и $y=15$ можно изобразить трассировочной таблицей (2.8).

Шаг	Значения переменных q	r
1	0	100
2	1	85
2	2	70
2	3	55
2	4	40
2	5	25
2	6	10

(2.8)

Процесс заканчивается, как только $r<y$. Полученные результаты ($q=6$ и $r=10$) удовлетворяют соотношениям (2.6.1):

$$100=6*15+10 \text{ и } 0\leqslant 10<15 \qquad (2.9)$$

Эти два примера описывают последовательные процессы, в которых присваивания выполняются строго последовательно во времени. В дальнейшем мы ограничимся рассмотрением только последовательных процессов и под словом «процесс» далее всегда будем подразумевать *последовательный процесс*. Мы умышленно не рассматриваем непоследовательные процессы не только потому, что обычные вычислительные машины последовательно выполняют операции, но прежде всего потому, что конструирование непоследовательных программ (или систем последовательных, но взаимозависимых и параллельно выполняемых программ) является весьма деликатной и трудной задачей, требующей полного овладения искусством составления последовательных алгоритмов.

Эти два примера показывают также, что каждая программа описывает последовательность преобразований состояния набора ее переменных. Если одна и та же программа выполняется дважды с различными начальными значениями (x и y), то было бы ошибкой говорить, что эти два процесса или вычисления одинаковы. Однако несомненно, что *характер поведения* у них одинаков. Описание характера поведения безотносительно к какому-либо конкретному процессору обычно называется *алгоритмом*. Термин *программа* соответственно используется для алгоритмов, которые сконструированы так, чтобы они были приемлемы для совершенно определенного типа процессоров. Различие между *общим* (иногда говорят абстрактным) *алгоритмом* и *программой машины* состоит главным образом в том, что в последней правила поведения должны быть специфици-

рованы до мельчайших подробностей и она должна быть составлена в точном соответствии с правилами записи. Это объясняется тем, что машина имеет ограниченный набор команд, который она способна понимать и выполнять, обусловлено ее абсолютным послушанием, основанным на полном отсутствии с ее стороны какой-либо критической позиции. Эти особенности вычислительной машины вызывают нарекания со стороны большинства начинающих программистов, поскольку требуется педантичная точность и скрупулезное внимание, когда имеешь дело с вычислительными машинами. В самом деле, даже тривиальная описка в программе может привести к совершенно непредсказуемому и «бессмысленному» поведению машины. Профессионалы также недовольны столь явным отсутствием какого-либо «здравого смысла», на который программист мог бы положиться (всякий раз, когда ему изменяет собственный разум). Поэтому были предприняты усилия, имеющие целью устранить этот кажущийся недостаток. Опытный программист, однако, умеет пользоваться рабской покорностью машины, поскольку в этом случае он может задавать любой даже самый «необычный» характер поведения. Очевидно, нельзя требовать того же от человека, выступающего в роли процессора, ибо он привык каждую инструкцию интерпретировать приблизительно, подгоняя ее к правдоподобному и удобному для себя виду.

3. СТРУКТУРА ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Чтобы конструировать программы для вычислительной машины, программист должен прежде всего знать свой инструмент. Чем тщательнее он изучил процессор, тем больше у него возможностей учесть частные особенности процессора при переводе своих алгоритмов в программы. В то же время, чем больше алгоритм «подстраивается» к процессору, тем большие усилия требуются для получения программы. Обычно приходится искать такое решение, когда усилия, затрачиваемые на составление программ, остаются в разумных пределах, но обеспечивается достаточно высокое качество (т. е. эффективность) получаемых программ. Для достижения этого компромисса программисту необходимо знать такие способы «подгонки» программы к своему процессору, которые сравнительно легко осуществимы и в то же время дают ощутимый эффект. Для этого важно учитывать наиболее существенные характеристики машин, отвлекаясь от сугубо специальных их черт и особенностей.

Во всех современных цифровых вычислительных машинах можно выделить две главные компоненты.

1. *Запоминающее устройство* (или *память*). В памяти в закодированном виде содержатся объекты, с которыми осуществляются различные операции. Эти закодированные объекты называются *данными*. К основным характеристикам памяти относят ее емкость (размер) и скорость, с которой данные можно заносить в память и извлекать их из нее. В любом случае память имеет *конечную* емкость.
2. *Процессор* (арифметическое и логическое устройство). Это устройство выполняет операции сложения, умножения, сравнения и т. д. Во время работы данные извлекаются (читаются) из памяти, а результаты заносятся (записываются) в память. В каждый отдельный момент в процессоре находятся только те данные, с которыми он непосредственно оперирует, т. е. очень небольшое количество операндов. Элементы его собственной памяти называются *регистрами*. Все данные, в которых нет непосредственной необходимости, находятся в памяти, играющей, таким образом, роль «камеры хранения».

Пример: Вычисление выражения

Чтобы вычислить арифметическое выражение, имеющее несколько операндов и промежуточных результатов, мы опять-таки применяем технику разложения сложной задачи на последовательность более простых задач. Это приводит к тому, что при выполнении каждой отдельной арифметической операции операнды и результат располагаются в регистрах процессора. Вычисление выражения

$$a*b+c*d \tag{3.1.1}$$

сводится к выполнению следующих более простых команд:

$$\begin{array}{l} \mathrm{R1} := a \\ \mathrm{R2} := b \\ \mathrm{R1} := \mathrm{R1}*\mathrm{R2} \\ z := \mathrm{R1} \\ \mathrm{R1} := c \\ \mathrm{R2} := d \\ \mathrm{R1} := \mathrm{R1}*\mathrm{R2} \\ \mathrm{R2} := z \\ \mathrm{R1} := \mathrm{R1}+\mathrm{R2} \end{array} \tag{3.1.2}$$

Здесь R1 и R2 обозначают регистры процессора, а z — промежуточный результат вычислений, временно помещаемый в память. Конечный результат оказывается в регистре R1.

Вычисление выражения таким образом вылилось в короткую программу, которая содержит команды или инструкции трех видов:

(a) команды выборки операндов из памяти,
(b) (арифметические) операции, которые выполняются только с содержимым регистров, и
(c) команды занесения результатов в память.

Этот метод разложения инструкций на элементарные шаги и временного хранения промежуточных результатов в памяти приводи к тому, что одни и те же вычислительные процессы могут выполня ться как на относительно простых, так и на очень сложных вычис лительных машинах, с той лишь разницей, что первым для вычислений требуется больше времени.

В этом методе и заключается сущность программирования для цифровых вычислительных машин; именно он лежит в основе применения относительно простых механизмов для решения зада огромной сложности. Чтобы успешно завершить вычисление, требующее миллиарда отдельных шагов (их операндами непременно будут результаты вычисления предыдущих шагов), необходим конечно, абсолютно надежный процессор, работающий с достаточно высокой скоростью. Реализацию таких процессоров следует отнести к величайшим достижениям современной техники.

Пример вычисления выражения свидетельствует также о необходимости тесной взаимосвязи между процессором и памятью, так как обмен информацией между ними довольно интенсивный. К объектам, которые хранятся в памяти, мы адресуемся с помощью различных имен (типа a, b, z, ...). Следовательно, организация памяти должна чем-то напоминать систему абонентных почтовых ящиков. То есть объекты содержатся в множестве отдельно идентифицируемых *ячеек памяти*, каждая из которых имеет свой персональный *адрес*. Каждое обращение к памяти должно сопровождаться указанием адреса соответствующей ячейки.

Ячейки памяти машины имеют сходство с почтовыми ящиками, с которыми мы имеем дело в повседневной жизни, в том смысле, что они содержат и хранят объекты. Но на этом аналогия и кончается. Способность памяти вычислительной машины хранить данные основывается совсем не на том, что объекты физически присутствуют в ней, а на том, что состояние самой ячейки соответствует некоторому *закодированному* представлению объекта. Следовательно, ячейка должна принимать некоторое количество *дискретных состояний*. Трудно изготовить компоненты, способные принимать любое из множества отличных друг от друга состояний и оставаться в нем сколь угодно долго. Однако можно без особых затруднений строить запоминающие элементы, которые имеют только два различных состояния. Такие элементы называются двоичными запоминающими элементами. Если сгруппировать n двоичных запоминающих элементов, то эта группа может принимать 2^n различных комбинаций состояний. Если эту группу рассматривать как некоторую неделимую компоненту, то она представляет собой ячейку памяти с 2^n возможными состояниями.

Пример: Кодирование объектов в виде групп двоичных цифр

Мы выберем позиционное представление натуральных чисел, включая 0. Число x кодируется следующей последовательностью из n двоичных цифр (называемых *битами*), т. е. нулей и единиц:

$$x\colon\ b_{n-1}\ldots\ b_1 b_0 \qquad (3.2.1)$$

причем правило кодирования задается следующим образом:

$$x = b_0 + 2b_1 + \ldots + 2^{n-1}b_{n-1} \qquad (3.2.2)$$

Разумеется, это правило не является единственно возможным, хотя во многих отношениях оно для нас более приемлемо. Достаточно сказать, что в конце концов то же самое правило лежит в основе представления арабских (десятичных) чисел, т. е.

$$x\colon\ d_{m-1}\ldots d_1 d_0 \qquad (3.3.1)$$

где

$$x = d_0 + 10 * d_1 + 10^2 * d_2 + \ldots + 10^{m-1} * d_{m-1} \qquad (3.3.2)$$

В следующей таблице приводится несколько примеров двоичного и десятичного кодирования (представления) чисел:

Двоичное	Десятичное
1101	13
10101	21
111111	63
1101011	107

(3.4)

Самым важным фактом, который мы извлекли из этого примера, является тот факт, что конечные запоминающие ячейки, т. е. ячейки, принимающие лишь конечное число различных состояний (собственно других и не существует в природе), способны запоминать числа только из некоторого *конечного диапазона значений*. Количество двоичных запоминающих элементов, сгруппированных в одну однозначно адресуемую ячейку памяти (слово), обычно называется *длиной слова*. Характеристики арифметического устройства существенно зависят от этой длины. Обычно длина слова n равна 8, 16, 32, 48 или 64, и соответственно допустимые множества различных значений равны 2^n.

Поскольку работа вычислительной машины целиком и полностью управляется программой, необходимо обеспечить легкий доступ машины к своей программе. Тогда возникает вопрос: где лучше всего хранить программу? Джону фон Нейману принадлежит замечательная идея (теперь она кажется тривиальной): поместить программу в память. Таким образом одна и та же память используется для хранения и объектов, и «рецептов» вычислительных процессов, т. е и данных, и программы.

Очевидно, что из самой концепции — *хранить программу в памяти* — следует, что команды также должны кодироваться. В нашем примере вычисления выражения каждая команда представляется *кодом операции* (задающим «чтение», «запись», «сложение», «умножение» и т. д.) и в некоторых случаях операндом. Если операнды представлять адресами ячеек памяти и если предположить, что адреса — это целые числа 0, 1, 2, ..., то проблема кодирования программ по существу решена; каждая программа может быть представлена последовательностью чисел (или групп чисел) и, следовательно, может храниться в памяти вычислительной машины.

Другим следствием идеи хранения программы в памяти является то, что каждая программа занимает некоторое количество ячеек, т. е. некоторое *пространство в памяти*. Это количество занимаемых ячеек пропорционально длине программы, а программа вместе с данными не может превышать общий объем памяти. Программист, следовательно, должен стремиться писать свою программу предельно лаконично. На концепции совместного использования памяти про-

граммой и данными основываются следующие важные возможности современных вычислительных машин:

1. Как только закончилось выполнение программы P, в память для следующего выполнения может быть загружена новая программа Q (гибкость, широкий диапазон применений).
2. Вычислительная машина может (в соответствии с некоторой программой) генерировать последовательность чисел, которая в дальнейшем будет рассматриваться и интерпретироваться как последовательность закодированных команд. Данные, полученные на первом шаге, становятся выполняемой программой на втором шаге.
3. Вычислительную машину X можно научить понимать последовательности чисел, в действительности представляющие программы, как данные и преобразовывать их (согласно некоторой транслирующей программе) в последовательности чисел, представляющие программы, закодированные для другой вычислительной машины Y.

4. СРЕДСТВА И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Вплоть до конца 50-х годов программирование сводилось к детальному кодированию длинных последовательностей команд (записанных вначале с применением каких-либо символических обозначений) двоичными, восьмеричными или шестнадцатеричными числами. Эта деятельность называлась *кодированием* в отличие от программирования, к которому относилась более трудная задача — конструирование алгоритмов. С появлением более быстрых вычислительных машин с бо́льшим объемом памяти трудности кодирования возросли и стало очевидно, что неразумно далее возлагать эту утомительную работу на человека.

1. Кодировщик был обязан учитывать в своих программах частные характеристики имеющейся в его распоряжении машины. Таким образом, он должен во всех подробностях знать машину, включая организацию ее процессора и ее систему команд. Обмен программами между различными машинами был невозможен, и большая часть приемов кодирования на одной машине совершенно не годилась для другой машины. В каждом институте разрабатывались свои собственные программы, и приходилось отказываться от них или кодировать их заново при замене старых машин на новые. Стало очевидным, что подгонка и настройка алгоритмов к частным характеристикам специфической вычислительной машины представляют собой неразумное использование человеческого интеллекта.
2. Благодаря тесной связи программиста с какой-либо одной машиной не только становилось возможным, но и поощрялось изобретение и применение разного рода трюков для достижения максимальной производительности машины. Поскольку всякого рода «трюкачество» считалось чуть ли не самой сутью программирования, программист тратил значительное время на получение самых «оптимальных» программ, проверить правильность которых зачастую было очень трудно. Практически не представлялось возможным понять принципы построения чужой программы; более того, часто трудно было объяснить принципы, положенные в основу своей собственной

программы! Этот артистизм кодирования в наше время, несомненно, утратил былую славу. Опытный программист сознательно избегает всякого рода трюков, предпочитая им систематические и ясные решения.

3. Программа, написанная на машинном языке (или, как говорят, в *машинных кодах*), содержит минимум избыточной информации, на основе которой можно было бы обнаружить формальные ошибки кодирования. В результате даже ошибки при набивке приводили к обескураживающим эффектам во время выполнения программы, и требовались значительные усилия и время для обнаружения таких ошибок.
4. Представление сложной программы в виде лишенной структуры линейной последовательности команд было самой неподходящей формой для человеческого восприятия и выражения. В дальнейшем мы покажем, что структуризация является принципиальным инструментом, помогающим программисту систематически синтезировать сложные программы, сохраняя полное о них представление.

Эти недостатки послужили стимулом для создания так называемых *языков программирования* «высокого уровня». Языки программирования высокого уровня стали средством, с помощью которого программирование ведется на некоторую идеализированную, гипотетическую машину, спроектированную не взирая на ограничения современной технологии, но учитывая традиционные способы и умение человека выражать свои мысли. Однако в этой ситуации у нас две машины: машина A, создание которой экономически оправдано, но которая не удобна в использовании, и машина B, которая вполне согласуется с человеческими нуждами, но существует только на бумаге. Роль моста через пропасть, которая разделяет эти два объекта, играет так называемое *программное обеспечение*. (В противоположность самой машине, которая называется *аппаратным обеспечением*.) Система программного обеспечения есть *программа* C для реально существующей машины A, которая дает возможность машине A переводить (транслировать) программы, написанные для гипотетической машины B, в ее собственные программы. Программа C называется транслятором или *компилятором*; она позволяет машине A выступать в роли идеализированной машины B.

Применение компилятора C, таким образом, освобождает программиста от необходимости рассматривать частные характеристики машины A. Но компилятор не освобождает его от обязанности постоянно учитывать тот факт, что в конечном итоге именно машина A будет выполнять его программу и что она имеет *определенные ограничения*, вытекающие из ограниченности ее скорости и памяти.

Обычно комбинированная аппаратно-программная система обрабатывает программу P в два этапа, которые во времени следуют

один за другим. На первом шаге программа P с помощью компилятора C переводится в форму, пригодную для интерпретации в машине A; этот шаг называется *компиляцией*. На втором шаге транслированная программа выполняется; этот шаг называется *выполнением*.

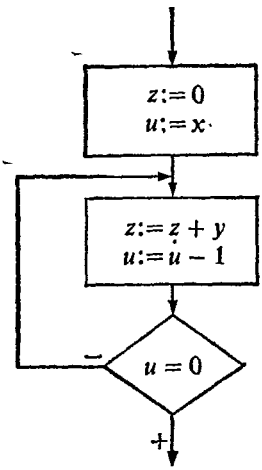
Компиляция: программа=компилятор C
входные данные=программа P на языке B
выходные данные=программа P на языке A

Выполнение: программа=P на языке A
входные данные=аргументы вычисления X
выходные данные=вычисленные результаты Y

5. НЕКОТОРЫЕ ПРИМЕРЫ ПРОСТЫХ ПРОГРАММ

В гл. 4 было ясно показано, почему программа должна состоять из инструкций, записанных таким образом, чтобы они были «понятны» вычислительной машине. Хотя мы не знаем, какого рода инструкции и формулы используются в языке программирования, но нам известно, что эти инструкции будут точно специфицировать желаемые действия. Это беспрекословное требование точности составляет, по-видимому, главное отличие общения между людьми от коммуникации между человеком и машиной. Работа с вычислительными машинами требует как точности, так и абсолютной ясности. Неопределенность и неоднозначность совершенно недопустимы.

Для изображения программ часто используется так называемая *блок-схема*. Ниже приводится программа (2.3) в виде блок-схемы.



(5.1)

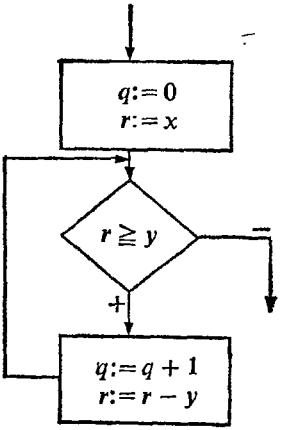
Это схематичное изображение явно показывает возможные последовательности шагов, наглядно иллюстрируя два вида инструкций:

(a) присваивания, обозначаемые прямоугольниками, и

(b) проверки условия, обозначаемые ромбами.

Шаг проверки условия имеет несколько возможных преемников. Если указанное отношение или условие выполняется, то реализуется путь, помеченный знаком +, в противном случае реализуется путь,

помеченный знаком —. Повторяемая часть образует *цикл*, т. е. замкнутую последовательность инструкций, причем по крайней мере одна из них является проверкой условия, определяющего окончание цикла. Программу (2.7) можно также изобразить в виде блок-схемы:



(5.2)

Программа задает характер поведения для неопределенного (часто даже бесконечного) числа возможных процессов. Один от другого процессы отличаются значениями переменных в соответствующих интервалах времени, и в частности *начальными значениями* или аргументами. Однако возникает вопрос: каким образом можно убедиться, что в любом из процессов, которые можно выполнить в соответствии с данной программой, получатся правильные результаты? Этот вопрос *правильности программ* является одним из самых основных, решающих и неизбежных вопросов программирования.

Правильность программ (2.3) и (2.7) демонстрировалась таблицами (2.4) и (2.8) для нескольких фиксированных пар значений x и y. Такой метод проверки правильности называется *тестированием программы*. Он заключается в том, что выбираются аргументы (x и y), процесс выполняется с этими выбранными аргументами и полученные результаты сравниваются с заранее известными правильными результатами. Такое экспериментальное тестирование повторяется для нескольких аргументов с помощью вычислительной машины, которая является идеальным для этого инструментом. Тем не менее этот традиционный способ дорог, обременителен и связан с большими затратами времени. Но, кроме того, он неудовлетворителен еще и потому, что рассеять все сомнения относительно правильности программы можно лишь тогда, когда будут выполнены все возможные вычисления, а не некоторые отобранные варианты. Если же ре-

зультаты всех процессов известны заранее, то едва ли есть смысл писать программу для вычислительной машины. Стоит ли говорить, что такое исчерпывающее тестирование программы просто невозможно. Предположим, например, что некоторая вычислительная машина выполняет сложение двух чисел за 1 мксек, и допустим также, что максимальное значение абсолютных величин, представимых в этой машине, равно 2^{60}. Тогда на 2^{2*60} различных сложений нужно было бы затратить

$$2^{2*60} * 10^{-6} \text{ сек} \approx 3.2 * 10^{22} \text{ лет}$$

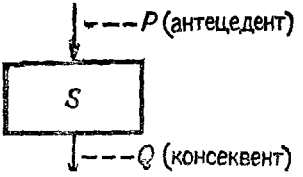
Поскольку такого рода исчерпывающее экспериментальное тестирование и бессмысленно, и невозможно, мы можем сформулировать следующее важное, основное правило:

Экспериментальное тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия.

Следовательно, необходимо абстрагироваться от индивидуальных процессов и постулировать некоторые общезначимые условия, которые можно вывести из характера поведения. Этот аналитический метод проверки называется *верификацией программ*. В отличие от тестирования программ, где исследуются свойства индивидуальных процессов, верификация имеет дело со свойствами *программы*.

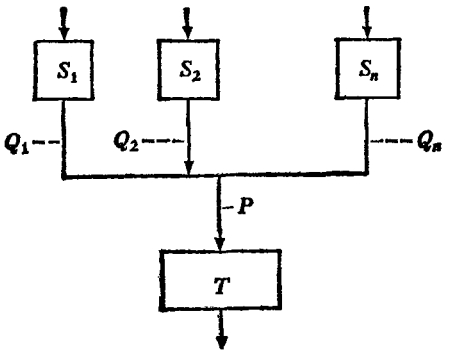
Верификация программ использует те же принципы, что и эмпирическое тестирование, и могла бы, следовательно, рассматриваться по аналогии с верификацией процесса. Но вместо записи индивидуальных значений переменных в трассировочной таблице мы после каждой инструкции задаем общезначимые диапазоны значений и отношения между переменными. Слово «общезначимые» следует понимать в смысле «допустимые для каждого процесса, соответствующего данной программе, и допустимые в той точке, где они встречаются, независимо от ранее выполненных инструкций». Сформулируем теперь четыре основных *правила аналитической верификации программ*.

1. Перед и после каждой инструкции задаются одно или несколько условий, которые должны удовлетворяться соответственно до и после выполнения этой инструкции. Эти условия называются *утверждениями*. Утверждения, которые предшествуют инструкции S, мы назовем *антецедентами* P, следующие за S — *сукцедентами* или *консеквентами* Q.



(5.3)

2. Если несколько путей на блок-схеме объединяются в один перед инструкцией T, то антецедент P инструкции T должен логически вытекать из консеквентов Q_i всех предшествующих инструкций S_i. Таким образом,

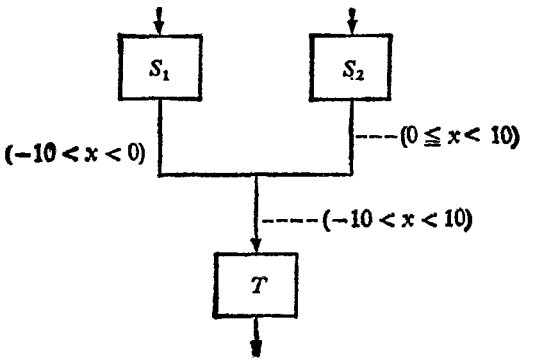


(5.4.1)

$$Q_i \supset P \text{ для } i=1 \ldots n$$

($Q \supset P$ читается как «Q влечет P».)

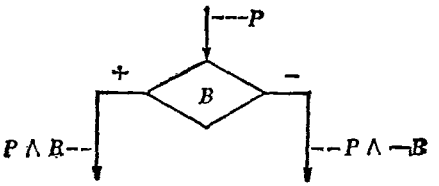
Пример:



(5.4.2)

3. Если утверждение P справедливо перед проверкой условия

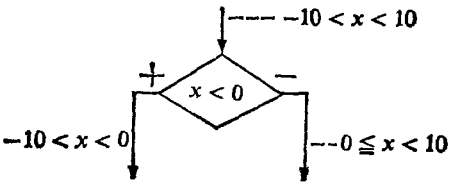
B, то двумя консеквентами будут



(5.5.1)

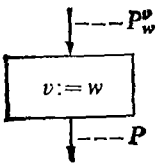
($P \wedge B$ читается как «P и B», $P \wedge \neg B$ — как «P и не B».)

Пример:



(5.5.2)

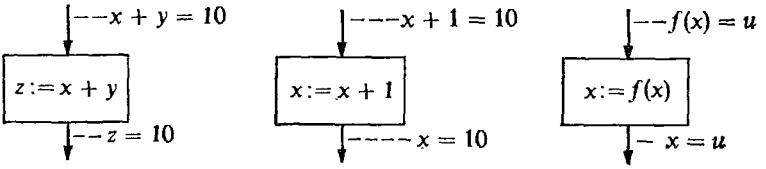
4. Если утверждение P справедливо после присваивания значения w переменной v, то антецедент присваивания получается подстановкой w вместо всех свободных вхождений v консеквента P.



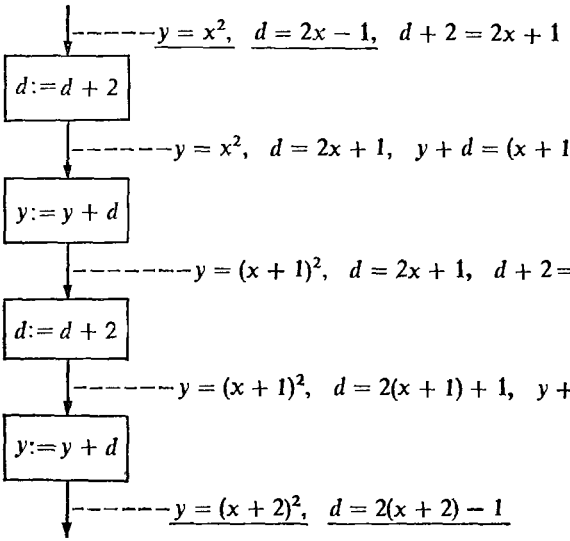
(5.6.1)

(Это правило можно также рассматривать как определение *эффекта присваивания*.)

Примеры:

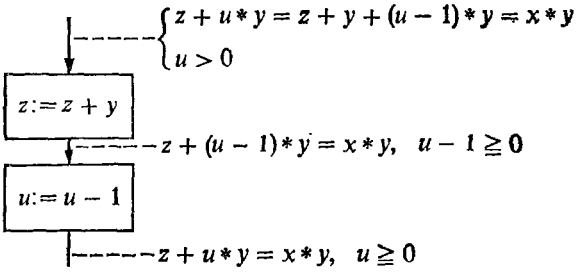


(5.6.2)

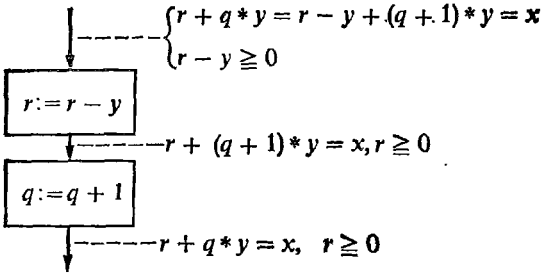


(5.7)

Теперь применим эти основные правила для проверки правильности программ (5.1) и (5.2), которые соответственно вычисляют произведение и частное двух натуральных чисел. Прежде всего мы получим следующие промежуточные формулы, применяя правила вывода для инструкции присваивания:

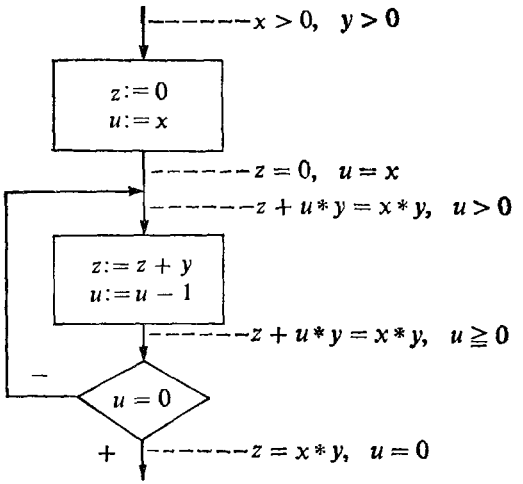


(5.8)

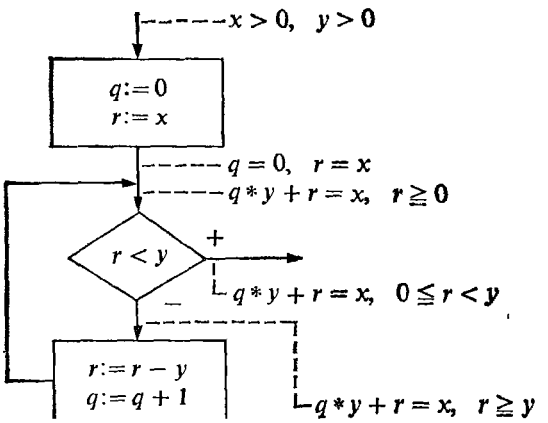


(5.9)

Две полные программы с соответствующими утверждениями показаны в блок-схемах (5.10) и (5.11).

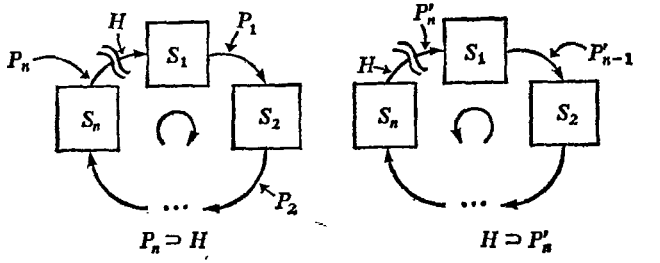


(5.10)



(5.11)

Вообще говоря, основываясь на правилах (5.3) — (5.7), очень просто определить утверждения в последовательности инструкций, если задан антецедент первой или консеквент последней инструкции. Однако возникают определенные трудности, если блок-схема содержит замкнутые циклические пути, т. е. если в программе имеются циклы. В этом случае лучше всего рассечь цикл в некотором месте и в точке разреза постулировать некоторую *гипотезу* H. Приняв эту гипотезу за исходную, можно построить вывод утверждений, двигаясь теперь уже по линеаризованной последовательности либо в прямом, либо в обратном направлении. Результирующее утверждение P_n в конце (т. е. в месте разреза) должно логически следовать из H (или быть следствием H) с тем, чтобы можно было склеить разрез. Если последнее условие не удовлетворяется, то следует принять другую гипотезу и всю процедуру повторить заново, как показано на схемах (5.12).



(5.12)

Разумно сделать разрез перед условием B, при выполнении которого заканчиваются повторения цикла. Тогда логическая комбинация условий H и B будет представлять собой консеквент всей группы повторяемых инструкций.

Такое утверждение, истинное независимо от числа ранее выполненных повторений, называется инвариантом цикла или просто *инвариантом*, так как оно представляет условие, которое не изменяется в ходе выполнения процесса. В программах (5.10) и (5.11) инвариантами соответственно будут

$$(z+u*y=x*y)\wedge(u\geqslant 0)$$

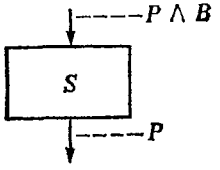
и

$$(q*y+r=x)\wedge(r\geqslant 0)$$

Поскольку повторения или циклы являются фундаментальными частями процессов и программ, рассмотренный выше прием мы можем сформулировать в виде правил вывода. Эти правила позволяют определить консеквент циклической инструкции, если задан ее антецедент и утверждения для повторяемой инструкции.

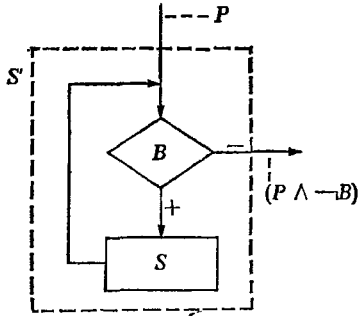
1. Если задано утверждение P, являющееся инвариантом инструкции S (т. е. задан как ее антецедент, так и ее консек-

вент), и, кроме того, известно, что



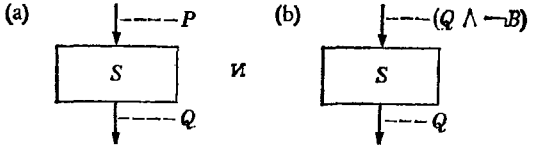
(5.13.1)

то можно построить следующее утверждение относительно повторяемой инструкции S':



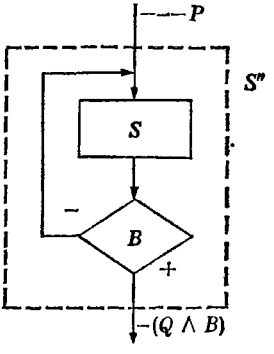
(5.13.2)

Если заданы две посылки для инструкции S



(5.14.1)

то справедливо следующее утверждение относительно повторяемой инструкции S'':



(5.14.2)

Заметим, что для циклов данного вида правило вывода можно применить лишь тогда, когда удовлетворяются *оба* условия.

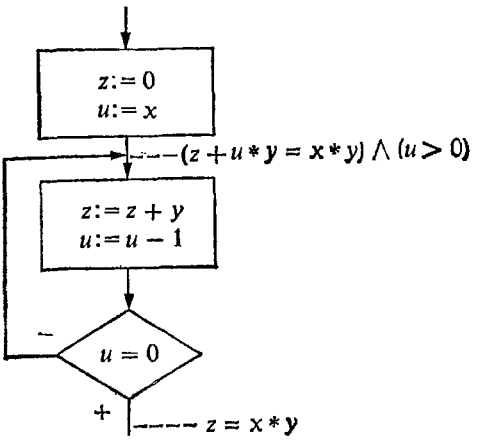
Такой цикл не без основания можно назвать более «опасным», так как часто ошибки программирования возникают из-за того, что программисты упускают из вида одну из этих посылок (обычно (а)). В сомнительных случаях рекомендуется использовать первый вид цикла, когда условие конца цикла B предшествует инструкции S.

По существу, эти указания следует рассматривать как неформальный подход к проблеме аналитической верификации свойств программ. Особенно важно понять трудности, касающиеся проблемы нахождения инвариантов. Следует усвоить всем программистам, что *явное указание инварианта для каждого цикла представляет наибольшую ценность в любой программной документации.* И даже в том случае, когда программой не пользуется никто другой, кроме ее автора, явное определение инвариантов помогает избежать многих ошибок, которые в противном случае либо удастся выявить тщательным тестированием, либо они навсегда останутся в программе. Столь же важно явно указывать диапазоны значений переменных, особенно диапазоны начальных значений, для которых сохраняются заданные свойства программы.

В заключение блок-схемы (5.15) и (5.16) показывают, из чего должна состоять необходимая и достаточная документация программы. Важно помнить, что программа может быть излишне документирована. Если в программе так много комментариев, что среди них трудно видеть сами инструкции, то такая программа бесполезна!

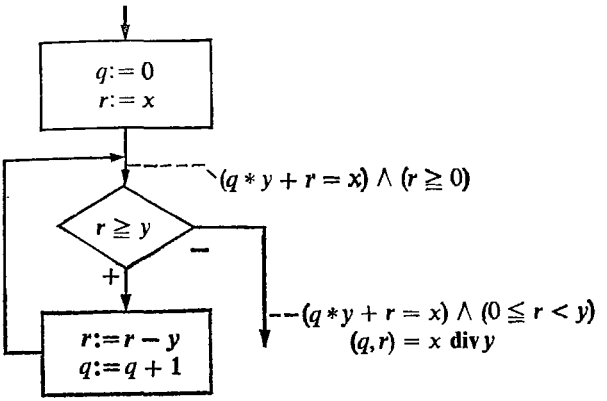
Умножение натуральных чисел

Аргументы: x, y
Результат: z
Вспомогательная переменная: u



(5.15)

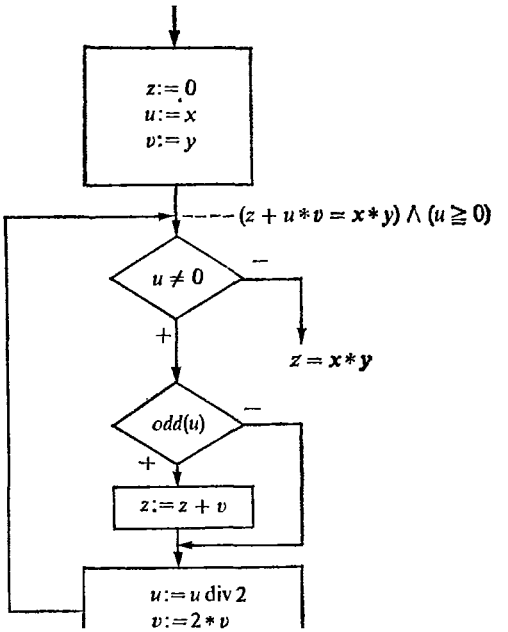
Деление натуральных чисел
Аргументы: x, y
Результаты: q (целое частное), r (остаток)



(5.16)

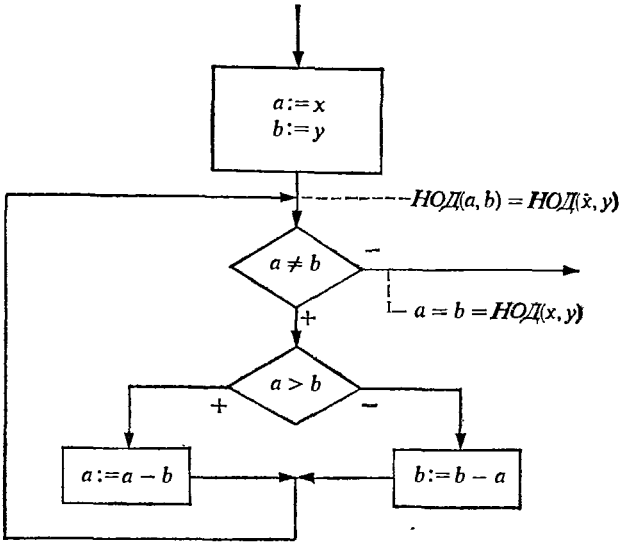
УПРАЖНЕНИЯ

5.1 В следующей программе для вычисления произведения $z := x * y$ используются только операции сложения, удваивания и деления пополам. Аргументы x, y — натуральные числа, а u и v — вспомогательные целые переменные. Условие $odd(u)$ удовлетворяется, если u — нечетное число. По заданному инварианту и правилам (5.3) — (5.6) определите антецеденты и консеквенты каждой инструкции.



(5.17)

5.2 Следующая программа вычисляет наибольший общий делитель (*НОД*) двух натуральных чисел x и y; a и b — целые переменные, в которых получается желаемый результат. Как и в упражнении (5.1), определите необходимые утверждения, используя известные соотношения функции *НОД*:
(a) $u>v$: $НОД(u, v)=НОД(u-v, v)$
(b) $НОД(u, v)=НОД(v, u)$
(c) $НОД(u, u)=u$

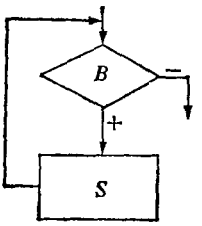


(5.18)

5.3 По аналогии с программой (5.17) напишите программу вычисления $z=x^y$ для натуральных чисел x и y. Получите утверждения, необходимые для проверки правильности вашей программы.

6. КОНЕЧНОСТЬ ПРОГРАММ

Цикл является самой типичной структурой в программах для вычислительных машин, поскольку он незаменим при *повторении* некоторого действия, что особенно хорошо согласуется со свойствами вычислительной машины как автомата. Способность вычислительной машины точно и с высокой надежностью выполнять тысячи повторяющихся операций имеет особую ценность. Но с другой стороны, именно эта неутомимость и слепое подчинение программе требуют от программиста повышенной осторожности. Одно из важных требований состоит в том, что каждая программа должна *завершиться* за *конечное* число повторений. К сожалению, случаи, когда процессы не заканчиваются, довольно обычны и приводят к существенным потерям на всех вычислительных установках. Этого можно избежать при тщательном проектировании и верификации программы. Чтобы объяснить необходимость защитных мер, обеспечивающих завершение программы, рассмотрим типичную структуру цикла:



(6.1)

Для окончания цикла требуется как минимум, чтобы S изменяла значения одной или нескольких переменных так, чтобы после конечного числа проходов условие B могло стать ложным. Вообще говоря, конечность повторений можно вывести формально, если выбрать целочисленную функцию N, зависящую от некоторых переменных программы, и показать, что (a) если B истинно, то $N>0$, и (b) каждое выполнение S уменьшает значение N. Применение этого правила тривиально в случае программы (5.2). Инструкция S есть

$$r := r - y$$
$$q := q+1$$

а условие B есть $r \geqslant y$, где r, q и y — натуральные числа. Мы выбираем $N=r-y$ так, как

(a) если $r \geqslant y$, то $N \geqslant 0$, и

(b) выполнение S уменьшает значение r, а следовательно, и N, т. е. конечность гарантируется. Заметим, в частности, необходимость начального условия $y>0$.

В качестве второго примера возьмем программу (5.18). Инструкция S в этой программе содержит две альтернативы:

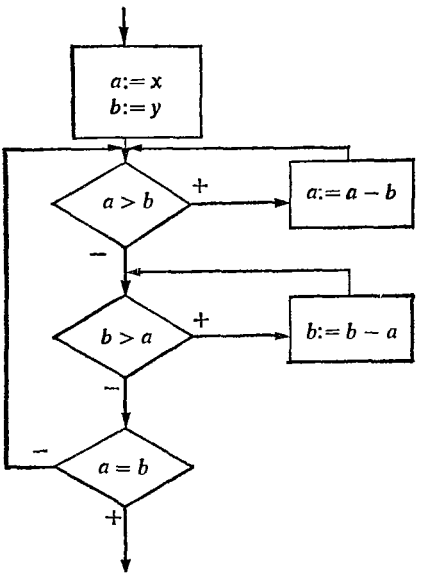
$$a := a - b, \text{ если } a>b, \text{ и } b := b - a, \text{ если } b>a,$$

а условием B является соотношение $a=b$. Переменные a и b — натуральные числа с начальными значениями $a>0$, $b>0$ и $a\neq b$. Подходящей функцией $N(a, b)$ будет $N=\max(a, b)$. Влияние S на N следует рассматривать в двух отдельных случаях. Если $a>b$, то значение b остается неизменным, а значение a уменьшается на b. Так как вначале $a>0$, $b>0$ и $a\neq b$, то первые два отношения останутся истинными (они инвариантны), а N уменьшится. Если $b>a$, то значение a останется прежним, а $b=\max(a, b)=N$ уменьшится на a. Итак, $N=\max(a, b)$ уменьшается при каждом повторении цикла, но, с другой стороны, $\min(a, b)$ остается всегда положительным, и, значит, наступит момент, когда $\max(a, b)=\min(a, b)$ и отношение $a\neq b$, следовательно, не будет больше выполняться; таким образом, цикл закончится.

УПРАЖНЕНИЯ

6.1 Определите диапазон значений x и y, которые будут гарантировать конечность программ из упражнений 5.1 и 5.3.

6.2 Для какого диапазона значений x и y следующая программа конечна?



(6.2)

6.3 В каких случаях программа (6.2) вычисляет наибольший общий делитель от x и y? Определите необходимые и достаточные утверждения.

7. ПОСЛЕДОВАТЕЛЬНАЯ НОТАЦИЯ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

7.1. Обзор

Программы, представленные в виде блок-схем, не пригодны для обработки в вычислительной машине, так как обычные устройства ввода не могут воспринять это двумерное изображение. Поэтому программы перед тем, как они будут выполняться, должны быть переведены в форму, пригодную для ввода в машину. Поскольку такой перевод, по всей видимости, является источником ошибок, возникает желание представлять программы в таком виде, который не только приемлем для машины, но удобен также и для программиста. Тогда такое представление служит *исходной формой*, которую программа обретает уже на стадии замысла, и, следовательно, вообще не возникает вопроса о том, какое из представлений (блок-схему или результат ее перевода) считать основным определением алгоритма.

Самыми распространенными устройствами ввода данных являются устройства ввода перфокарт и телетайпы (которые иногда совмещаются с устройствами ввода и подготовки перфоленты). Как в том, так и в другом случае данные имеют вид *линейной последовательности печатаемых литер*, т. е. вид последовательного текста. Системы обозначений, позволяющие представить программы в текстовом виде, обычно называются *языками программирования*. Они создаются в соответствии с правилами, которые точно определяют множество правильных предложений и инструкций. Набор таких правил называется *синтаксисом* языка. Поскольку программы, написанные на этих языках, должен читать и понимать автомат со сравнительно узким «кругозором», не удивительна «жесткость» синтаксических правил. Следовательно, чтобы овладеть языком, надо научиться понимать смысл любой из возможных форм и, кроме того, самым тщательным образом изучить синтаксические правила, управляющие языком. А это означает, что изменяются пропорции, и при изучении синтаксису языка программирования обычно уделяется несравненно большее внимание, чем синтаксису естественного языка, так как программы приходится не только читать и понимать, но, что более важно, придумывать и писать.

Если отвлечься от второстепенных проблем программирования, можно сказать, что главные усилия при конструировании программы направляются на составление и проверку лежащего в ее основе

алгоритма, усилия же на формулирование готового алгоритма с использованием специальных обозначений сравнительно невелики. Конструирование алгоритма оказывается в большинстве случаев сложным и длительным процессом, и для достижения конечной цели требуется несколько шагов. С каждым новым шагом программа специфицируется со все большими подробностями. Очевидно, используемая нотация (система обозначений) должна соответствовать решаемой задаче настолько, насколько это возможно, и совсем необязательно применять формальный язык в том виде, который удобен для ввода в машину. В качестве такой нотации можно с успехом использовать блок-схемы, математические формулы или даже естественный язык. Однако на последнем шаге мы должны получить программу, сформулированную на языке программирования, что прямым образом влияет на направление всего пошагового процесса разработки. Поэтому целесообразно ввести такую нотацию в самом начале курса программирования.

Перевод алгоритма на тот или иной машинный язык — задача сама по себе сложная, но она легко поддается механизации. Поэтому возникает большое желание иметь средства автоматизации кодирования. Особенно полезен язык, который содержит все фундаментальные и наиболее часто встречающиеся понятия программирования и позволяет выражать эти понятия в ясной и естественной форме, в то же время удобной для экономной и эффективной машинной обработки. При создании таких языков — ориентированных и на программиста, и на машину — существенное влияние оказали конкретные области применений, или существующие типы машин, или и то и другое вместе. Предмет вычислительной математики предопределил создание языка, который позволил в значительной степени приблизиться к традиционной и устоявшейся формальной математической записи. Впервые эти идеи были выдвинуты и сформулированы Х. Рутисхаузером в 1952 году [1]), но широкое распространение и использование они получили только после 1957 года, когда фирма IBM опубликовала язык программирования Фортран (FORTRAN— FORmula TRANslator) и выпустила компилятор, который автоматически транслировал программы в машинный код. В результате использование этого языка стало и теоретически интересным, и практически возможным. Однако даже в самой форме Фортрана явно просматривалась зависимость от типа вычислительных машин, выпускаемых фирмой IBM. Позднее язык был организован и определен так, чтобы оставалась возможность его улучшения [2]).

---

[1]) В это же время А. А. Ляпуновым был разработан аппарат логических схем программ, который лег в основу ранних ПП (программирующих программ). См., например, А. А. Ляпунов, О логических схемах программ, сб. «Проблемы кибернетики», вып. 1, Физматгиз, М. 1958.— *Прим. ред.*

[2]) U.S.A. Standard FORTRAN, USA Standards Inst., New York, 1966.

В 1958 году первоначальные идеи Х. Рутисхаузера были использованы группой специалистов при создании языка программирования. Этот язык получил название Алгол (ALGOL — ALGOrithmic Language) и стал предшественником языка Алгол-60, который в дальнейшем нашел широкое применение в научных приложениях. Алгол-60 был определен в 1960 году международной группой, состоящей из 13 ученых, и опубликован под редакцией П. Наура [1]). Сам по себе факт, что язык удалось определить точно и лаконично в относительно коротком документе, свидетельствует о достоинствах Алгола по сравнению с Фортраном. Кроме того, Алгол не ориентирован на какую-либо конкретную вычислительную машину, но в нем широко используются математические обозначения, с которыми знакомы ученые и инженеры. Для задания синтаксических правил был введен формализм, который позволял алгоритмически определить, является ли данная конструкция правильным предложением языка. Этот формализм, известный как *формализм Бэкуса — Наура* или как нормальная форма Бэкуса (НФБ), в дальнейшем использовался для определения других языков программирования.

Подобные усилия по созданию общего проблемно-ориентированного языка были предприняты в области обработки коммерческих данных. В 1962 году по заказу Министерства обороны США был разработан язык программирования Кобол [2]) (COBOL — COmmon Business Oriented Language). В нем учитывались нужды и особенности обработки коммерческих данных. В настоящее время Кобол один из самых распространенных (если не самый распространенный) язык программирования, но с точки зрения точности определения, стройности структуры и универсальности этот язык уступает даже Фортрану.

Отрицательным следствием быстрого распространения этих ранних языков стало то, что программирование разделилось на научную и коммерческую области применений. Считалось, что программирование в основном сводится к кодированию алгоритмов на специальном языке. Поэтому повсеместно установилось мнение, что так называемые научные и коммерческие программисты должны обучаться отдельно друг от друга. Однако на самом деле основные идеи конструирования программ и сами элементарные объекты, которые подвергаются обработке, совершенно не зависят от какой-либо области применения.

Усилия, направленные на «воссоединение» двух лагерей на базе общего языка, были предприняты фирмой IBM в период между 1964 и 1967 гг. С этой целью был определен и внедрялся язык, кото-

[1]) Naur P., Revised Report on the Algorithmic Language ALGOL 60, *Comm. ACM*, 6 (1963), 1—17 (есть русский перевод: Наур П. (под ред.), Алгоритмический язык АЛГОЛ-60, «Мир», М., 1965); Baumann R., Feliciano M., Bauer F. L., Samelson K., Introduction to ALGOL, Prentice-Hall, Inc., 1964.

[2]) U.S.A. Standard COBOL, USA Standards Inst., New York, 1968.

рый, как предполагалось, не только не будет зависеть от какой-либо частной машины, но будет также единообразно применяться в любой проблемной области. Этот язык был назван ПЛ/1 (PL/1). Сам язык, как и его описание, имеет весьма внушительные размеры. Из-за своих непомерно больших размеров (препятствующих возможности полного овладения языком) и из-за отсутствия систематической структуры с едиными основополагающими понятиями ПЛ/1 вряд ли может служить основным введением в программирование.

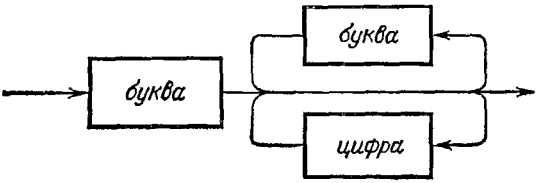
Следуя идее, что курс программирования прежде всего должен учить конструированию алгоритмов и лишь во вторую очередь рассматривать аспекты кодирования, мы здесь выбираем нотацию, которая не совпадает ни с одним из ранее упомянутых языков. Этот язык специально конструировался так, чтобы в нем нашли отражение самые фундаментальные понятия программирования в естественной, ясной и сжатой форме. Кроме того, его синтаксические правила просты, систематичны и хорошо поддаются автоматической обработке. Данный язык проектировался и определялся как язык, близкий к Алголу-60, так что его справедливо можно называть расширением Алгола-60 [1]. Мы будем знакомиться с отдельными элементами языка в логической последовательности по мере рассмотрения соответствующих понятий программирования. Поэтому в заключение этого раздела мы остановимся лишь на самых общих правилах задания синтаксической структуры языка.

Любой язык базируется на *словаре*. Предложения (в нашем случае программы) составляются с помощью соединения *основных символов* словаря в соответствии с *синтаксическими правилами*, определяющими язык. Обычно словарь (полный словарь для языка Паскаль приводится в приложении А) содержит буквы, цифры и специальные знаки (например, +, —, *). Поскольку обычно набор специальных символов в языках программирования довольно широкий, то для их обозначения часто используются также слова английского языка. Чтобы подчеркнуть, что эти слова (называемые *словами-разделителями*) являются основными символами, а не обычными последовательностями букв, в тексте они набираются жирным шрифтом (например, **begin**, **end**).

Синтаксис (см. приложение А) формируется таким образом, чтобы можно было легко проверить, является ли любая заданная последовательность символов правильным предложением языка. Правила задаются в виде диаграмм, которые называются *синтаксическими диаграммами*. Возможные пути, указанные в диаграмме, соответствуют допустимым последовательностям символов. Начинаясь с диаграммы «программа», путь приводит либо к другой диаграмме, если встречается прямоугольник, либо к основному символу *S*,

[1]) Wirth N., Programming Language PASCAL, *Acta Informatica*, **1** (1971), 35—63.

заключенному на диаграмме в кружок. Например:



(7.1)

Идентификаторы

Согласно диаграмме (7.1), следующие совокупности букв и цифр (наряду со многими другими) трактуются как допустимые идентификаторы

a
abcdef
*a*15
*q*2*p*9
Apollo

И наоборот, следующие цепочки не могут быть отнесены к идентификаторам:

word-delimiter
J.F.Kennedy
7*x*

7.2. Выражения и инструкции

Синтаксис языка определяет допустимые сентенциальные конструкции. Самыми важными среди них являются выражения и инструкции, которые встречаются практически во всех языках программирования.

Выражение — это формула или правило вычисления, которое специфицирует некоторое *значение* или результат. Выражение состоит из *операндов* и *операторов*. Операндами могут быть или константы (например, числа), или переменные, или значения, получаемые в результате вычисления функций. Операторы обычно подразделяются на *одноместные* и *двуместные*, т. е. имеющие соответственно один или два операнда. Если в выражении встречаются несколько операторов, то необходимо указать порядок, в котором они должны выполняться. Это можно сделать либо явно с помощью скобок, либо неявно, пользуясь правилами языка. Почти во всех языках двуместные операторы в свою очередь подразделяются (по крайней мере) на два класса: на операторы сложения (аддитивные) и операторы умножения (мультипликативные). Последним присваивается более высокий приоритет.

Кроме того, мы будем считать, что последовательности операторов с равными приоритетами всегда выполняются в обычном порядке — слева направо. Ниже приводятся примеры, иллюстрирующие эти простые правила.

$$\begin{array}{ll} x+y+z & =(x+y)+z \\ x*y+z & =(x*y)+z \\ x+y*z & =x+(y*z) \\ x-y*z-w & =(x-(y*z))-w \\ x*y-z*w & =(x*y)-(z*w) \\ -x+y/z & =(-x)+(y/z) \\ x*y/z & =(x*y)/z \\ x/y*z & =(x/y)*z \end{array} \qquad (7.2)$$

Самой элементарной *инструкцией* является инструкция присваивания. Она изображается следующим образом:

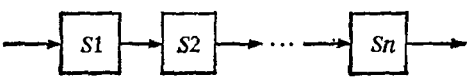
$$V := E \qquad (7.3)$$

где V обозначает переменную, а E — выражение. В отличие от выражения, которому соответствует значение, инструкция описывает *действие*.

Последовательности инструкций (составные инструкции), условные инструкции и циклы изображаются конструкциями, названными *структурными инструкциями*. Мы введем шесть базисных форм часто встречающихся структурных инструкций. Их смысл описывается эквивалентными блок-схемами.

Составные инструкции

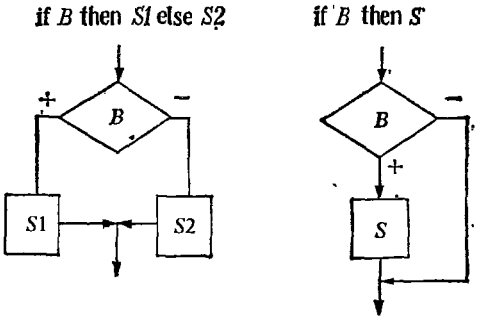
begin *S*1; *S*2; . . . ; *Sn* **end**



(7.4)

Разделитель «;» является оператором следования, он означает, что следующая инструкция будет выполняться только тогда, когда закончится выполнение предыдущей инструкции. Основные символы **begin** и **end** представляют «жирные» скобки или так называемые *инструктивные скобки*. Так как последовательности инструкций часто могут представлять собой довольно длинные тексты, то использование бросающихся в глаза скобок помогает выделять группы компонент, образующих составные инструкции.

Условные инструкции



(7.5)

Вторую форму можно рассматривать как сокращенную запись инструкции в том случае, когда отсутствует альтернатива $S2$.

В следующем примере иллюстрируется роль инструктивных скобок. Программа

if B **then** $S1$; $S2$ (7.6.1)

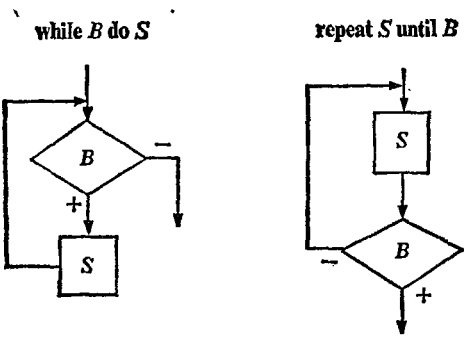
равносильна

begin if B **then** $S1$ **end**; $S2$ (7.6.2)

но отличается от

if B **then begin** $S1$; $S2$ **end** (7.6.3)

Циклические инструкции



(7.7)

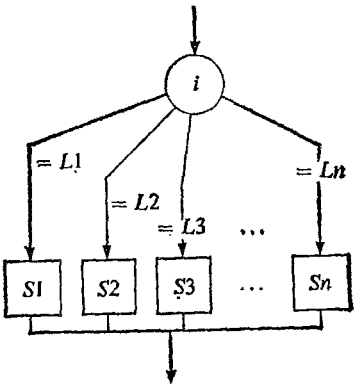
Замечание: Так как два основных символа **repeat** и **until** уже играют роль пары скобок, то конструкция

repeat $S1$; $S2$; ...; Sn **until** B

может не содержать дополнительной пары **begin — end.**

Селективные инструкции

case i **of** $L1$ $S1$; $L2$:$S2$; . . . ; Ln:Sn **end**



(7.8)

В случае если выражение i принимает значение L_k, выбирается и выполняется инструкция S_k; $L_i \neq L_j$ при $i \neq j$. *Замечание:* Если $S_i = S_j = \ldots = S_k$, то можно использовать следующую сокращенную запись L_i, L_j, . . ., $L_k : S$.

Так же как и в блок-схемах, должна существовать возможность легко включать в программу комментарии и утверждения. Поэтому условимся, что любой текст, заключенный в фигурные скобки, рассматривается как невыполняемый комментарий и, следовательно, может игнорироваться процессором. Теперь появляется возможность выразить элементарные правила вывода, используемые при верификации, применительно к схемам программ в линейной нотации [1]). В общем виде правило выглядит так:

$$\{P\}\ S\ \{Q\}$$

где P — антецедент, а Q — консеквент инструкции S.

Правила верификации в линейной записи

1. Инструкция присваивания

$$\{P^v_w\}\ v := w\ \{P\} \tag{7.9}$$

(см. 5.6.1)

2. Составная инструкция

Посылки: $\{P\}\ S1\ \{Q\}$, $\{Q\}\ S2\ \{R\}$ (7.10)

Следствие: $\{P\}\ S1; S2\ \{R\}$

[1]) Hoare C. A. R., An Axiomatic Basis for Computer Programming, *Comm. ACM*, **12** (Oct. 1969), 576—583.

3. Условные инструкции

Посылки: $\{P \wedge B\}\, S1\, \{Q\}$
$\{P \wedge \neg B\}\, S2\, \{Q\}$
Следствие: $\{P\}$ **if** B **then** $S1$ **else** $S2\, \{Q\}$ (7.11)

Посылки: $\{P \wedge B\}\, S\, \{Q\}$
$\{P \wedge \neg B\} \supset Q$
Следствие: $\{P\}$ **if** B **then** $S\, \{Q\}$ (7.12)

4. Циклические инструкции

Посылка: $\{P \wedge B\}\, S\, \{P\}$
Следствие: $\{P\}$ **while** B **do** $S\, \{P \wedge \neg B\}$ (7.13)

Посылки: $\{P\}\, S\, \{Q\}$
$\{Q \wedge \neg B\}\, S\, \{Q\}$
Следствие: $\{P\}$ **repeat** S **until** $B\, \{Q\}$ (7.14)

5. Селективная инструкция

Посылки: $\{P \wedge (i = L_k)\}\, S_k\, \{Q\}$ для всех k
Следствие: $\{P\}$ **case** i **of**
$L1 : S1;$
$L2 : S2;$
$\ldots$
$Ln : Sn$ **end** $\{Q\}$ (7.15)

(Следствие имеет место тогда и только тогда, когда существует некоторое k, для которого $i = L_k$.)

7.3. Линейная запись простых программ

Теперь можно записать программы, рассмотренные в предыдущих главах, используя последовательную нотацию.

Умножение двух натуральных чисел (см. 5.15)

```
begin z := 0; u := x;
      repeat {z + u * y = x * y, u > 0}
         z := z + y; u := u - 1
      until u = 0
end
```
(7.16)

Целочисленное деление двух натуральных чисел (см. 5.16)

```
begin q := 0; r := x;
      while r ⩾ y do
      begin {q * y + r = x, r ⩾ y}
         r := r - y; q := q + 1
      end
end
```
(7.17)

Умножение двух натуральных чисел
(см. 5.17)

```
begin z := 0; u := x; v := y;
    while u ≠ 0 do
    begin {z + u * v = x * y, u > 0}
        if odd(u) [1] then z := z + v;
        u := u div 2; v := 2 * v
    end
    {z = x * y}
end
```
(7.18)

Вычисление наибольшего общего делителя
(см. 5.18)

```
begin a := x; b := y;
    while a ≠ b do
    if a > b then a := a − b else b := b − a
        {НОД(a, b) = НОД(x, y)}
    {НОД(x, y) = a = b}
end
```
(7.19)

Заметим, что благодаря ступенчатой записи легко прослеживается структура алгоритма в программе. В частности, инструкции, принадлежащие одной и той же структурной единице, должны начинаться на одинаковом расстоянии от левого края. Тот же прием позволяет легко находить соответствующие друг другу открывающие и закрывающие скобки.

Программы (7.16) и (7.18) дают одинаковые результаты, но *вычислительные затраты* на получение этих результатов будут различны. Меру этих затрат можно получить подсчетом количества необходимых основных операций каждого типа. Программа (7.16) требует двух сложений и двух присваиваний при каждом повторении цикла. Если обозначить затраты на сложение через a и затраты на одно присваивание через z, то общие затраты на умножение с помощью программы (7.16) равняются

$$2z+2(z+a)*x \tag{7.20}$$

При каждом повторении цикла программа (7.18) требует одного удваивания, одного деления пополам, одной проверки на четность, двух или трех присваиваний и еще одного сложения. Наличие двух случаев в вычислениях затрудняет точный подсчет затрат, так как он зависит от операндов. Однако легко указать худший и лучший из случаев и, что более важно, увидеть, что благодаря делению попо-

---

[1]) В английском языке слово «odd» означает «нечетный». Таким образом, значением функции будет true, если u нечетно, и false — в противном случае.— *Прим. перев.*

лам u при каждом повторении цикла, максимальное число повторений равняется $\log_2(x)+1$, округленному до целого. Общее количество затрат в худшем случае равняется

$$3z+\log_2(x) * (3z+a+2h) \tag{7.21}$$

Так как вычислительные машины, в которых используется двоичное внутреннее представление чисел, могут умножать и делить на 2 и проверять на четность очень быстро (т. е. $h \leqslant a$), то программа (7.18) даже при малых значениях x превосходит программу (7.16), и, следовательно, ее нужно предпочесть во всех практических приложениях.

Программа (7.22) является улучшенным вариантом алгоритма деления (7.17), который основывается на том же принципе «деления пополам». Задание инвариантов позволяет легко установить правильность улучшенного алгоритма и в то же самое время делает очевидным его сущность. Значениями переменных q, r и w должны быть натуральные числа.

Программа деления натуральных чисел x на y

```
begin r := x; q := 0; w := y;
      while w ⩽ r do w := 2 * w;
      {w = 2^n * y > x}
      while w ≠ y do
      begin {q * w + r = x, r ⩾ 0}                    (7.22)
         q := 2 * q; w := w div 2;
         if w ⩽ r then
              begin r := r − w; q := q + 1
              end
      end
      {q * y + r = x, 0 ⩽ r < w; q = x div y}
end.
```

Здесь затраты на вычисления опять-таки уменьшаются, поскольку требуется только $\log_2(x/y)+1$ повторений цикла вместо x/y.

Программу (6.2), вычисляющую наибольший общий делитель, также можно сделать более эффективной, если вместо многократных вычитаний использовать деление. Чтобы упростить запись, мы введем оператор **mod**, который получает остаток от целочисленного деления тех же самых операндов x и y, то есть

$$\underbrace{(x \textbf{ div } y)}_{\text{частное}} * y + \underbrace{(x \textbf{ mod } y)}_{\text{остаток}} = x \tag{7.23}$$

Теперь многократные вычитания

$$\textbf{while } a \geqslant b \textbf{ do } a := a - b \tag{7.24.1}$$

можно заменить простым присваиванием

$$a := a \bmod b \qquad (7.24.2)$$

При этом программа (6.2), написанная в линейной нотации (7.25), преобразуется в программу (7.26).

```
begin a:=x; b:=y;
      repeat {a>0, b>0}
          while a>b do a:=a—b;
          while b>a do b:=b—a;                    (7.25)
      until a=b
      {a=b=НОД(x, y)}
end.
```

```
begin a:=x; b:=y;
    repeat {a>0, b>0}
        if a≥b then a:=a mod b;                   (7.26)
           {0≤a<b}
        if a>0 then b:=b mod a else Exchange(a, b)
    until b=0
    {a=НОД(x, y)}
end.
```

(В программе (7.26) используется функция *Exchange* (*a*, *b*), которая обменивает значения *a* и *b* между собой.) Этот вариант алгоритма вычисления наибольшего общего делителя принадлежит Евклиду и известен как один из самых ранних математических алгоритмов. Обычно он приводится в следующем эквивалентном виде:

```
begin a:=x; b:=y;
      repeat a:=a mod b; Exchange (a, b)
      until b=0                                   (7.27)
      {a=НОД (x, y)}
end.
```

Для верификации можно использовать следующее отношение:

$$x>y : НОД(x, y)=НОД(x \bmod y, x) \qquad (7.28)$$

УПРАЖНЕНИЯ

7.1 Пользуясь синтаксическими диаграммами, приведенными в приложении А, определите, какие из заданных последовательностей символов являются числами, константами, переменными, множителями, термами, выражениями или инструкциями. Обратите внимание на классификацию операторов по приоритетам.

операторы отношения $= \neq < \leqslant \geqslant >$
аддитивные операторы $+ - \vee$
мультипликативные операторы $* /$ **div** mod $\wedge$

Числа	0.31	+237.2	3.5	—0.005
	4.555	3+5	3*E*5	два
	33,75	.389	1*E*00	15
	10*E*—4	1.5+2	00037	3,250
Константы	100	*true*	+15.5	*красный*
	'A'	*девять*	9/5	*'**'
Переменные	*x*	*A* [*i*]	*x*+*y*	*B* [*i*, *j*]
Множители	*B* [*i*, *j*]	*sin* (*x*)	*p*	$p \vee q$
	(*x*)	$x * y$	*x*—*y*	*exp* (*y* * *ln* (*x*))
Термы	(*x*)	$x * y$	*x*—*y*	(*x*—*y*)
Выражения	*x*	2	*a*=*b*	+*x* * *y*
	(*x*)	$(x \leqslant y) \wedge (y < z)$	$p < q \wedge r < s$	*true*

Инструкции

a := *b* *a* := 2 2 := *a* sin (*x* * *y*)
begin *a*:= 1 **end**
if *a*=2 **then** *a* := *a*+1 **else** *P* (*x*, *y*)
while *a* > 0 **do** *a* := *a*—1 **end**
if *x* < *y* **then**; *z* := *true*; **else** *z* := *false*
repeat *z* := *z*+1.5, *y* := *u*—1 **until** *y*=0

7.2 Вычислите следующие выражения:

2 * 3—4 * 5=
15 **div** 4 * 4=
80 / 5 / 3=
2 / 3 * 2=

Напишите следующие выражения на языке, синтаксис которого определен в приложении А.

$$a^2 - c + \cfrac{a}{b * c + \cfrac{c}{d + \cfrac{e}{f}}}$$

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\frac{1}{2} * ln \left| \frac{w - a}{w + a} \right|$$

$$\cfrac{\cfrac{1}{a} + \cfrac{1}{b}}{c + d}$$

7.3 Выполните указанные программы с заданными значениями *x* и *y* и составьте трассировочные таблицы.
(a) Программы (7.16) и (7.18):
(*x*, *y*)=(3,5), (2,11), (10,8), (19,2)
(b) Программы (7.17) и (7.22):
(*x*, *y*)=(83,15), (117,9), (23,27), (1191,37)
(c) Программы (7.19), (7.25) и (7.27):
(*x*, *y*)=(84,36), (36,84), (770,441), (15,15)

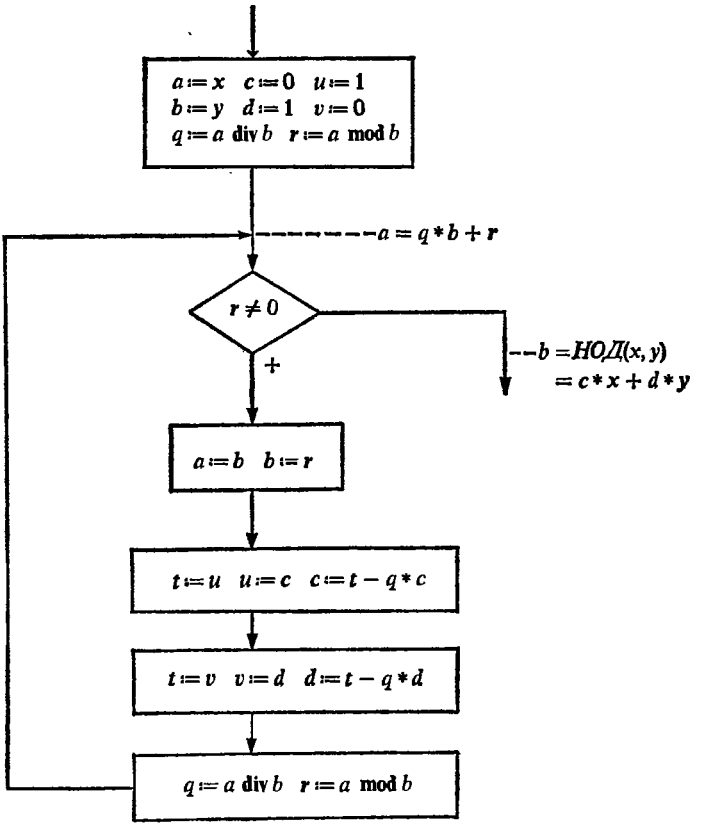
7.4 Определите верхнюю и нижнюю границы количества операций, выполняемых в программе 7.22 (как функции от *x* и *y*). Кроме того, определите необходимые и достаточные утверждения для верификации этой программы.

7.5 Переведите блок-схему (7.29) в последовательную нотацию. Обратите внимание, что программа вычисляет наибольший общий делитель *НОД* (*x*, *y*)

и два миожителя c и d, такие, что

$$c*x+d*y=НОД(x, y)$$

Определите необходимые и достаточные утверждения для верификации программы.



(7.29)

7.6 Составьте программу, в которой вычисляется $НОД(x, y)$ и используются только следующие соотношения:

(a) $НОД(2*m,\ 2*n)=2*НОД(m,\ n)$
(b) $odd(n) : НОД(2*m,\ n)=НОД(m,\ n)$
(c) $m>n : НОД(m-n,\ n)=НОД(m,\ n)$
(d) $НОД(n,\ m)=НОД(m,\ n)$
(e) $odd(m) \wedge odd(n) : \neg odd(m-n)$

Используйте только вычитание, сравнение, деление пополам и удвоение. Сопроводите программу утверждениями, необходимыми для ее верификации.

7.7 Всегда выполняется одно и только одно из условий P, Q и R. Такие условия называются взаимно исключающими условиями. Вероятность выполнения условия P есть W_P (W_Q и W_R соответственно для Q и R). Определите ожидаемое значение затрат на вычисление P, Q и R (как функций от W_P, W_Q, W_R) для следующих инструкций:

(a) **if** P **then** A **else if** Q **then** B **else** C
(b) **if** Q **then** B **else if** R **then** C **else** A
(c) **if** R **then** C **else if** P **then** A **else** B

Какую из трех эквивалентных инструкций следует предпочесть, если $>W_Q>W_R$?

Пример: $P=(x>y)$, $Q=(x=y)$, $R=(x<y)$;
$W_P=0.5$, $W_Q=0.3$, $W_R=0.2$

Обобщите полученное правило на случай с n взаимно исключающими виями ($n>3$).

8. ТИПЫ ДАННЫХ

В гл. 5 мы видели, как явная спецификация всех переменных в начале программы, составляющая существенную часть ее документации, значительно облегчает чтение программы. В этом случае введение новой переменной должно сопровождаться спецификацией ее возможных значений. Это требование диктуется следующими важными причинами.

1. Знание диапазона значений переменных очень существенно для понимания алгоритма. Если отсутствует явная спецификация, то обычно трудно определить вид объектов, которые могут быть значением переменной, а это означает, что поиск ошибок в программе отнимет больше времени и сил.
2. В большинстве случаев правомерность и корректность использования программы зависят от диапазона значений ее аргументов. Следовательно, спецификация диапазонов значений такая же важная часть документации программы, как и пояснение результатов.
3. Количество элементов памяти, необходимое для хранения значений переменной в машине, зависит от диапазона этих значений. Например, если переменная должна принимать n различных значений, то потребуется $\log_2(n)$ битов. Чтобы распределить память для переменных, компилятору необходимо знать диапазоны их значений.
4. Обычно встречающиеся в выражениях операторы определены только для некоторых диапазонов значений. Если диапазоны значений указаны, то на основании заданной информации компилятор может проверить допустимость той или иной комбинации операторов и операндов и выявить таким образом ошибки в программировании. С этой точки зрения указание диапазонов значений представляет собой определенную избыточность, которой можно воспользоваться при проверке некоторых свойств программы.
5. Реализация операторов часто зависит от диапазона значений их возможных аргументов. В этом случае знание диапазона значений абсолютно необходимо для получения эффективного представления программы. Типичным примером является представление чисел и реализация арифметических операто-

ров; выбор команд машины, выполняющих арифметическую операцию, обычно зависит от того, принадлежат ли ее аргументы множеству вещественных чисел или только множеству целых чисел.

Множество значений, которые может принимать переменная, играет столь важную роль для характеристики переменной, что оно называется *типом* переменной. Поэтому мы рекомендуем, чтобы все переменные были описаны в начале программы. *Описание переменной* задается в виде

$$\textbf{var } v : T \tag{8.1}$$

где v — идентификатор новой переменной, а T — ее тип. Если описывается несколько переменных одного и того же типа, то можно воспользоваться более короткой формой

$$\textbf{var } v_1, v_2, \ldots, v_m : T \tag{8.2}$$

где $v_1, \ldots, v_m$ — идентификаторы описываемых переменных.

Описание всех идентификаторов в начале программы имеет дополнительное преимущество, состоящее в том, что компилятор в этом случае может проверить, был ли описан каждый используемый в программе идентификатор. Если идентификатор не был описан (например, из-за ошибки в пробивке или из-за описки программиста), то компилятор может сообщить программисту об этой ошибке, а не вводить новую переменную с ошибочным именем. Здесь снова избыточность в тексте программы используется для повышения надежности программирования.

Теперь перед нами встают два вопроса: как правильно задать тип данных в программе и каким удобным образом типы данных можно представить в памяти машины? Прежде всего следует различать классы типов данных. Самой важной и, пожалуй, единственной отличительной чертой оказывается структурность значений того или иного типа. Если значение *не структурное*, т. е. не распадается на компоненты, то оно называется *скаляром*. В этой главе мы познакомимся только со скалярными типами, структурные типы будут рассматриваться в гл. 10 и 11.

Общая форма *определения типа* следующая:

$$\textbf{type } t = T \tag{8.3}$$

где t есть вновь введенный идентификатор, а T — описание типа. Скалярный тип описывается перечислением его компонент. Для этого используется запись вида

$$\textbf{type } t = (w_1, w_2, \ldots, w_n) \tag{8.4}$$

Такое определение вводит *идентификатор типа* t и n *константных идентификаторов* $w_1, w_2, \ldots, w_n$. Ниже приводятся примеры определений скалярных типов.

type *цвет=(красный, желтый, зеленый, голубой)*
type *масть=(бубны, черви, пики, трефы)*
type *фигура=(треугольник, прямоугольник, круг)*
type *состояние=(твердый, жидкий, газообразный)*

Заметим, что имя типа можно не задавать, если объединить описание переменных (8.2) и определение типа (8.4) следующим образом:

$$\textbf{var } v_1, v_2, \ldots, v_m : (w_1, w_2, \ldots, w_n) \tag{8.5}$$

В записи, определяющей тип, элементы множества значений не только отличны друг от друга, но и *упорядочены*. Для любого скалярного типа t, определенного записью вида (8.4), мы постулируем следующие аксиомы:

1. $w_i \neq w_j$, если $i \neq j$ (различимость)
2. $w_i < w_j$, если $i < j$ (упорядоченность) (8.6)
3. Значениями типа t могут быть только $w_1, \ldots, w_n$

Наличие упорядоченности позволяет ввести функции получения следующего или предыдущего элемента:

$$succ(w_i)=w_{i+1} \quad \text{для} \quad i=1,\ldots, n-1$$
$$pred(w_i)=w_{i-1} \quad \text{для} \quad i=2,\ldots, n \tag{8.7}$$

Если в программу требуется включить несколько определений скалярных типов, то мы будем твердо придерживаться правила, в соответствии с которым константный идентификатор определяется только один раз. Тогда с каждым константным идентификатором можно однозначно соотнести тип значения, которое он обозначает. Следует избегать комбинации таких определений, как

type *теплыйцвет=(красный, желтый, зеленый)*
type *холодныйцвет=(зеленый, голубой)* (8.8)

поскольку в них тип некоторых констант (в данном случае цвет *зеленый*) определяется неоднозначно [1]).

Некоторые скалярные типы используются так часто, что и они, и их операторы присутствуют в любой вычислительной системе. Такие типы (их называют стандартными типами) нет необходимости определять в программе, поскольку предполагается, что они известны любому процессору. К ним относятся логические значения (ложь, истина), целые и вещественные числа, а также множество литер (которые можно напечатать). Они так часто используются, что соответствующие им константы (за исключением логических значений, см. разд. 8.1) обозначаются не идентификаторами, а конструкциями, которые различаются на синтаксическом уровне. Теперь мы перейдем к рассмотрению четырех главных стандартных типов.

---

[1]) Мы сознательно отказались от таких понятий, как объединение и пересечение типов.

8.1. Тип BOOLEAN (логический)

Тип *Boolean* определяет диапазон логических значений, который содержит два элемента *true* (истина) и *false* (ложь). Само название типа происходит от имени Джорджа Буля (George Boole, 1815—1864), заложившего основы алгебры логики. Этот тип определяется как

$$\textbf{type } Boolean = (false,\ true) \tag{8.9}$$

Над аргументами этого типа определяются следующие стандартные операторы:

$\vee$ дизъюнкция — OR (или)
$\wedge$ конъюнкция — AND (и)
$\neg$ отрицание — NOT (не)

Если заданы логические аргументы p и q, то значение выражений $p \vee q$, $p \wedge q$ и $\neg p$ определяются таблицей (8.10).

p	q	$p \vee q$	$p \wedge q$	$\neg p$
false	*false*	*false*	*false*	*true*
true	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
true	*true*	*true*	*true*	*false*

(8.10)

Из этой таблицы можно вывести соотношения (8.11) — (8.14), которые полезны во многих случаях, особенно тогда, когда требуется найти более простые эквивалентные формы заданного выражения.

1. $p \vee q = q \vee p$
   $p \wedge q = q \wedge p$ — Законы коммутативности (8.11)
2. $(p \vee q) \vee r = p \vee (q \vee r)$
   $(p \wedge q) \wedge r = p \wedge (q \wedge r)$ — Законы ассоциативности (8.12)
3. $(p \wedge q) \vee r = (p \vee r) \wedge (q \vee r)$
   $(p \vee q) \wedge r = (p \wedge r) \vee (q \wedge r)$ — Законы дистрибутивности (8.13)
4. $\neg (p \vee q) = \neg p \wedge \neg q$
   $\neg (p \wedge q) = \neg p \vee \neg q$ — Законы де Моргана (8.14)

По определению оператор $\vee$ имеет более низкий приоритет, чем $\wedge$, приоритет которого в свою очередь ниже чем $\neg$. Например, $\neg p \vee q \wedge r$ понимается как $(\neg p) \vee (q \wedge r)$.

Все *операторы отношения* дают результат типа *Boolean*. Например, выражение $x = y$ имеет значение *true*, если x равен y, и *false* в противном случае. Наиболее употребительными операторами отношения являются $=$, $\neq$, $<$, $\leqslant$, $\geqslant$, $>$. Последние четыре оператора, очевидно, применяются только к упорядоченным (т. е. к

скалярным) типам. Для этих операторов справедливы следующие соотношения:

$$\begin{aligned} x \neq y &\leftrightarrow \neg (x=y) \\ x \leqslant y &\leftrightarrow (x<y) \vee (x=y) \\ x \geqslant y &\leftrightarrow \neg (x<y) \\ x>y &\leftrightarrow \neg (x<y) \wedge \neg (x=y) \end{aligned} \qquad (8.15)$$

Следовательно, шесть операторов отношения с помощью логических связок $\wedge$, $\vee$ и $\neg$ можно выразить только через отношения $=$ и $<$.

8.2. Тип INTEGER (целый)

Этот тип представляет множество *целых чисел*. Предполагается, что в данной системе программирования определены и доступны следующие операторы:

$+$ сложение
$-$ вычитание
$*$ умножение
div целочисленное деление
mod остаток от целочисленного деления

В каждой вычислительной машине определено некоторое подмножество целых чисел (лежащих в некоторых пределах), таких, с которыми ее арифметическое устройство может оперировать прямо и эффективно. Следовательно, тип *integer* обозначает множество чисел, определенное данной вычислительной машиной. Разумеется, эти пределы будут меняться от машины к машине. Важно отметить, что обычные аксиомы арифметики, вообще говоря, нельзя применять к арифметике вычислительной машины. Они не верны в тех случаях, когда истинный результат операции лежит вне заданного конечного диапазона значений. Например, если для некоторой вычислительной системы тип *integer* определен как множество чисел, по абсолютной величине не превосходящих *max* ($|x| \leqslant max$), и если сложение, выполняемое машиной, обозначить через $\oplus$, то

$$x \oplus y = x + y$$

только тогда, когда $|x+y| \leqslant max$. Следовательно, обычный ассоциативный закон, вообще говоря, для машинного сложения не выполняется. Соотношение

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

верно тогда, когда $|x \oplus y| \leqslant max$ и $|y \oplus z| \leqslant max$. Например, подставив значения и положив $max = 100$, получим

$$60 \oplus (50 \oplus (-40)) = 60 + 10 = 70$$

в то время как результат вычисления

$$(60 \oplus 50) \oplus (-40)$$

не определен. На первый взгляд эта ситуация кажется почти безнадежной. Но не следует забывать, что в большинстве вычислительных машин диапазон целых чисел существенно превышает тот, который требуется при решении задач. Поэтому не столь уж часто приходится встречаться со слишком большими числами и, следовательно, с неопределенными результатами (которые называют *переполнением*). Тем не менее при переполнении любая вычислительная система должна хотя бы выдавать предупреждающий сигнал и прекращать вычисления, поскольку продолжать счет с неопределенными результатами едва ли имеет смысл. Однако, за исключением случая переполнения, предполагается, что все *операции над аргументами типа integer выполняются точно.*

Во многих случаях заранее известно, что значения некоторых переменных лежат внутри определенного интервала. Эту информацию следует сообщить при определении типа переменной, явно указывая предполагаемый интервал. Такой тип называется *интервальным типом* и определяется следующим образом:

type *t=min..max* (8.16)

Константы *min* и *max* задают границы интервала или отрезка. Указание отрезка часто значительно облегчает понимание программ. Конечно, тип *integer* сам является отрезком, границы которого определяются конкретной вычислительной машиной или используемым компилятором.

8.3. Тип CHAR (литерный)

Этот тип обозначает конечное упорядоченное *множество литер*. В вычислительной системе определяется не только диапазон чисел, приемлемых для обработки, но и набор литер, с помощью которых она общается с внешним миром. Литерами из этого набора должны быть снабжены периферийные (читающие и печатающие) устройства вычислительной системы. Крайне желательна стандартизация наборов литер (она даже обязательна, если различные вычислительные системы связываются для совместной работы, передачи данных, дистанционной обработки данных и т. д.). И хотя на полное согласие в стандартизации наборов литер едва ли можно рассчитывать, тем не менее существует общепринятый набор литер для вычислительных систем, содержащий 26 латинских *букв*, 10 десятичных арабских *цифр* и некоторое количество *специальных литер*, таких, как знаки пунктуации. Кроме того, получил широкое распространение набор литер, утвержденный Международной организацией стандартов ISO (International Standards Organisation), и американский вариант этого набора — ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией).

Набор ASCII состоит из 128 литер. Поскольку $128=2^7$, каждая литера однозначно представляется или кодируется некоторой комбинацией из 7 битов. Отображение комбинаций битов на множество литер называется *кодом*; отсюда ASCII называется *семибитовым кодом* (см. приложение B).

Литеры ASCII делятся на *печатаемые литеры* и *управляющие литеры*. Печатаемые литеры в свою очередь подразделяются на заглавные буквы, строчные буквы, цифры и специальные знаки. Далее различается *полный* и *ограниченный наборы* ASCII. Последний, в котором нет строчных букв, используется на многих серийно выпускаемых устройствах.

Смысл управляющих литер объясняется в приложении B, они играют большую роль при передаче данных. Здесь мы приводим только две управляющие литеры: **cr** — возврат каретки (carriage return) и **lf** — перевод строки (line feed). Если данные передаются на печатающее терминальное устройство, то эти литеры служат сигналом, по которому печатающий механизм приготовится к печати с начала новой строки.

Таблица литер в коде ASCII

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	qs	-	=	M	]	m	}
14	so	rs	.	>	N	^	n	¬
15	si	us	/	?	O	_	o	del

Следующие две стандартные функции (которые зависят от выбранного набора литер) позволяют отображать множество литер на подмножество натуральных чисел и наоборот. Эти функции называются *функциями преобразования*.

ord(*c*) является порядковым номером литеры *c* из упорядоченного набора литер. (Если используется

таблица литер ASCII, то $ord(c) = 16* x + y$, где x и y обозначают координаты литеры c.)

chr(i) является литерой с порядковым номером i.

Поэтому справедливы соотношения

$$chr(ord(c)) = c \text{ и } ord(chr(i)) = i \tag{8.17}$$

и упорядоченность набора литер определяется отношением

$$c_1 < c_2 \leftrightarrow ord(c_1) < ord(c_2) \tag{8.18}$$

Для обозначения константы типа *char* обычно литеру заключают в апострофы (или в одиночные кавычки). Например, если переменной c типа *char* присваивается знак вопроса, то это присваивание записывается так:

$$c := '?'$$

В последующих главах мы не будем в программах ссылаться на какой-то конкретный набор литер. Однако программы будут иметь смысл и выполняться в полном соответствии с их спецификациями только в том случае, когда применяемый набор литер отвечает следующим минимальным требованиям.

1. Набор литер должен содержать буквы A—Z и цифры 0—9.
2. Подмножества букв и цифр должны быть упорядоченными и связанными. То есть c есть буква только тогда, когда $'A' \leqslant c$ и $c \leqslant 'Z'$, и c есть цифра только в том случае, если $'0' \leqslant c$ и $c \leqslant '9'$.
3. Набор литер должен содержать пробел, разделитель строк, который обозначается **eol** (**end of line** — конец строки), и несколько других литер, таких, как запятая, точка и т. д.

8.4. Тип REAL (вещественный)

Тот факт, что в машине можно представить только значения из конечного диапазона, имеет особо важные последствия для обработки вещественных чисел. В случае целых чисел можно было утверждать, что при любых обстоятельствах, кроме переполнения, в результате выполнения арифметических операций получались точные значения, но применительно к арифметике с вещественными числами это утверждение неверно. Причина заключается в том, что на любом сколь угодно малом интервале оси вещественных чисел содержится бесконечно много значений; ось вещественных чисел образует так называемый *континуум*. В программировании, следовательно, тип *real* не представляет бесконечное, несчетное множество вещественных чисел; ему соответствует *конечное множество представителей интервалов* континуума вещественных чисел. Результаты вычислений, в которых участвуют приближенные, а не точные зна-

чения, в значительной степени зависят от решаемой задачи и от выбранного алгоритма. В лучшем случае полученные результаты будут приближениями с неизбежными погрешностями. Оценка этих погрешностей, которые являются следствием подмены вещественного континуума конечным множеством представителей, является очень трудной задачей и служит объектом изучения *вычислительной математики*. Вычисления, выполняемые над данными типа *real*, называются *численными процессами*. Не вкладывая какого-либо отрицательного смысла, можно сказать, что слово «численный» синоним слова «неточный».

Но бессмысленно производить вычисления, не зная ничего о природе и степени ожидаемой погрешности. Чтобы понимать и даже проводить такие измерения, необходимо знать, какого вида представления используются для вещественных чисел с конечным числом цифр. В современных цифровых вычислительных машинах обычно используют так называемое *представление с плавающей точкой*, т. е. вещественное число x изображается с помощью двух целых чисел e и m, каждое из которых содержит конечное число цифр, так что

$$x=m*B^e \quad -E<e<E \quad -M<m<M \tag{8.19}$$

Здесь m называется *мантиссой*, e — *порядком*, а B, E и M являются константами, характеризующими представление. Число B называется *основанием* представления с плавающей точкой. Обычно B не равно 10, а является малой степенью 2. Для любого заданного значения x можно найти много различных пар $\langle m, e\rangle$. Каноническая или *нормализованная форма* определяется дополнительным соотношением

$$\frac{M}{B} \leqslant |m| < M \tag{8.20}$$

При использовании только нормализованной формы плотность представителей интервалов на оси вещественных чисел экспоненциально уменьшается с увеличением $|x|$. Например, интервал [0.1 : 1] содержит приблизительно (точно, если $B=10$) столько же представителей, сколько интервал [10000 : 100000]. Трудно определить, как конкретно скажется на той или иной программе это неравномерное распределение представителей; выскажем лишь самые общие соображения. Очевидно, нельзя абсолютно точно описать элементарные операции над аргументами типа *real*, поэтому они определяются в виде минимальных условий, которые всегда выполняются независимо от специфики арифметики с плавающей точкой. Эти условия можно сформулировать в виде аксиом.

A1. Тип *real* (обозначается R) является *конечным* подмножеством множества R вещественных чисел.

$$R \subset \mathrm{R}$$

A2. Каждому числу $x \in \mathbf{R}$ ставится в соответствие число $\tilde{x} \in R$, которое называется *представителем* x.

A3. Каждое $\tilde{x} \in R$ представляет много $x \in \mathbf{R}$, но множество представляемых значений является связным интервалом на вещественной числовой оси. То есть, если $x_1 < x_2$, $\tilde{x}_1 = r$ и $\tilde{x}_2 = r$, то $\tilde{x} = r$ для всех $x_1 \leqslant x \leqslant x_2$. Более того,

$$\text{из } x \in R \text{ следует } \tilde{x} = x$$

В частности, 0 и 1 представляются точно. То есть, $0 \in R$ и $1 \in R$; следовательно, $\tilde{0} = 0$ и $\tilde{1} = 1$.

A4. Существует максимальное значение *max*, такое, что для всех $|x| \geqslant max$ представители не определены. Диапазон чисел $|x| \geqslant max$ называется *диапазоном переполнения* $\mathbf{U}$. Множество $\mathbf{R} - \mathbf{U}$ есть связное множество.

Из аксиом A1—A4 следует, что

$$\begin{aligned} &\text{если } x < y, \text{ то } \tilde{x} \leqslant \tilde{y} \\ &\text{если } x = y, \text{ то } \tilde{x} = \tilde{y} \\ &\text{если } x > y, \text{ то } \tilde{x} \geqslant \tilde{y} \end{aligned} \tag{8.21}$$

A5. Множество R симметрично относительно 0, т. е.

$$(-x)\tilde{} = -(\tilde{x}) \tag{8.22}$$

Аксиомы A6—A9 постулируют совокупность требований, которые безоговорочно должны выполняться любой машинной арифметикой. Основные математические операции — сложение, вычитание, умножение и деление — обозначаются соответственно $\oplus$, $\ominus$, $\otimes$ и $\varnothing$. Мы предполагаем, что $x, y \in R$.

A6. *Коммутативность* сложения и умножения

$$x \oplus y = y \oplus x, \quad x \otimes y = y \otimes x \tag{8.23}$$

A7.
$$x \geqslant y \geqslant 0 \rightarrow (x \ominus y) \oplus y = x \tag{8.24}$$

A8. *Симметричность* основных операций относительно 0

$$\begin{aligned} x \ominus y &= x \oplus (-y) = -(y \ominus x) \\ (-x) \otimes y &= x \otimes (-y) = -(x \otimes y) \\ (-x) \varnothing y &= x \varnothing (-y) = -(x \varnothing y) \end{aligned} \tag{8.25}$$

A9. *Монотонность* основных операций

$$\begin{aligned} &\text{если } 0 \leqslant x \leqslant a \text{ и } 0 \leqslant y \leqslant b \\ &\text{то } x \oplus y \leqslant a \oplus b \quad x \ominus b \leqslant a \ominus y \\ &\qquad x \otimes y \leqslant a \otimes b \quad x \varnothing b \leqslant a \varnothing y \end{aligned} \tag{8.26}$$

Из A9 следует, в частности, что для некоторых x и y, таких, что $0 \leqslant x < a$ и $0 \leqslant y < b$, вполне может случиться, что

$$x \oplus y = a \oplus b \text{ или } x \otimes y = a \otimes b$$

но не

$$x \oplus y > a \oplus b \text{ или } x \otimes y > a \otimes b$$

Из аксиом А1—А9 можно вывести следующие теоремы, которые представляют важные основные свойства арифметики:

$$\begin{gathered} y \geqslant 0 \rightarrow x \oplus y \geqslant x \\ x \geqslant y \rightarrow x \ominus y \geqslant 0 \\ (x \geqslant 0) \wedge (0 \leqslant y \leqslant 1) \rightarrow x \otimes y \leqslant x \\ 0 < x \leqslant y \rightarrow x \varnothing y \leqslant 1 \end{gathered} \tag{8.27}$$

$$\begin{gathered} x \ominus x = 0 \\ x \oplus 0 = x \ominus 0 = x \\ x \otimes 0 = 0 \\ x \otimes 1 = x \varnothing 1 = x \\ x \varnothing x = 1 \end{gathered}$$

Обратите внимание, что обычные для арифметики *законы ассоциативности и дистрибутивности* отсутствуют в аксиомах А1—А9. Это не случайно. Рассмотрим пример, который иллюстрирует нарушение ассоциативного закона при сложении. В этом примере числа представлены четырьмя десятичными цифрами. (Порядок e задается соответствующим расположением десятичной точки.)

$$x = 9.900 \quad y = 1.000 \quad z = -0.999$$

1. $(x \oplus y) \oplus z = 10.90 \oplus (-0.999) = 9.910$
2. $x \oplus (y \oplus z) = 9.900 \oplus 0.001 = 9.901$

Рассмотрим пример, в котором нарушается дистрибутивный закон (по-прежнему используется четырехразрядная десятичная арифметика).

$$x = 1100. \quad y = -5.000 \quad z = 5.001$$

1. $(x \otimes y) \oplus (x \otimes z) = -5500. \oplus 5501. = 1.000$
2. $x \otimes (y \oplus z) = 1100. \otimes 0.001 = 1.100$

Определенную «опасность» представляют операции сложения и вычитания. Они являются причиной значительных ошибок, особенно при вычитании двух почти равных значений. В этом случае большое количество значащих цифр взаимно уничтожаются и в полученной разности может не оказаться нескольких, а может быть, и всех значащих цифр. Это явление называется *сокращением*.

Деление также представляет собой потенциальную опасность. В случае малых делителей результат легко может оказаться в области переполнения. Поэтому следует избегать деления на нуль или число «близкое» к нулю. Например, не следует в программе пользоваться отношением вида $abs\ (t/x) \leqslant eps$, его лучше заменить отношением $abs(t) \leqslant eps * abs\ (x)$, не содержащим деления.

Мера *точности* арифметики с плавающей точкой основывается на величине ε, которая определяется формулой

$$\varepsilon = \min_{x>0}(x \mid (1+x)^{\sim} \neq 1) \tag{8.28}$$

То есть ε является наименьшим положительным числом, таким, что представители 1 и $1+\varepsilon$ отличаются друг от друга. Например, если вещественные числа в вычислительной машине представляются в виде n десятичных цифр, то ε приблизительно равняется 10^{-n}.

Хотя в математическом смысле целые числа являются подмножеством вещественных чисел, обычно типы *integer* и *real* рассматриваются как непересекающиеся типы. Чтобы удовлетворить основному правилу записи, в соответствии с которым тип константы должен определяться из самой записи этой константы, условимся, что число считается числом типа *integer* тогда и только тогда, когда его запись не содержит ни десятичной точки, ни масштабного множителя (см. гл. 7). Все числа, запись которых содержит десятичную точку и/или масштабный множитель, считаются числами типа *real*. При программировании с использованием объектов типа *real* будут учитываться следующие дополнительные условия:

1. В выражении типа *real* любой операнд типа *real* можно заменить операндом типа *integer*. Таким образом, нет необходимости явно указывать преобразование из типа *integer* в тип *real*. Однако программист должен помнить, что компилятор в соответствующих местах вставит *неявные команды преобразования*, если в машине используются различные внутренние представления для значений этих двух типов (в большинстве вычислительных машин они различны).
2. Если аргумент типа *real* используется там, где допустим только аргумент типа *integer*, то необходимо явно указать функцию преобразования. В качестве стандартной функции преобразования мы взяли ту, которая проще других реализуется на существующих вычислительных машинах, а именно

$$trunc(x)$$

Эта функция дает целое число, получаемое в результате отбрасывания дробной части x. Например:

$$trunc\,(5.8) = 5, \qquad trunc(4.3) = 4$$

Функция округления выражается следующим образом:

$$round\,(x) = \begin{cases} trunc\,(x+0.5) & \text{для } x \geqslant 0 \\ -trunc\,(0.5-x) & \text{для } x < 0 \end{cases}$$

Пример: Решение квадратного уравнения

Этот пример заставит программиста задуматься о тех опасностях, с которыми можно встретиться, когда используется арифметика с вещественными числами, и продемонстрирует, каким образом эти опасности можно преодолеть.

Требуется найти два корня x_1 и x_2 квадратного уравнения

$$a*x^2+b*x+c=0 \quad a\neq 0 \tag{8.29.1}$$

Буквальный перевод хорошо известной формулы

$$x_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{8.29.2}$$

приводит нас к следующим инструкциям программы:

$d := sqrt(sqr(b)-4*a*c)$;
$x_2 := -(b+d)/(2*a)$; $x_1 := (d-b)/(2*a)$

Допустим, что $a=1.000$, $b=-200.0$ и $c=1.000$, и предположим также, что все арифметические операции выполняются с четырьмя десятичными цифрами. В результате мы получим

$$\begin{aligned} d&=sqrt(40000-4.000)=200.0\\ x_1&=400.0/2.000=200.0\\ x_2&=0.000/2.000=0.000 \end{aligned} \tag{8.30}$$

Правильными результатами, однако, будут $x_1=200.0$ и $x_2=-0.005$. Если качество программы оценивать с точки зрения относительной точности, то результат x_2 следует отвергнуть как совершенно неверный.

Алгоритм, в котором учитывается особенность использования арифметики с плавающей точкой и конечной точностью, основывается на правиле Виета

$$x_1*x_2=c/a \tag{8.31}$$

Теперь с помощью (8.29.2) вычисляется только один корень с бóльшим абсолютным значением. Второй корень получается, согласно (8.31), с помощью одного умножения и одного деления — операций, которые сохраняют относительную точность аргументов. В результате мы приходим к следующей программе:

```
d := sqrt(sqr(b)-4*a*c);
if b>=0 then x1 := -(b+d)/(2*a)
        else x1 := (d-b)/(2*a);
x2 := c/(x1*a)
```
(8.32)

В этом примере приводится довольно характерный случай, когда математические методы, известные еще со школьной скамьи,

требуют дополнительного и тщательного исследования, если решение получается с помощью вычислительной машины.

УПРАЖНЕНИЯ

8.1 Какие из следующих выражений синтаксически правильны (имеется в виду совместимость типов)? Определите типы этих выражений, исходя из следующих описаний переменных:

```
var x, y, z: real; i, j, k: integer
x+y*i          i mod (j+y)        i+j-k
i div j+x      x+y < i+j          k-trunc (x*i)
i*x+j*y        x < y ∧ y < z      x=i
```

В каждом случае подсчитайте количество необходимых неявных преобразований из целого типа в вещественный.

8.2 Дополните программы (7.16), (7.17), (7.18), (7.19), (7.22) и (7.25) необходимыми описаниями переменных.

8.3 Напишите программу вычисления частичной суммы ряда

$$1-1/2+1/3-1/4+\ldots+1/9999-1/10000$$

Вычисление проведите следующими способами:

(a) сложите члены ряда слева направо,
(b) сложите члены ряда справа налево,
(c) сложите положительные и отрицательные члены отдельно, но в обоих случаях слева направо,
(d) сложите положительные и отрицательные члены ряда отдельно и в обоих случаях справа налево.

Попробуйте написать несколько программ для вычисления этих четырех сумм и оцените их относительные достоинства и недостатки. Сравните полученные с помощью вычислительной машины четыре результата и объясните их разницу. *Указание*: Инструкции вида $a := b*(-1)$ следует писать, как $a := -b$. Для сравнения ниже приводятся 30 верных десятичных цифр ответа

$$0.693097183059945296917232371458$$

8.4 Напишите программу, которая многократно умножает комплексную переменную z на комплексную константу $c=0.6+0.8i$. Комплексное число $z=x+iy$ ($i=\sqrt{-1}$) представляется в виде пары вещественных переменных x и y. *Замечание*: Абсолютное значение c равняется 1. Если в программу включить вычисление

$$|z|=sqrt(x^2+y^2)$$

то различие значений $|z|$ в начале и в конце вычислений можно использовать в качестве меры точности арифметики с плавающей точкой. Выберите $n=500$ и начальное значение $|z|=1$.

8.5 Покажите, что для любой арифметики с плавающей точкой и конечной точностью и основанием $B>2$ существуют такие числа x, что

$$(x \oplus x)/2 \neq x$$

Указание: Сначала найдите такое x, используя десятичное представление, содержащее две цифры.

9. ПРОГРАММЫ, ОСНОВАННЫЕ НА РЕКУРРЕНТНЫХ СООТНОШЕНИЯХ

9.1. Последовательности

В предыдущих главах мы ввели некоторые фундаментальные структуры программ и основные типы данных. Теперь мы детально исследуем программы, которые по существу содержат *одну повторяемую инструкцию*, т. е. программы следующего общего вида:

$$\textbf{while } B \textbf{ do } S \tag{9.1}$$

где B — логическое выражение, а S — инструкция. Прежде всего заметим, что цикл может закончиться только в том случае, если инструкция S так влияет на выражение B, что через конечное число повторений значением выражения B станет *false*. Но из этого следует, что в S должно быть по крайней мере одно присваивание переменной, которая встречается в B. Обозначим через V множество всех переменных, встречающихся в программе (таким образом, каждая переменная является элементом множества V). Тогда инструкцию (9.1) можно переписать в виде так называемой *схемы программы:*

$$\begin{array}{|l} V := v_0; \\ \textbf{while } p(V) \textbf{ do } V := f(V) \end{array} \tag{9.2}$$

Здесь p обозначает условие (логическое выражение), а f — функцию. Запись (9.2) называется схемой потому, что заменяя соответствующим образом V, p и f, можно получить различные конкретные программы, имеющие одинаковую структуру и характер поведения.

Если обозначить через v_i значение переменной V после i-го выполнения S, то V в указанном порядке примет последовательность значений

$$v_0, v_1, \ldots, v_n \tag{9.3}$$

которые имеют следующие свойства:

$$\begin{array}{lll} 1. & v_i = f(v_{i-1}) & \text{для всех } i>0 \\ 2. & v_i \neq v_j & \text{для всех } i \neq j \\ 3. & \neg p(v_n) & \\ 4. & p(v_i) & \text{для всех } i<n \end{array} \tag{9.4}$$

Правило 1 следует из определения присваивания (7.9), правило 2 — из предыдущих рассуждений, правила 3 и 4 — из определения инструкции **while** (7.13). Заметим, что для V должно быть точно определено *начальное значение* v_0; нарушение этого правила является наиболее распространенной ошибкой в программировании. Итерации заканчиваются, если существует n, такое, что удовлетворяется условие 3.

Основной урок, который мы должны извлечь из сказанного, состоит в том, что инструкция **while** является подходящей формой для выражения программ, в основе которых лежит *рекуррентное соотношение* (9.4.1).

Пример: Вычисление факториала

Функцию

$$f(n)=n!=1*2*\ldots*n \quad n\geqslant 0 \tag{9.5}$$

можно вычислить с помощью программы, построенной по схеме (9.2) на основе рекуррентных соотношений

$$\begin{aligned} f(i)&=i*f(i-1)\\ f(0)&=1 \end{aligned} \tag{9.6}$$

В программе мы вводим переменные F и K, значения которых после i-го выполнения повторяемой инструкции равны соответственно $f(i)$ и i. То есть используются рекуррентные соотношения

$$\left.\begin{aligned} f_i&=k_i*f_{i-1}\\ k_i&=k_{i-1}+1 \end{aligned}\right\} \text{ для } i>0 \tag{9.7}$$

и начальные значения

$$f_0=1,\ k_0=0$$

. Программа, которая получается после подстановки рекуррентных соотношений (9.7) в схему (9.2), имеет вид

```
var F, K: integer; {n ⩾ 0}
begin F := 1; K := 0;
      while K ≠ n do
      begin {F = K!}
        K := K+1; F := K*F
      end
      {F = n!}
end
```
(9.8)

Так как K растет, принимая значения из последовательности натуральных чисел, и так как $n\geqslant 0$, то программа обязательно завершится. Следует обратить особое внимание на то, что существенно важен порядок, в котором выполняются две повторяемые инструк-

ции. Если их поменять местами:

$$F := K * F; \quad K := K+1 \tag{9.9.1}$$

то иными станут и соответствующие им рекуррентные соотношения:

$$\begin{aligned} f_i &= k_{i-1} * f_{i-1} \\ k_i &= k_{i-1} + 1 \end{aligned} \tag{9.9.2}$$

Несмотря на кажущееся сходство, рекуррентные соотношения (9.9.2) кардинально отличаются от (9.7)

Пример: Вычисление $1/x$

Пусть две последовательности вещественных чисел a_0, a_1, . . . и c_0, c_1, . . . определяются рекуррентными соотношениями

$$\left.\begin{aligned} a_i &= a_{i-1} * (1 + c_{i-1}) \\ c_i &= c_{i-1}^2 \end{aligned}\right\} \text{ для } i > 0 \tag{9.10}$$

и начальными значениями

$$a_0 = 1, \quad c_0 = 1 - x \quad 0 < x < 2$$

С помощью алгебраических преобразований можно показать, что

$$a_n = \frac{1 - c_n}{x} \tag{9.11}$$

А так как $c = c_0^{2^n}$ и $|c_0| < 1$,

$$\lim_{n \to \infty} a_n = \frac{1}{x} \tag{9.12}$$

Указание: Чтобы из (9.10) вывести (9.11), используйте равенства

$$a_n = (1 + c_{n-1}) * \ldots * (1 + c_1) * (1 + c_0)$$
$$\frac{1 + c_{i-1}}{1 - c_i} = \frac{1}{1 - c_{i-1}}$$

Подставляя рекуррентные соотношения (9.10) в схему (9.2), мы получим программу (9.13), которая вычисляет *приближенное значение* $1/x$, используя только операции сложения и умножения:

```
var A, C: real; {0 < x < 2}
begin A := 1; C := 1—x;                                   (9.13)
      while C > ε do
      begin {A * x = 1 — C, abs(C) < 1}
            A := A * (1 + C); C := sqr(C)
      end
      {(1 — ε)/x ≤ A < 1/x}
end.
```

Программа (9.13) завершится, когда C достигнет значения $c_n \leqslant \varepsilon$. Так как $|c_0| < 1$ (см. 9.10) и $c = c_0^{2^n}$, то для сколь угодно малого ε существует значение n, такое, что $c_n \leqslant \varepsilon$. Для всех $i < n$, однако, $c_i > \varepsilon$ (см. 9.4).

Пример: Вычисление квадратного корня

Пусть две последовательности вещественных чисел $a_0, a_1, \ldots$ и $c_0, c_1, \ldots$ определяются рекуррентными соотношениями

$$\left.\begin{aligned} a_i &= a_{i-1} * \left(1+\tfrac{1}{2}c_{i-1}\right) \\ c_i &= c_{i-1}^2 * \tfrac{1}{4}(3+c_{i-1}) \end{aligned}\right\} \text{ для } i>0 \qquad (9.14)$$

и начальными значениями

$$a_0=x, \quad c_0=1-x \quad 0<x<2$$

С помощью соответствующих алгебраических преобразований можно показать, что

$$a_n=\sqrt{x*(1-c_n)} \qquad (9.15)$$

Так как $|c_0|<1$,

$$\lim_{n\to\infty} c_n=0 \text{ и } \lim_{n\to\infty} a_n=\sqrt{x} \qquad (9.16)$$

Указание: Чтобы из (9.14) вывести (9.15) используйте равенства

$$a_n=\left(1+\tfrac{1}{2}c_{n-1}\right)*\left(1+\tfrac{1}{2}c_{n-2}\right)*\ldots*\left(1+\tfrac{1}{2}c_0\right)*x$$

$$\left(1+\tfrac{1}{2}c_{i-1}\right)=\frac{\sqrt{1-c_i}}{\sqrt{1-c_{i-1}}}$$

и

$$\frac{x}{a_n}=\frac{1}{\left(1+\frac{1}{2}c_0\right)\ldots\left(1+\frac{1}{2}c_{n-1}\right)}=$$

$$=\frac{\sqrt{1-c_{n-1}}}{\left(1+\frac{1}{2}c_0\right)\ldots\left(1+\frac{1}{2}c_{n-2}\right)\sqrt{1-c_n}}=\ldots=\frac{\sqrt{1-c_1}}{\left(1+\frac{1}{2}c_0\right)\sqrt{1-c_n}}=$$

$$=\frac{\sqrt{1-c_0}}{\sqrt{1-c_n}}=\frac{\sqrt{x}}{\sqrt{1-c_n}}$$

Подставив рекуррентные соотношения (9.14) в схему (9.2), получим следующую программу:

```
var A, C: real; {0 < x < 2}
begin A := x; C := 1-x;
    while c > ε do
    begin {A² = x*(1-C), C ≥ 0}
        A := A*(1+0.5*C);
        C := sqr(C)*(0.75+0.25*C)
    end
    {x*(1-ε) ≤ A² < x}
end.
```

(9.17)

Программа (9.17) обязательно завершится, поскольку (9.16.1) гарантирует нам, что для сколь угодно малого ε найдется n, такое, что $c_n \leqslant \varepsilon$.

9.2. Ряды

Циклическая инструкция годится для вычислений не только последовательностей, но также и *рядов* чисел. Если задана последовательность членов ряда

$$t_0,\ t_1,\ t_2,\ \ldots \tag{9.18}$$

то ряд частичных сумм

$$s_0,\ s_1,\ s_2,\ \ldots \tag{9.19}$$

определяется так, что

$$s_i = t_0 + t_1 + \ldots + t_i \tag{9.20}$$

Если последовательность задается рекуррентным соотношением

$$t_i = f(t_{i-1}) \text{ для } i>0 \tag{9.21}$$

то ряд определяется следующим образом:

$$\begin{gathered} s_i = s_{i-1} + t_i \text{ для } i>0 \\ s_0 = t_0 \end{gathered} \tag{9.22}$$

На рис. (9.23) показана схема программы, позволяющая с помощью соответствующей замены f и t_0 получить программы, которые на i-м повторении будут присваивать переменной S значения s_i.

```
T := t0; S := T;
while p(S, T) do
begin T := f(T); S := S+T
end
```
(9.23)

Из определения инструкции присваивания и циклической инструкции **while** можно вывести следующие отношения, определяющие основные свойства этой схемы программы:

1. $t_i = f(t_{i-1})$ для $i>0$
2. $t_i \neq t_j$ для $i \neq j$
3. $s_i = s_{i-1} + t_i$ для $i>0$
3. $\neg p(s_n, t_n)$
5. $p(s_i, t_i)$ для всех $i<n$

(9.24)

Пример: Вычисление приближенного значения exp(x)
Частичные суммы

$$s_i = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^i}{i!} \tag{9.25}$$

определяются рекуррентным соотношением

$$t_j = t_{j-1} * x/j \quad j>0 \tag{9.26}$$

и начальным значением $t_0=1$. Известно, что частичная сумма ряда (9.25) имеет предел и

$$\lim_{n\to\infty} s_n = exp\,(x) \tag{9.27}$$

Ряд сходится для любого вещественного x; то есть члены ряда убывают так, что их сумма стремится к конечному пределу. Этот факт используется в программе (9.28), которая получается подстановкой рекуррентных соотношений в схему (9.23).

```
var T, S: real; K: integer;
begin T := 1; S := T; K := 0;
      while T > ε do
      begin {S = 1+x+ ... +x^K/K!, T = x^K/K! > ε}        (9.28)
            K := K+1; T := T*x/K; S := S+T
      end
end.
```

Погрешность вычисления, которая является разностью конечного значения S и истинной предельной суммы, равна $\sum_{i=K+1}^{\infty} (x^i/i!)$ и может быть сколь угодно малой при достаточно большом K. Обычно процесс суммирования заканчивается в зависимости от величины членов ряда *относительно* общей суммы, а не от абсолютной величины последнего члена ряда. Это требует, однако, дальнейшего исследования сходимости ряда, особенно если ряд знакопеременный. В следующем примере как раз и рассматривается этот случай.

Пример: Вычисление приближенного значения sin (x)
Частичные суммы ряда

$$s_i = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^{2i-1} * \frac{x^{2i-1}}{(2^i-1)!} \tag{9.29}$$

содержат члены, определяемые рекуррентными соотношениями

$$\begin{aligned} t_j &= -t_{j-1} * \frac{x^2}{k_j * (k_j-1)} \\ k_j &= k_{j-1}+2 \end{aligned} \tag{9.30}$$

при $j>0$ и начальных значениях $t_0=x$ и $k_0=1$. Известно, что ряд s сходится и

$$\lim_{n\to\infty} s_n = sin\,(x) \tag{9.31}$$

Программа получается с помощью подстановки рекуррентных соотношений (9.30) в схему (9.23).

```
var S, T : real; K : integer;
begin T := x; K := 1; S := T;
    while abs(T) > ε * abs(S) do                          (9.32)
    begin K := K+2; T := —T * sqr (x)/(K * (K—1));
          S := S+T
    end
end.
```

Следует обратить особое внимание на то, что в программах (9.13), (9.17), (9.28) и (9.32) число необходимых членов ряда и число повторений определяется не просто. Эти числа зависят от значения ε (оно входит в условие окончания цикла и определяет требуемую точность), а также от *быстроты сходимости* ряда. Следовательно, использование такого рода рекуррентных соотношений в программировании требует большой осторожности, даже если математический анализ гарантирует нам абсолютную сходимость, так как на практике важна быстрая сходимость (ср. упражнения 9.3 и 9.8).

Ряд, частичные суммы которого имеют вид (9.25), сходится быстро только в случае малых положительных значений x. Поэтому мы рекомендуем пользоваться соотношениями

$$exp\ (—x)=1/exp\ (x) \quad \text{для} \quad x<0 \tag{9.33.1}$$

и

$$exp\ (i+y)=exp\ (i) * exp\ (y) \quad \text{для} \quad x>1 \tag{9.33.2}$$

где $i=trunc\ (x)$ и $y=x—i$. Значение $exp\ (i)$ при этом вычисляется просто многократным умножением основания натурального логарифма.

Ряд, частичные суммы которого имеют вид (9.29), называется *знакопеременным* и сходится быстро только в случае малых значений x. Поэтому рекомендуется использовать следующие соотношения для больших значений x:

$$\begin{array}{ll}
sin\ (x)=sin\ (x—2\pi n) & \text{для } 2\pi n\leqslant|x|<2\pi\ (n+1)\\
sin\ (x)=—sin\ (x—\pi) & \text{для } \pi\leqslant|x|<2\pi\\
sin\ (x)=sin\ (\pi—x) & \text{для } \frac{\pi}{2}\leqslant|x|<\pi\\
sin\ (x)=cos\ (\frac{\pi}{2}—x) & \text{для } \frac{\pi}{4}<|x|<\frac{\pi}{2}\\
sin\ (x)=—sin\ (—x) & \text{для } x<0
\end{array} \tag{9.34}$$

Благодаря этим формулам программу (9.32) можно использовать для значений x, удовлетворяющих $0\leqslant|x|\leqslant\pi/4$. А в этом интервале быстрота сходимости практически нас удовлетворяет и число вычисляемых членов ряда достаточно мало, причем ошибки округле-

ния и неточного представления чисел (из-за использования конечной арифметики) остаются в допустимых пределах.

Наконец, на примере программы (9.32) мы продемонстрируем использование еще одного основного правила программирования. Внутри повторяемой инструкции вычисляется значение x^2. Возведение в квадрат выполняется многократно, хотя x на протяжении всего счета остается неизменным. Вычисление x^2 можно сделать один раз (перед инструкцией **while**), присвоить полученное значение некоторой вспомогательной переменной, а последнюю подставить во всех местах, где встречается вычисление x^2. Этот прием можно сформулировать как следующее основное правило:

Если выражение $f(x)$ вычисляется внутри повторяемой инструкции S и если аргумент x не изменяется в цикле, то следует ввести вспомогательную переменную h, которой один раз перед выполнением S присваивается значение $f(x)$ и которая подставляется вместо $f(x)$ внутри S, т. е. конструкция

while P **do**
begin ... $f(x)$ ... **end** (9.35.1)

заменяется на

$h := f(x)$;
while P **do**
begin ... h ... **end** (9.35.2)

УПРАЖНЕНИЯ

9.1 Используя формулы (9.33) и (9.34), перепишите программы (9.28) и (9.32) так, чтобы функции *exp* (x) и *sin* (x) вычислялись эффективно (и более точно).

9.2 Напишите программу вычисления *cos* (x) с относительной точностью ε. Используйте схему программы (9.23) и ряд

$$cos\,(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

9.3 Используя схему (9.23), напишите программу вычисления интеграла

$$\int_0^x exp\,(-u^2)\,du = x - \frac{x^3}{3*1!} + \frac{x^5}{5*2!} - \frac{x^7}{7*3!} + \ldots$$

Указание: Обратите внимание, что для $x>1$ сходимость медленная. Чего следует ожидать, если используется вычислительная машина с конечной точностью и конечным диапазоном значений? (Проследите, например, за вычислением членов ряда для $x=1, 2, 3 \ldots$.)

9.4 Напишите программу в соответствии со схемой (9.2) для получения чисел Фибоначчи двумя различными методами:

(а) с использованием рекуррентных соотношений

$$f_{i+1} = f_i + f_{i-1} \quad i>0$$
$$f_0 = 0 \quad f_1 = 1$$

(b) с использованием формулы

$$f_i = roind(c^i/\sqrt{5})$$

где $c=(1+\sqrt{5})/2$.

Вместо $\sqrt{5}$ используйте приближенное значение 2.236068 и определите наименьшее i, для которого два вычисленных значения f_i отличаются.

9.5 Проверьте результаты программ (9.13) и (9.17) с помощью вывода необходимых и достаточных утверждений после каждой инструкции из указанных инвариантов.

9.6 Исходя из схемы (9.2), получите программу вычисления логарифма с основанием 2 от вещественного числа. Используйте соотношения

$$\begin{aligned} log(x) &= -log(1/x) && \text{для } 0<x<1 \\ log(x) &= n+log(x/2^n) && \text{для } x>2^n \end{aligned}$$

Для $1\leqslant x<2$ используйте рекуррентные соотношения

$$a_i = \begin{cases} a_{i-1}^2 & \text{если } a_{i-1}^2 < 2 \\ \frac{1}{2}a_{i-1}^2 & \text{если } a_{i-1}^2 \geqslant 2 \end{cases}$$

$$b_i = \frac{1}{2}b_{i-1}$$

$$s_i = \begin{cases} s_{i-1} & \text{если } a_{i-1}^2 < 2 \\ s_{i-1}+b_i & \text{если } a_{i-1}^2 \geqslant 2 \end{cases}$$

для $i>0$ и $a_0=x$, $b_0=1$, $s_0=0$. Вычисление должно заканчиваться, когда для достаточно малого ε выполняется неравенство $b_n\leqslant\varepsilon$. Найдите инвариант цикла, проверьте, что

$$\lim_{n\to\infty} s_n = log(x)$$

и установите, что алгоритм всегда завершается.

9.7 Что будет, если в программах (9.13) и (9.17) ε заменить на 0, а в программах (9.28) и (9.32) условие окончания цикла заменить условием $s=s+t$? Закончится ли вычисление по каждой из этих программ, если принять во внимание конечную точность арифметики с плавающей точкой.

9.8 Выполните на машине программы (9.13), (9.17), (9.28) и (9.32). Предусмотрите в этих программах счетчики, подсчитывающие число выполняемых циклов. Экспериментально определите, как зависит быстрота сходимости от различных значений аргумента x.

9.9 Ниже приводится программа, которая была написана для вычисления суммы s первых n членов разложения синуса в ряд (9.29) для аргументов x, удовлетворяющих условию $0\leqslant x\leqslant\pi/4$. Правильная ли эта программа? Какие критические замечания можно сделать по ее поводу?

```
var i, v: integer; h, u, s: real;
begin i := 0; u := x; v := 1;
      h := sqr(x); s := u;
      repeat i := i+2; u := -u*h;
             v := v*i*(i+1); s := s+u/v
      until i > 2*n
end.
```

9.10 Задана относительная точность представления вещественных чисел на машине — $\varepsilon=10^{-6}$ (10^{-10}, 10^{-14}). Определите, сколько потребуется (в худшем случае) членов ряда (9.25) и (9.29), чтобы получить максимально возможную точность вычисления $exp(x)$ для $0\leqslant x<1$ и $sin(x)$ для $0\leqslant x<\pi/4$ (ср. с 8.28).

10. ФАЙЛОВАЯ СТРУКТУРА ДАННЫХ

10.1. Понятие файла

Для всех рассмотренных ранее типов данных наиболее характерны два свойства: неделимость и упорядоченность их значений. Поэтому они и называются скалярными типами. Например, каждое значение типа *integer* (т. е. каждое целое число) есть объект, не распадающийся на компоненты, а множество целых чисел упорядочено. Следовательно, совершенно бессмысленно ссылаться на i-ю цифру (компоненту) целого числа; можно, однако, говорить об i-й цифре десятичного представления целого числа, которое само по себе является не целым числом, а последовательностью литер. Поэтому в данном случае, очевидно, удобно иметь возможность адресоваться к представлению числа как к одному объекту, хотя он и состоит из отдельных цифр. Эта возможность давать коллективное имя всему множеству элементов имеет большое значение в обработке данных вообще. Такие множества значений или переменных с одним общим именем называются *структурными*. Существует несколько методов структурирования, каждый из которых отличается способом доступа к индивидуальным компонентам и, следовательно, способом обозначения этих компонент.

Переменные, состоящие из нескольких компонент, называются *структурными переменными*. Для определения типа (диапазона значений) структурной переменной необходимо задать

(a) способ структурирования и
b) тип(ы) ее компонент.

Во многих отношениях простейшей структурой данных является *последовательность*. По-видимому, наиболее известным примером переменной с *последовательной структурой* является *колода карт*. В области обработки данных для описания последовательности часто используется термин *последовательный файл*. Здесь мы будем говорить просто файл, подразумевая, что он «последовательный». Чтобы определить тип F, т. е. диапазон значений, которые являются последовательностями компонент типа T, мы используем запись

type F=**file of** T (10.1)

Очевидно, все компоненты последовательностей, определенных таким образом, должны быть *одного и того же* типа T.

Пример:

type *text*=**file of** *char*

Такой (заметим, кстати, бесконечный) диапазон значений можно определить формально с помощью операции конкатенации: конкатенация двух файлов

$$\alpha=\langle x_1, x_2 \ldots x_m\rangle \text{ и } \beta=\langle y_1, y_2 \ldots y_n\rangle$$

обозначается как

$$\alpha\cdot\beta=\langle x_1 \ldots x_m, y_1, y_2 \ldots y_n\rangle \qquad (10.2)$$

Тогда диапазон значений типа F, соответствующий (10.1), строго определяется следующими аксиомами:

1) $\langle\ \rangle$ есть файл типа F (пустая последовательность),
2) если f есть файл типа F и t есть объект типа T, то $f\cdot\langle t\rangle$ есть файл типа F,
3) никакие другие значения не являются файлами типа F.

Описание файловой переменной f соответствует соглашениям, принятым в гл. 8:

var f : F или **var** f : **file of** T (10.3)

Ее значением по определению всегда будет файл типа F. По причинам, которые мы объясним в разд. 10.2, мы предполагаем, что с каждым описанием новой файловой переменной f автоматически вводится дополнительная переменная типа T. Такую переменную мы будем называть *буферной переменной* и обозначать $f\uparrow$. Она используется либо для добавления новой компоненты в конец файла, либо для выборки компонент из файла при просмотре.

Файлы играют важную роль в любой вычислительной системе. Файловая структура соответствует данным, хранящимся на устройствах, в которых используется механическое перемещение, и поэтому последовательный доступ к компонентам часто оказывается единственно возможным, так как ячейки памяти проходят под головками чтения или записи в строго последовательном порядке. Операции, которые вычислительная система может выполнять на различных устройствах памяти, определяются принципами, заложенными в их конструкции. В следующих широко используемых устройствах данные обычно хранятся в виде последовательных файлов.

1. *Магнитные ленты, диски и барабаны.* Возможно чтение, запись и стирание (при смене позиции и повторной записи).
2. *Читающие устройства с перфокарт и перфоленты, устройства вывода на перфокарты и перфоленту.* Для читающих устройств возможно только последовательное чтение. Файл,

представленный колодой карт или перфолентой, установленной на читающем устройстве, называется *файлом ввода*. Аналогично колода карт или перфолента, получаемая из устройства вывода, называется *файлом вывода*.

3. *Печатающие устройства.* Соответствующий файл, компонентами которого являются литеры, предназначенные для печати, называется файлом вывода.

Привязка файлов к тем или иным конкретным устройствам и вытекающие из этого ограничения на правила работы с файлом задаются так называемой *диспозицией*. Но поскольку нас не интересует физическое представление данных, то нам не нужны языковые средства для задания диспозиции.

Понятие файла используется как *абстракция* данных, хранящихся на любом из запоминающих устройств, и позволяет единообразно формулировать их общие характеристики, а также операции над ними. В следующих разделах описываются наиболее важные операции с файлами.

10.2. Генерирование файла

При описании файловой переменной указывается ее имя, структура и тип. Кроме того, подразумевается, что вначале число компонент равно нулю. Число компонент называется *длиной* файла. Файл с нулевой длиной называется *пустым* и обозначается как $\langle\ \rangle$. Длина растет по мере того, как добавляются новые компоненты с помощью стандартного файлового оператора *put*. Оператор *put* (f) мы определяем как процедуру, которая добавляет в конец файла f одну-единственную компоненту. Значение этой компоненты копируется из буферной переменной, которая появляется неявно в связи с описанием файла f (10.2). Формально результат выполнения инструкции *put* (f) можно выразить следующим образом:

$$\{(f=\alpha)\wedge(f\uparrow=x)\}\quad put\ (f)\quad \{f=\alpha\cdot\langle x\rangle\} \tag{10.4}$$

В целях более эффективной реализации файлов на реальных вычислительных машинах выгодно запретить пользоваться значением буферной переменной $f\uparrow$ после операции *put* (f). Поэтому мы считаем, что после операции *put* (f) значение переменной $f\uparrow$ не определено.

Пример: Генерирование файла

Требуется сгенерировать файл целых чисел, в котором значение каждой i-й компоненты равно i^2 и который содержит квадраты всех натуральных чисел, не превосходящих n. (Заметим, что из рекуррентных соотношений

$$\left.\begin{aligned} a_i &= a_{i-1}+b_i \\ b_i &= b_{i-1}+2 \end{aligned}\right\}\ \text{для}\ i>0 \tag{10.5}$$

и $a_1=b_1=1$ непосредственно следует, что $a_i=i^2$.)

```
var A, B:integer;
    f:file of integer;
begin A := 1; B := 1;
    repeat {A=(B+1)²/4}
        f↑ := A; put (f);
        B := B+2; A := A+B
    until A ⩾ n
end.
```
(10.6)

10.3. Просмотр файла

После того как генерирование файла окончено, он пригоден для просмотра. *Просмотр* осуществляется строго последовательно, начиная с первой компоненты файла. Во время просмотра важной характеристикой файла является его *позиция*, которая отделяет уже просмотренную часть файла от еще не прочитанной. Например, если файл находится на магнитной ленте, то эта позиция соответствует положению ленты относительно читающей головки.

При *генерации файла* его позиция задается неявно, поскольку новые компоненты добавляются *в конец* файла, в то время как в процессе *просмотра файла* его позиция должна быть указана *явно*. Удобным явным заданием позиции может служить указание двух частей файла: $\overleftarrow{f}$ — уже прочитанной части файла, и $\vec{f}$ — еще не прочитанной части файла. При конкатенации этих двух частей всегда получается весь файл:

$$f=\overleftarrow{f}\cdot\vec{f} \tag{10.7}$$

Текущей просматриваемой компонентой можно тогда считать первый элемент $\vec{f}$, т. е.

$$first(\langle x_1, x_2, \ldots, x_n\rangle)=x_1$$

Теперь введем стандартный файловый оператор *reset* (*f*), который вызывает установку файла в начальную позицию. Формально эту операцию можно описать следующим образом:

$$\{f=\alpha\}\quad reset\ (f)\quad \{(\overleftarrow{f}=\langle\,\rangle)\wedge(\vec{f}=\alpha)\wedge(f\uparrow=first\ (\vec{f}))\} \tag{10.8}$$

В этом определении предусматривается также присваивание буферной переменной $f\uparrow$ значения первой компоненты файла, если, конечно, она есть. Для перехода к следующей компоненте мы вводим оператор *get*(*f*). При выполнении этого оператора позиция файла смещается на одну ячейку вправо и переменной $f\uparrow$ присваивается значение следующей компоненты. Формально оператор *get* можно опре-

делить следующим образом:

$$\{(\overleftarrow{f} = \alpha) \wedge (\overrightarrow{f} = \langle x \rangle \cdot \beta)\}\ get\,(f)$$

$$\{(\overleftarrow{f} = \alpha \cdot \langle x \rangle) \wedge (\overrightarrow{f} = \beta) \wedge (f\uparrow = first\,(\overrightarrow{f}))\} \qquad (10.9)$$

Заметим, что равенство

$$f\uparrow = first\,(\overrightarrow{f})$$

встречается в консеквентах операторов *reset* и *get*. Однако функция получения первого элемента (*first*) определена только в том случае, когда $\overrightarrow{f}$ содержит по крайней мере одну компоненту, т. е. если $\overrightarrow{f}$ не пустая часть файла. Поэтому необходимо иметь возможность проверить, является ли $\overrightarrow{f}$ пустой частью. Эту возможность мы предоставляем в виде следующей логической функции (*eof* — end of file — конец файла):

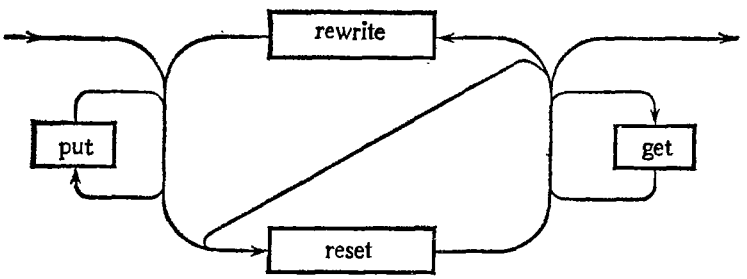
$$eof\,(f) \equiv \overrightarrow{f} = \langle\ \rangle \qquad (10.10)$$

Следовательно, после операций *reset* (*f*) и *get* (*f*) значение $f\uparrow$ определено только в случае $\neg eof\,(f)$.

Помимо операторов *put*, *reset*, *get* и функции *eof* введем четвертый и последний файловый оператор *rewrite* (*f*). Он позволяет уничтожить значение файловой переменной и дает возможность генерировать новый файл с тем же именем. Действие этого оператора формально можно описать следующим образом:

$$rewrite\,(f) \quad \{f = \langle\ \rangle\} \qquad (10.11)$$

Допустимые последовательности операций над файлами представлены на диаграмме (10.12). На этом графе ясно показано, что «фаза» генерирования файла во времени отделена от «фазы» просмотра файла.



(10.12)

Наиболее подходящей конструкцией для описания просмотра файла, если учесть его последовательную структуру, является циклическая инструкция. После обработки очередной компоненты файла в инструкции S переменной $f\uparrow$ с помощью операции *get* (*f*)

присваивается значение следующей компоненты и позиция файла смещается на одну ячейку. Это представлено схемой программы (10.13).

```
while ¬ eof(f) do
      begin S; get(f)
      end
```
(10.13)

Если файл f не пустой (и только в этом случае), можно использовать схему (10.14). Тогда схема (10.14) эквивалентна схеме (10.13):

```
repeat S; get(f)
until eof(f)
```
(10.14)

Пример: Подсчет количества компонент файла

Требуется вычислить длину файла f и полученное значение присвоить переменной L. Используя схему (10.13) и подставляя $L := L+1$ вместо S, получим программу (10.15).

```
var L:integer;
begin L := 0;
      while ¬ eof(f) do
      begin {L=число прочитанных компонент}
         L := L+1; get(f)
      end
end.
```
(10.15)

Пример: Вычисление среднего значения и дисперсии

Дано $n>0$ измерений x_i, представленных в виде файла f вещественных чисел. Необходимо вычислить среднее значение m и стандартное отклонение s по формулам

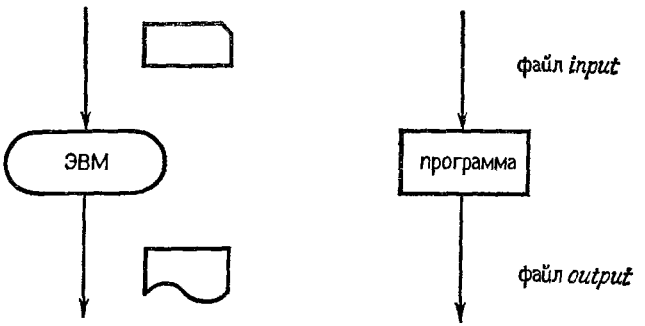
$$m = \frac{1}{n}\sum_i x_i, \qquad s^2 = \frac{1}{n}\sum_i x_i^2 - m^2 \tag{10.16}$$

Воспользовавшись схемой (10.14), получим программу (10.17).

```
var f:file of real;
    m, s:real; n:integer;
begin {генерирование f}
      n := 0; m := 0; s := 0;
    repeat n := n+1; m := m+f↑; s := s+sqr(f↑);
      {n=количество, m=сумма, s=сумма квадратов
                прочитанных значений} get(f)
    until eof(f)
    m := m/n; s := sqrt(s/n-sqr(m))
end.
```
(10.17)

10.4. Текстовые файлы

Файл, компонентами которого являются литеры, предназначенные для печати, называется *текстовым файлом*. Текстовые файлы играют большую роль при обработке данных, поскольку в большинстве программ для ввода и вывода данных используются текстовые файлы. Перфоленты, подготовленные на телетайпе, колоды карт, полученные с устройств подготовки перфокарт, результаты вычислений, печатаемые на бумажной ленте, поток данных между терминальными телетайпами и вычислительной машиной — все это рассматривается как текстовые файлы. Можно считать, что вычислительный процесс, использующий любую из этих сред в качестве носителя данных, осуществляет преобразование одного текстового файла, называемого *input* (ввод), в другой текстовый файл, называемый *output* (вывод).



Поэтому удобно ввести стандартный тип и две *стандартные переменные*. Предполагается, что они определены в любой вычислительной системе, т. е. неявно описаны следующим образом:

```
type text = file of char
var input, output: text
```
(10.18)

Предполагается, что эти две переменные в любой вычислительной системе обозначают носители ввода и вывода. Поэтому разумно ввести для них следующие ограничения.

1. Файл *input* можно просматривать только один раз, его нельзя повторно генерировать и нельзя вернуться к какой-либо прежней позиции. Следовательно, допустима только одна операция *get*. Генерирование этого файла (см. 10.12) происходит заранее до запуска соответствующей программы.

2. Файл *output* можно только генерировать, его нельзя читать и нельзя смещать позицию. Следовательно, допустима только одна операция *put*. Просмотр этого файла (см. 10.12) осуществляется после завершения соответствующей программы.

Относительно управляющей литеры **eol** (см. разд. 8.3) следует помнить следующее:

1. Если файл *input* представлен колодой карт, то он состоит из литер, расположенных вплотную друг к другу в том порядке, как они были пробиты на перфокартах. Концу каждой карты в файле *input* соответствует литера **eol**.
2. Если файл *output* печатается, то каждая литера **eol** включает механизм транспорта бумаги в печатающем устройстве.

Поскольку эти два стандартных файла присутствуют практически в каждой законченной программе, мы вводим сокращения (10.19), которые применяются только в отношении этих двух файлов. Здесь c, c_1, . . ., c_m обозначают переменные, e, e_1, . . ., e_n— выражения типа *char*.

Полная запись	Сокращенная запись
$c := input\uparrow$; *get* (*input*)	*read* (*c*)
$output\uparrow := e$; *put* (*output*)	*write* (*e*)
read (c_1); . . .; *read* (c_m)	*read* (c_1, . . ., c_m)
write (e_1); . . .; *write* (e_n)	*write* (e_1, . . ., e_n)

(10.19)

Пример: Исключение лишних пробелов

Непустой файл *input* копируется в файл *output* так, что каждая последовательность подряд стоящих пробелов заменяется одним пробелом. Предполагается, что последняя литера файла *input* не есть пробел. Заменяя соответствующим образом инструкцию S в схеме (10.13), получим программу (10.20).

```
var  c:char;
begin while ¬ eof(input) do
      begin read (c); write(c);
          if c = ' ' then
          begin repeat read(c) until c ≠ ' ';
                write(c)
            end
      end
end.
```

(10.20)

Пример: Графическое представление функции

Вещественная функция f представляется в виде графика на построчно печатающем устройстве. Можно, например, печатать звездочку в позициях, соответствующих координатам (x_0, y_0), ... , (x_n, y_n), где $x_0, \ldots, x_n$— равноотстоящие абсциссы. Предположим, что ось абсцисс направлена вдоль бумажной ленты и масштаб выбран так, что $d=x_i-x_{i-1}$ соответствует расстоянию между соседними строками. Чтобы определить положение звездочки с координатой x_i, необходимо вычислить $y_i=f(x_i)$, умножить y_i на масштабный множитель s и округлить произведение до ближайшего целого числа. Полученное значение соответствует числу пробелов, предшествующих звездочке в строке. В программе (10.21) для конкретности предполагается, что

$f(x)=exp(-x)*sin(2\pi x) \quad 0\leqslant x<4$

$d=1/32$ (32 строки на интервал $[x, x+1]$)

$s=50$ (50 позиций в строке соответствует интервалу $[y, y+1]$)

$h=65$ (расстояние оси x от левого края бумаги, выраженное позицией литеры в строке).

```
const d = 0.03125; s = 50; h = 65; c = 6.28318; lim = 128;
var x, y: real; i,n: integer;
begin i := 0;
    repeat x := i * d; y := exp (— x) * sin (c * x);
        n := round (s * y) + h                                    (10.21)
        repeat write (' '); n := n — 1
        until n = 0;
        write ('*', eol); i := i + 1
    until i = lim
end.
```

С помощью литеры **eol** текстовые файлы разбиваются на отдельные строки. Это типичный пример разбиения файла на *логические секции*. Секции (в данном случае *строки*) могут состоять из разного числа компонент (в данном случае *литер*). Естественно, разбиение файла на логические секции оказывает влияние на структуру программы просмотра этого файла. Лучше всего рассматривать файл как последовательность секций, каждая из которых заканчивается специальной литерой-разделителем, а сами секции можно рассматривать как последовательности компонент исходного файла.

Наиболее приемлемой структурой для такой программы просмотра будет *вложенный цикл*. Во «внешнем» цикле при каждом повторении обрабатывается одна секция, а во «внутреннем» — одна компонента. Сказанное можно выразить схемой программы (10.22), в которой учитывается специальный случай, когда секциями являются строки текстового файла, а разделителем — литера *eol*.

```
while ¬ eof (input) do
begin read (c); S1;
      while c ≠ eol do
      begin S2; read (c)
      end;
      S3
end
```
(10.22)

В этой схеме инструкция $S2$ специфицирует действия, выполняемые применительно к каждой компоненте файла, кроме разделителей строк. Инструкции $S1$ и $S3$ специфицируют действия, которые должны выполняться соответственно в начале и в конце каждой строки.

Если достоверно известно, что файл содержит по крайней мере одну строку, а каждая строка содержит по крайней мере одну литеру (помимо **eol**), то вместо (10.22) можно воспользоваться схемой (10.23).

```
repeat read (c); S1;
       repeat {c ≠ eol}
           S2; read (c)
       until c = eol;
       S3
until eof(input)
```
(10.23)

Пример: Включение в текст литер, управляющих печатью

В практике использования алфавитно-цифровых устройств очень часто первая литера каждой строки воспринимается устройством не как обычная литера, которая должна быть напечатана, а как управляющая литера. Эта литера управляет транспортом бумаги при переходе на следующую строку или следующую страницу. Соглашение возникло как результат широкого применения техники буферизации данных вывода (например, если данные необходимо сохранить до тех пор, пока не освободится устройство). Поэтому нужно, чтобы данные, поступающие на печатающее устройство из буферной памяти, содержали необходимую управляющую информацию, и включение ее в файл в виде дополнительной литеры в начале каждой строки имеет очевидные преимущества. Ниже приводятся некоторые из таких управляющих литер.

' ' (пробел) обычный интервал между строками
'0' двойной интервал между строками
'1' переход к следующей странице

В этом примере программа копирует файл *input* в файл *output* и при этом в начало каждой строки вставляет один пробел, чтобы при печати между строками был обычный интервал. Программа (10.24) получается из схемы (10.22) подстановкой соответствующих инструкций *write* вместо *S1*, *S2* и *S3*.

```
var c:char;
begin
while ¬ eof (input) do                                    (10.24)
    begin read (c);
      write (' '); {управляющая литера для печатающего устройства}
       while c≠eol do
          begin write (c); read (c)
          end;
       write (eol);
    end
end.
```

УПРАЖНЕНИЯ

10.1 Даны два файла ввода f и g, содержащие упорядоченные последовательности целых чисел

$$f_1, f_2, \ldots, f_m \text{ и } g_1, g_2, \ldots, g_n$$

такие, что

$$f_{i+1} \geqslant f_i \quad \text{и} \quad g_{j+1} \geqslant g_j \quad \text{для всех } i, j$$

Напишите программу, которая сливает эти два файла в один упорядоченный файл h, такой, что

$$h_{k+1} \geqslant h_k \quad \text{для } k = 1, \ldots, m+n-1$$

10.2 Дополните программу (10.21) так, чтобы кроме функции $f(x)$ печаталась также ось x.

10.3 Измените программу (10.20) так, чтобы

(a) в начало каждой строки вставлялся пробел (в качестве управляющей литеры) и

(b) не исключались пробелы, если они встречаются в начале строки.

10.4 Предположим, что в основном наборе имеются два разных разделителя: **eop** (end of paragraph) — «конец параграфа» и **eol** — «конец строки». Пусть также задан текстовый файл *input*, который с помощью **eop** подразделяется на параграфы, а с помощью **eol** параграфы разбиваются на строки. Постройте схему программы, которая просматривает файл и выполняет инструкции:

S1 в начале каждого параграфа,
S2 в начале каждой строки,
S3 для каждой литеры в строке,
S4 в конце каждой строки и
S5 в конце каждого параграфа.

Предполагается, что **eop** может встретиться только непосредственно за **eol** или другими **eop** и что файл заканчивается литерой **eop**. *Указание*: Обобщите схему программы (10.22).

10.5 Задан файл f чисел x_i. Последовательность $x_i \ldots x_k$ называется *цепочкой*, если ее члены упорядочены ($x_i \leqslant x_{i+1} \leqslant \ldots \leqslant x_{k-1} \leqslant x_k$). Цепочка называется максимальной, если также $x_{i-1} > x_i$ и $x_k > x_{k+1}$. Величина $k-i+1$ называется длиной цепочки.

Напишите программу, которая определяет длину самой длинной из максимальных цепочек заданного файла f.

11. МАССИВ КАК СТРУКТУРА ДАННЫХ

Подобно файлу, переменная, имеющая структуру массива, является совокупностью *компонент-переменных одного и того же типа.* У массива, однако, есть характерные особенности, которые позволяют четко различать эти два вида структур.

1. Каждая отдельная компонента массива может быть явно обозначена и к ней имеется прямой доступ.
2. Число компонент массива определяется при его описании и в дальнейшем не меняется.

Эти особенности массива требуют определенных соглашений в нотации, чтобы можно было

(a) обозначать отдельные компоненты массива и
(b) определять типы для массивов.

Для обозначения компонент используется имя переменной-массива и так называемый *индекс*, который однозначно указывает желаемый элемент. Тот факт, что индекс может быть вычисляемым объектом, выделяет массив среди многих других структур данных. Индексы, следовательно, должны иметь один из доступных типов данных, который называется *типом индекса* массива. Мы условимся, что типами индексов могут быть только скалярные типы. Поэтому все элементы массива линейно упорядочены. Теперь мы постулируем следующие соглашения, касающиеся нотации:

1. Определение типа для массивов включает спецификации типа компонент и типа индексов. Существует взаимно однозначное соответствие между компонентами массива и значениями индекса. Определение типа для массивов имеет следующий вид:

   type A=**array** [$T1$] **of** $T2$ (11.1)

   где A — идентификатор нового типа, $T1$ — тип индекса, а $T2$ обозначает тип компонент. Ниже приводятся примеры описания массивов (с безымянными типами).

   var x : **array** [1..20] **of** *real* (11.2)
   var y : **array** [*color*] **of** *color*

В соответствии с этими описаниями x состоит из 20 компонент типа *real* со значениями индекса 1, 2, . . ., 20 и y имеет четыре компоненты типа *color* (см. гл. 8) со значениями индекса *красный*, *желтый*, *зеленый*, *голубой*.

2. Элемент массива A, соответствующий значению индекса i, обозначается как $A[i]$ или A_i. Примеры (см. 11.2):

$$\begin{array}{ll} x\,[10] & x\,[i+j] \\ y\,[\textit{красный}] & y\,[y\,[\textit{желтый}]] \end{array} \qquad (11.3)$$

Два массива считаются равными тогда и только тогда, когда их типы совпадают и все соответствующие компоненты попарно равны, т. е.

$$u=v \leftrightarrow u_i=v_i \quad \text{для всех } i \qquad (11.4)$$

Поскольку каждую компоненту можно явно обозначить, естественно потребовать, чтобы элементы массива (в памяти вычислительной машины) были одинаково доступны. Тем самым налагаются серьезные ограничения на выбор запоминающих устройств, пригодных для представления переменных, являющихся массивами. В частности, для этой цели не годятся все устройства с последовательной выборкой, такие, как перфоленты, магнитные ленты и (в несколько меньшей степени) диски. Крайне желательно иметь запоминающие устройства с одинаковым временем выборки из любой ячейки памяти. Самыми известными представителями такого класса устройств являются память на магнитных сердечниках и полупроводниковые запоминающие устройства на интегральных схемах, в которых нет механически перемещающихся деталей. Такие устройства называются *запоминающими устройствами с произвольным доступом*, поскольку к любой случайно выбранной ячейке можно получить немедленный доступ. Но стоимость таких устройств значительно выше стоимости устройств с последовательным доступом. Поэтому последние всегда используются там, где необходимо хранить большие объемы данных. Но если объем данных таков, что они укладываются в прямо адресуемую память машины — так называемую *первичную память*,— то такие данные целесообразно представить в виде массива. В противоположность первичной памяти имеющиеся в вычислительной системе запоминающие устройства большой емкости с последовательным доступом называются *вторичной памятью*.

Пример: Поиск компоненты массива

Задан массив с компонентами $A[1]$, . . ., $A[n]$ и значение x. Присвоить q значение *false*, если нет k, такого, что $A[k]=x$. В противном случае q присвоить значение *true*, а переменной i при-

своить значение k.

```
var i: 0 .. n; q:Boolean; {n > 0}
      A:array [1 .. n] of T;
begin {присваивание значений массиву A}          (11 5)
    i := 0;
    repeat i := i+1; q := A[i]=x
           {A[j]≠x для j=1 ... i—1}
    until q ∨ (i=n)
end.
```

Поиск закончится, если выполнится хотя бы одно из двух условий: q (искомая компонента найдена) и $i=n$ (все компоненты отличаются от x). Чтобы упростить это составное условие и тем самым ускорить работу алгоритма, обычно применяют следующий прием:

1. К массиву A добавляется еще одна компонента.
2. Новой компоненте присваивается значение x, и оно служит *граничным признаком*, который гарантирует окончание поиска.

В результате получается следующая программа:

```
var   i:0 .. n1; {n1 := n+1}
      A:array [1 .. n1] of T;                    (11.6)
begin {присвоить значения компонентам A[1] ... A[n]}
      i := 0; A[n1] := x;
      repeat i := i+1 until A[i]=x;
      {A[j]≠x для j=1 ... i—1}
end.
```

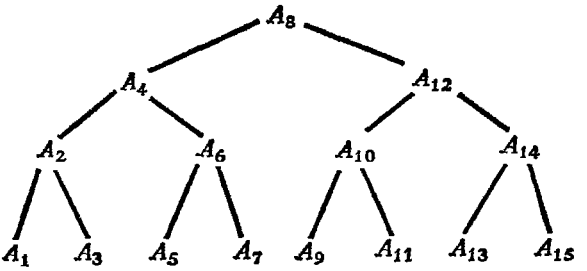
Пример: Поиск компоненты в упорядоченном массиве

Задача та же самая, что и в предыдущем примере. Но теперь компоненты массива упорядочены так, что $A[i]<A[j]$ для любых $i<j$. Хотя программу (11.5) можно применить и в этом случае, мы рекомендуем воспользоваться упорядоченностью массива и применить более эффективный поиск. Улучшенный метод поиска основан на следующей идее.

Компоненты массива будем считать узлами дерева, которое в каждом узле имеет не более двух поддеревьев (и поэтому называется *бинарным деревом*). На рис. (11.7) представлено «дерево» с $n=15$ узлами.

Если вначале взять из середины массива компоненту с индексом $k=(1+n)/2$ и сравнить ее с x, то возможны три случая.

1. $A[k]=x$ — искомая компонента найдена.
2. $A[k]<x$ — в левом поддереве не может быть компоненты равной x, так как все индексы j в этом поддереве меньше k, и, следовательно, $A[j]<A[k]$. Дальнейший поиск можно поэтому ограничить только правым поддеревом.
3. $A[k]>x$ — из аналогичных соображений дальнейший поиск следует вести только в левом поддереве.



(11.7)

Эти соображения приводят к следующей программе:

```
var   i, j, k:integer; q:Boolean;                                    (11.8)
      A:array [1 .. n] of T;
begin {присваивание значений массиву A}
      i := 1; j := n; q := false;
      repeat k := (i+j) div 2;
         if A[k]=x then q: = true else
         if A[k]<x then i := k+1 else j := k-1
      until q ∨ (i > j)
end.
```

Стратегия поиска, которая лежит в основе программы (11.8), называется *бинарным поиском*. Заметим, что число требуемых сравнений в среднем значительно меньше, чем при линейном просмотре, а точнее говоря, не более чем $\log_2 n$ вместо n в программе (11.6). Математическое ожидание будет гораздо меньше чем $\log_2 n$. Конечно, бинарный поиск можно использовать только тогда, когда компоненты массива упорядочены (См. также (11.32).)

Пример: Вычисление скалярного произведения

Даны числа $x_1, \ldots, x_n$ и $y_1, \ldots, y_n$. Вычислить скалярное произведение

$$s = \sum_{i=1}^{n} x_i * y_i \qquad (11.9)$$

Представив эту сумму в виде рекуррентных соотношений

$$s_i = s_{i-1} + x_i * y_i, \qquad s_0 = 0 \tag{11.10}$$

мы получим следующую программу:

```
var   s: real; i: integer;
      x, y: array [1 .. n] of real;
begin {присваивание начальных значений x и y}                (11.11)
      s := 0; i := 0;
      repeat { s = Σ_{j=1}^{i} x[j] * y[j] }
          i := i + 1; s := s + x[i] * y[i]
      until i = n
end.
```

Как и в (11.6), элементы массива просматриваются строго последовательно, но отличие состоит в том, что в данной программе всегда просматриваются *все* компоненты. Однако *порядок* просмотра несуществен — можно даже выполнять все n умножений одновременно.

Этот случай при работе с массивами встречается так часто, что оправдано введение специальной нотации. Нам потребуется еще одна циклическая инструкция, аналогичная инструкциям **while** и **repeat**. Если S — повторяемая инструкция, V — скалярная переменная (называемая *параметром цикла*), а a и b — выражения того же типа, что и V, то инструкция

$$\textbf{for } V := a \textbf{ to } b \textbf{ do } S \tag{11.12}$$

означает, что две инструкции

$$V := x;\ S \tag{11.13}$$

выполняются для каждого значения x в интервале от a до b. В полном соответствии с традицией, которая сложилась в наиболее распространенных языках программирования, повторные действия выполняются последовательно для значений x, выбираемых в возрастающем порядке в интервале от a до b. Инструкцию (11.12) можно тогда рассматривать как эквивалентную последовательности инструкций

$$\textbf{begin } V := v_1;\ S;\ V := v_2;\ S;\ \ldots;\ V := v_n;\ S \textbf{ end} \tag{11.14}$$

где $v_1 = a$, $v_n = b$ и $v_i = succ(v_{i-1})$ для $i = 2, \ldots, n$. (Из всего этого следует, что функция получения следующего элемента (*succ*) над типом переменной V должна быть определена, и поэтому тип V не может быть *real*.) Инструкцию (11.4) можно представить в виде

следующей схемы программы:

```
if a ⩽ b then
begin V := a; S;
      while V < b do
          begin V := succ(V); S
          end
end
```
(11.15)

Из этой схемы в частности видно, что в инструкции **for** не предусмотрено никаких действий, если $a>b$. Сложность схемы для $a\leqslant b$ свидетельствует о сложности соответствующих правил верификации, которые мы приведем в интересах полноты изложения. Пусть P и $Q(V)$ — некоторые произвольные условия. Тогда следующим образом можно записать посылки

(a) $\{(V=a)\wedge P\}\ S\ \{Q(a)\}$
(b) $\{Q(pred(x))\}\ S\{Q(x)\}$ для всех $a<x\leqslant b$ (11.16.1)

и следствия

(a) $\{(a\leqslant b)\wedge P\}$ **for** $V:=a$ **to** b **do** $S\{Q(b)\}$
(b) $\{(a>b)\wedge P\}$ **for** $V:=a$ **to** b **do** $S\{P\}$ (11.16.2)

Применение этого индуктивного правила верификации можно продемонстрировать, если переписать программу (11.11) в виде

```
begin s := 0;
    for i := 1 to n do s := s + x[i] * y[i]
end
```
(11.17)

Подставив в (11.16.1) $s=0$ вместо P и $s=\sum_{j=1}^{i} x_j*y_j$ вместо $Q(i)$, получим две посылки:

(a) $\{(i=1)\wedge(s=0)\}\ s := s+x[i]*y[i]\left\{s=\sum_{j=1}^{1} x_j*y_j\right\}$ (11.18.1)

(b) $\left\{s=\sum_{j=1}^{i-1} x_j*y_j\right\}\ s := s+x[i]*y[i]\left\{s=\sum_{j=1}^{i} x_j*y_j\right\}$

Легко проверить, что условие (b) выполняется для всех $i=2, \ldots, n$. Теперь с помощью той же подстановки в (11.16.2) получим следствие

$$\{s=0\}\ \textbf{for}\ i := 1\ \textbf{to}\ n\ \textbf{do}\ s := s+x[i]*y[i]\left\{s=\sum_{j=1}^{n} x_j*y_j\right\} \tag{11.18.2}$$

Очевидно, условие $Q(V)$ соответствует инварианту в правилах верификации для инструкций **while** и **repeat** в (7.13) и (7.14). Однако необходима явная идентификация параметра цикла, так как присваивание этой переменной в заголовке инструкции **for** специфицируется неявно. В заключение следует отметить существенное преимущество инструкции **for**: не нужно доказывать, что цикл завершится. В самом деле, доказательство непосредственно следует из того факта, что множество значений $a, \ldots, b$ конечно.

Разумеется, инструкцию **for** можно использовать не только в тех программах, в которых обрабатываются массивы. В следующем правиле суммируются случаи, когда удобно применять инструкцию **for**.

Рекомендуется использовать инструкцию **for**, если число повторений *заранее* известно. Если число повторений становится известно лишь в процессе выполнения цикла, то рекомендуется пользоваться инструкциями **while** и **repeat**.

Три следующих примера демонстрируют использование массивов и инструкции **for**.

Пример: Поиск максимального значения $x[j]$

Найти индекс j, такой, что $x_j = \max(x_m, \ldots, x_n)$.

```
var j, k: m .. n;
    x: array [m .. n] of T;
begin j := m;                                    (11.19)
    for k := m+1 to n do
        if x[k] > x[j] then j := k
end
```

Условием $Q(k)$, используемым для верификации, является

$$x[j] \geqslant x[i] \text{ для всех } i = m, \ldots, k \tag{11.20}$$

Пример: Сортировка массива

Элементы массива переставляются таким образом, чтобы их значения были расположены в убывающем порядке. Чтобы выполнить эти перестановки

(a) с помощью программы (11.19) определяется максимальный элемент x_j,
(b) меняются местами x_j и x_1,
(c) шаги (a) и (b) повторяются соответственно для множеств $x_2, \ldots, x_n$, $x_3, \ldots, x_n$ и т. д., до тех пор пока в множестве останется только x_n.

Эти действия можно сформулировать в виде следующей инструкции:

```
for   h := 1 to n—1 do
begin {Q (h—1), если h > 1}
      1: Найти наибольший элемент x_j = max (x_h, ..., x_n);
      2: Поменять местами x_h и x_j                          (11.21)
         {Q (h)}
end.
```

Инструкция 1 подробно описана в (11.19), а инструкцию 2 можно представить в виде трех последовательных присваиваний с использованием вспомогательной переменной u,

$$u := x[h];\ x[h] := x[j];\ x[j] := u \qquad (11.22)$$

Для верификации программы используются следующие условия:

$$P: \text{пусто (т. е. } true\text{)}$$
$$Q(h): x_1 \geqslant x_2 \geqslant \ldots \geqslant x_h \geqslant x_i \text{ для всех } i > h \qquad (11.23)$$

Делая соответствующие подстановки в (11.21), мы получаем следующую программу сортировки:

```
    var h, j, k: 1 .. n;
        x: array [1 .. n] of T; u: T;
begin ...
        for h := 1 to n—1 do
        begin j := h;
          for k := h+1 to n do
              if x[k] > x[j] then j := k;                  (11.24)
          u := x[h]; x[h] := x[j]; x[j] := u
        end
end.
```

Эта программа содержит инструкцию **for**, в которую вложена другая инструкция **for**. Инструкция, начинающаяся с «**if** $x[k] > x[j]$ **then** . . .», выполняется

$$(n-1)+(n-2)+\ldots+2+1=\frac{n}{2}(n-1) \qquad (11.25)$$

раз. Затраты на выполнение сортировки с помощью этого примитивного метода, грубо говоря, пропорциональны квадрату числа сортируемых элементов. Поэтому для большого числа n рекомендуется использовать более совершенные методы сортировки.

Элементы массива могут быть не только скалярами, но также и структурными компонентами. Если они являются снова массивами, то первоначальный массив A называется *многомерным*. Если элементы компонентных массивов являются скалярами, то массив A называется *матрицей*. Многомерный массив описывается в полном

соответствии с (11.1). Например, в описании

var M: **array** [$a..b$] **of array** [$c..d$] **of** T (11.26)

объявляется, что M состоит из $b-a+1$ компонент (которые часто называют строками матрицы) с индексами $a, \ldots, b$, каждая из которых является массивом, содержащим $d-c+1$ компонент типа T с индексами $c, \ldots, d$. Для обозначения i-й компоненты (строки матрицы) M используется традиционная запись

$$M[i] \quad a \leqslant i \leqslant b \tag{11.27.1}$$

а j-я компонента типа T этой строки обозначается как

$$M[i][j] \quad a \leqslant i \leqslant b, \quad c \leqslant j \leqslant d \tag{11.27.2}$$

Обычно удобно пользоваться следующими сокращениями, каждое из которых эквивалентно соответственно записям (11.26) и (11.27.2):

var M: **array** [a .. b, c .. d] **of** T (11.28)
$M[i, j]$

Пример: Умножение матриц

Заданы две вещественнозначные матрицы $A(m \times p)$ и $B(p \times n)$. Вычислить произведение, т. е. матрицу $C(m \times n)$, элементы которой определяются как

$$C_{ij} = \sum_{k=1}^{p} A_{ik} * B_{kj} \tag{11.29}$$

для $i=1, \ldots, m$ и $j=1, \ldots, n$.

Непосредственно используя формулу (11.29), напишем следующую программу:

```
var  i: 1..m; j: 1..n; k: 1..p; s: real
     A: array [1..m, 1..p] of real;
     B: array [1..p, 1..n] of real;
     C: array [1..m, 1..n] of real;
begin  {присваивание начальных значений массивам A и B}
       for i := 1 to m do
           for j := 1 to n do
               begin s:= 0;                                  (11.30)
                 for k := 1 to p do
                 s := s+A[i, k]*B[k, j];
                 C[i, j] := s
               end
end.
```

В этой программе используются многократные вложения циклических инструкций друг в друга. Поскольку такие программы неизбежно связаны с относительно большим количеством вычисле-

ний, необходим более тщательный анализ их эффективности. Очевидно, что цикл по i выполняется m раз, по j — $m*n$ раз, а цикл по k выполняется $m*n*p$ раз. Если предположить, что m, n и p — большие числа ($\gg 1$), то затраты на выполнение инструкции

$$s := s+A[i, k]*B[k, j]$$

составят самую значительную часть всех вычислений. В связи с этим следует помнить простое, но очень важное правило — «самая внутренняя» циклическая инструкция должна формулироваться с особой тщательностью с тем, чтобы по возможности минимизировать затраты на вычисления и повысить эффективность программы. Затраты на умножение матриц растут как *третья* степень числа n, если предположить, что $m=n=p$. (См. упражнение (11.8).)

УПРАЖНЕНИЯ

11.1 Дана матрица A

$$A=\begin{pmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \\ 1 & 2 & 1 \end{pmatrix}$$

(a) Выполните инструкцию

```
for i := 1 to 3 do
for j := 1 to 3 do C[i, j] := A[A[i,j], A[j,i]]        (11.31)
```

Каким будет результирующее значение C?

(b) Существен ли порядок, в котором выбираются значения индексов i и j?

(c) Вместо C в (11.31) подставьте A. Каким будет результирующее значение A?

(d) Повторите п. (c), заменив прямой порядок перебора индексов i и j на обратный:

(3,3), (3,2), ..., (1,2), (1,1)

Сравните полученное значение A с результатом вычислений в п. (c).

11.2 Проверьте следующий вариант программы бинарного поиска:

```
i := m; j := n;
repeat k := (i+j) div 2;
    if A[k] ≤ x then i := k+1;                          (11.32)
    if A[k] ≥ x then j := k-1;
until i > j
```

Сопоставьте количество сравнений, выполняемых в этой программе, с количеством сравнений в (11.8). Обратите внимание на то, что в (11.32) условие окончания проще, чем в (11.8).

11.3 Комплексная матрица Z представляется парой $\langle X, Y\rangle$ вещественных матриц так, что $Z=X+iY$. Напишите программу вычисления произведения двух комплексных матриц $\langle A, B\rangle$ и $\langle C, D\rangle$, т. е.

$$X+iY=(A+iB)*(C+iD) \tag{11.33}$$

Указание:

$$(A+iB)*(C+iD)=(AC-BD)+i(AD+BC) \tag{11.34}$$

Получите три матрицы

$$R=A*D,\quad S=B*C,\quad T=(A+B)*(C-D)$$

а затем матрицы

$$X=T+R-S \quad \text{и} \quad Y=R+S$$

Определите число необходимых сложений и умножений (в зависимости от размера матрицы n) и сравните с числом сложений и умножений, если прямо пользоваться формулой (11.33).

11.4 Многочлен

$$P_n(x)=a_0x^n+a_1x^{n-1}+\ldots+a_{n-1}x+a_n \tag{11.35}$$

представляется массивом коэффициентов a. Напишите программу вычисления $P_n(x)$ для заданного x. *Указание*: Используйте схему Горнера, а именно

$$P_n(x)=(\ldots(a_0x+a_1)*x+\ldots+a_{n-1})*x+a_n \tag{11.36}$$

11.5 Напишите программу нахождения наибольшего и наименьшего значения в массиве A, содержащем n чисел:

var A : **array** [1..n] **of** *integer*

Указание: Можно сделать так, что потребуется меньше, чем $(3/2)n$ сравнений.

11.6 Задан массив

var M: **array** [1..n,1..n] **of** *integer*

Напишите программу, присваивающую элементам этого массива натуральные числа 1, 2, ..., n^2, так, чтобы получился *магический квадрат*, т. е.

$$\sum_{k=1}^{n} M[i,\ k]=\sum_{k=1}^{n} M[k,\ i]=C \qquad i=1,\ \ldots,\ n$$

и

$$\sum_{k=1}^{n} M[k,\ k]=\sum_{k=1}^{n} M[k,\ n-k+1]=C$$

где $C=(n/2)(n^2+1)$. Предполагается, что n нечетно. *Указание*: Присваивайте элементам массива M последовательно числа 1, ..., n^2. Начните с $i=(n+1)/2$ и $j=n$, чтобы значение 1 было присвоено элементу $M[(n+1)/2,\ n]$. Затем в течение $n-1$ шагов на каждом шаге увеличивайте i и j на 1 (по модулю n); на каждом n-м шаге i оставляйте без изменения, а j уменьшайте на 1.

11.7 Ниже приводится программа, которая была написана для суммирования первых $n+1$ членов разложения синуса в ряд [см. (9.29)] для аргументов x, удовлетворяющих неравенству $0\leqslant x\leqslant\pi/4$. (Предполагается, что результатом будет значение переменной s.)

```
var i: integer; h, s: real;
    t: array [0..n] of real;
begin t[0] := x; h := sqr(x); s := x;                    (11.37)
   for i := 1 to n do
      t[i] := -t[i-1]*h/(2*i*(2*i+1));
   for i := 1 to n do s := s+t[i]
end.
```

Правильно ли написана программа? Какие критические замечания можно высказать по ее поводу?

11.8 Даны две вещественнозначные матрицы X и Y, имеющие $2n$ строк и столбцов каждая. Напишите программу, вычисляющую произведение матриц $Z=X*Y$, используя следующее соотношение (оно принадлежит Винограду) для вычисления скалярных произведений [см. (11.29)]:

$$\sum_{k=1}^{2n} x_k * y_k = \sum_{k=1}^{n} (x_{2k}+y_{2k-1}) * (x_{2k-1}+y_{2k}) \underbrace{-\sum_{k=1}^{n} x_{2k} * x_{2k-1}}_{\bar{x}} \underbrace{-\sum_{k=1}^{n} y_{2k} * y_{2k-1}}_{\bar{y}} \tag{11.38}$$

Указание: В задаче требуется вычислить $4n^2$ скалярных произведений

$$\sum_{k=1}^{2n} x_{ik} * y_{kj}$$

Если воспользоваться (11.38), то потребуется только $2n$ значений $\bar{x}$ и $\bar{y}$. Обычный метод умножения матриц (11.30) требует $8n^3$ сложений и умножений. Определите число операций (как функцию от n), которые требуются для выполнения вашей программы.

12. ПОДПРОГРАММЫ, ПРОЦЕДУРЫ И ФУНКЦИИ

12.1. Основные понятия и терминология

Часто некоторую последовательность инструкций требуется повторить в нескольких местах программы. Чтобы программисту не приходилось тратить время и усилия на копирование этих инструкций, в большинстве языков программирования предусматриваются средства для организации *подпрограмм*. Таким образом, программист получает возможность присвоить последовательности инструкций произвольное имя и использовать это имя в качестве сокращенной записи в тех местах, где встречается соответствующая последовательность инструкций. Следуя терминологии Алгола, такую именованную последовательность инструкций будем называть *процедурой*. Если процедура дает одно результирующее значение и, следовательно, может использоваться в выражениях, то такая процедура называется *функцией*. Определение сокращенной записи называется *описанием процедуры* или *описанием функции*. Использование этого сокращения в программе называется *оператором процедуры* или *вызовом процедуры*. Функция, если она встречается в выражении, называется *указателем функции* или *обращением к функции*.

Конкретная нотация, используемая в описаниях и операторах процедур, приводится в синтаксических диаграммах приложения А и примерах этой главы.

Пример: Описание и оператор процедуры

Для последовательности инструкций

$$t := r \bmod q;\ r := q;\quad q := t \tag{12.1}$$

можно ввести сокращение, описав процедуру следующим образом:

procedure P;
begin $t := r \bmod q;\ r := q;\ q := t$ **end** (12.2)

И теперь всякий раз, когда в программе встречается эта последовательность инструкций, ее можно заменить оператором процедуры

$$P \tag{12.3}$$

Описание процедуры состоит из двух частей: *заголовка процедуры* и *тела процедуры*. Заголовок (первая строка в (12:2)) содержит идентификатор процедуры. Тело (вторая строка в (12.2)) состоит из одной или нескольких инструкций, для которых вводится сокращение.

Вряд ли стоило подробно говорить о столь простой форме записи, если бы за ней не скрывались важные и основополагающие понятия. В действительности процедура является одним из тех немногих фундаментальных инструментов в искусстве программирования, которые оказывают решающее влияние на стиль и качество работы программиста. Процедура — это способ сокращения текста и, что более важно, средство разложения программы на логически связанные, замкнутые компоненты, определяющие ее структуру. Разложение на части существенно для понимания программы, особенно если программа сложна и из-за большой длины текста трудно-обозрима. Разложение на подпрограммы необходимо как для документирования, так и для верификации программы. Поэтому желательно оформлять последовательность инструкций в виде процедуры, даже если процедура используется однократно и, следовательно, отсутствует мотив, связанный с сокращением текста программы. Дополнительную информацию о переменных (которые используются или изменяются в процедуре) или об условиях, которым должны удовлетворять аргументы, удобно задавать в заголовке процедуры.

О полезности процедуры, в частности о ее роли при структуризации программы, неоспоримо свидетельствуют еще два понятия в программировании. Часто некоторые переменные или объекты (их обычно называют *вспомогательными переменными*), используемые внутри заданной последовательности инструкций, не имеют смысла за пределами этих инструкций. В программе существенно проще разобраться, если явно указаны области действия таких объектов. Процедура выступает как естественная текстовая единица, с помощью которой ограничивается *область существования* так называемых *локальных объектов*.

Очень часто некоторые последовательности инструкций, встречающиеся в разных местах программы, не идентичны, но очень близки по форме. Особенно важна ситуация, когда различие между отдельными вхождениями инструкций можно устранить систематической заменой идентификаторов и выражений. В этом случае инструкции, для которых определяется сокращенная запись, можно представить в более общем виде *схемой процедуры*. Объекты, которые придется заменять для каждого индивидуального вхождения, называются *параметрами процедуры*.

12.2. Локальность

Если объект (константа, переменная, процедура, функция или тип) имеет смысл только в пределах некоторой части программы, то этот объект называется *локальным*. В таком случае этой части программы можно присвоить имя (т. е. оформить ее как процедуру). Локальные объекты процедуры описываются в ее заголовке. Так

как сами процедуры можно определять локально, то описания процедур могут быть вложенными.

Пример: Описание процедуры с описанием локальной переменной

```
procedure P;
    var t: integer;                                    (12.4)
begin t := r mod q; r := q; q:=t end
```

В теле процедуры используются объекты двух сортов: локальные (в примере — *t*) и нелокальные объекты. Последние определены в контексте, являющемся *средой* для описания процедуры. Если они определены в главной программе, то такие объекты называются *глобальными*; если же они определены в самом языке (т. е. в контексте, в который «погружается» программа), то они называются *стандартными* объектами. Областью существования локального объекта является весь текст процедуры. Это означает, что после окончания процесса, описанного процедурой, пространство в памяти, занятое локальными переменными, становится снова свободным и его можно использовать для других переменных. Очевидно, при повторном вызове той же самой процедуры значения ее локальных переменных снова не определены, точно так же как они не были определены при первом вызове процедуры.

При идентификации локальных объектов существенно то, что мы можем выбирать их имена вне зависимости от среды. Но может случиться, что идентификатор (например, *x*), который был выбран для некоторой локальной переменной в процедуре *P*, совпадает с именем объекта, определенного вне *P*. Конечно, эта ситуация имеет смысл только тогда, когда нелокальный *x* не используется в *P*. Поэтому мы условимся, что при совпадении имен, *x* внутри *P* будет означать локальную переменную, а *x* вне *P* рассматривается как нелокальный объект.

Пример: Процедура, в которой идентификатор локальной переменной (d) совпадает с именем глобального объекта

```
var a, b, d, e: integer; {глобальные переменные}
procedure Multiply; {глобальная процедура}
   var c, d: integer; {локальные переменные}
begin {e := a*b, см. (7.18)}                           (12.5)
    c := a; d := b; e := 0;
    while d≠0 do
    begin if odd(d) then e := e+c;
          c := 2*c; d := d div 2
    end
end;
begin {главная программа} a := 5; b := 7; d := 10;
      Multiply {a=5, b=7, d=10, e=35}
end.
```

Главную программу удобно рассматривать как процедуру без имени. Ее средой является операционная система вычислительной машины, где предварительно определены все стандартные объекты. Таким образом, ясно, почему идентификаторы можно выбирать, не считаясь со стандартными именами. Если стандартный объект не используется в программе, то употребление (случайное или преднамеренное) его идентификатора в качестве имени локального объекта не вызовет каких-либо нежелательных эффектов.

12.3. Параметры процедуры

Если некоторая последовательность операций применяется к *различным* операндам в разных частях программы, то такая последовательность оформляется как *процедура*, а ее операнды становятся *параметрами*. Идентификаторы, введенные в заголовке процедуры для обозначения операндов, называются *формальными параметрами*. Они используются только в теле процедуры и локальны по отношению к ней. Объекты, подставляемые вместо формальных параметров, называются *фактическими параметрами*. Они задаются в каждом операторе процедуры или указателе функции. *Тип* фактического параметра определяется типом формального параметра, который специфицируется в заголовке процедуры. Помимо спецификации типа параметра необходимо также указать желаемый *способ* подстановки, поскольку вместо формального параметра можно подставить текущее значение либо имя фактической переменной или выражение. Принято различать три способа подстановки параметров.

1. Фактический параметр вычисляется и полученное значение подставляется вместо соответствующего формального параметра. Этот способ называется *подстановкой значения* и имеет наибольшее распространение.

2. Фактический параметр есть переменная. Возможно, потребуется вычисление индексов. Определенная таким образом переменная заменяет соответствующий формальный параметр. Этот способ называется *подстановкой переменной (ссылки)* и используется в тех случаях, когда параметр вычисляется в процедуре и является ее результатом.

3. Фактический параметр подставляется буквально без каких-либо вычислений. Это называется *подстановкой по имени*. На практике этот способ используется редко.

Пример:

На примере программы (12.6) демонстрируются три способа подстановки параметров. Мы рассмотрим результаты выполнения опе-

ратора процедуры $P(a[i])$.

```
var i: integer;
    a: array [1..2] of integer;
procedure P(x: integer);                    (12.6)
    begin i := i+1; x := x+2
    end;
begin {Главная программа}
    a[1] := 10; a[2] := 20; i := 1;
    P(a[i])
end.
```

Случай 1: *Подстановка значения*
Переменная x имеет начальное значение 10.
Конечное значение переменной a=(10, 20)

Случай 2: *Подстановка переменной*
$x \equiv a[1]$
Инструкция $x := x+2$ теперь означает $a[i] := a[1]+2$.
Конечное значение переменной a=(12, 20)

Случай 3: *Подстановка по имени*
$x \equiv a[i]$
Инструкция $x := x+2$ теперь означает $a[i] := a[i]+2$.
Конечное значение переменной a=(10, 22)

Для того чтобы отличать разные способы подстановки, мы сформулируем следующие правила, касающиеся нотации:

1. *Подстановка значения* одна из самых распространенных подстановок, и поэтому именно она подразумевается по умолчанию, т. е. в том случае, когда отсутствует явная спецификация.
2. *Подстановка переменной* указывается с помощью символа **var** перед одним или несколькими формальными параметрами.
3. Мы воздержимся от введения обозначений для *подстановки по имени*, поскольку аналогичные ситуации рассматриваются в разд. 12.4.

В соответствии с этими правилами мы можем переписать программу (12.5) в виде процедуры с тремя параметрами, как показано в (12.7).

Пример: Процедуры с параметрами

```
var a, b, c, d, e, f: integer;
procedure Multiply (x, y: integer; var z: integer);
   {x, y, z—формальные параметры}
begin z := 0;                               (12.7)
   while x≠0 do
   begin if odd(x) then z := z+y;
         y := 2*y; x := x div 2
   end
end;
```

```
begin {Главная программа}
   a := 5; b := 7; d := 11; e := 13;
   Multiply (a, b, c); Multiply (d—b, e—a, f)
   {a=5, b=7, c=35, d=11, e=13, f=32}
end.
```

В программе (12.7) показано, что в случае подстановки значения формальный параметр обозначает локальную переменную, которой вначале присваивается результат вычисления соответствующего фактического параметра. И после этого начального присваивания далее уже не существует какой-либо связи между фактическим и формальным объектами. Это позволяет нам сформулировать два общих правила, полезных при выборе подстановки.

1. Если параметр является аргументом, а не результатом процедуры (функции), то (как правило) следует предпочесть *подстановку значения*.
2. Если параметр обозначает результат процедуры, то необходима *подстановка переменной*.

Метод подстановки переменной таит в себе некоторую опасность, поскольку доступ к *одной и той же* переменной может осуществляться с помощью более чем одной идентификации. Опасность неизмеримо возрастает, если мы имеем дело со структурными переменными, например, с массивами. В этих случаях необходимо, чтобы программист строго соблюдал дисциплину программирования и, в частности, следующее важное правило: *изменяемые параметры нельзя связывать с какими-либо другими параметрами, т. е. никакой параметр не должен обозначать другой параметр или его компоненту*. Опасности, которые могут проявиться при нарушении этого правила, показаны на примере простой процедуры умножения матриц.

Пример: Неоднозначность при использовании зависимых изменяемых параметров

```
type matrix = array [1..2, 1..2] of integer;
procedure mult (var x, y, z: matrix);                    (12.8)
begin z[1, 1] := x[1, 1]*y[1, 1]+x[1, 2]*y[2, 1];
      z[1, 2] := x[1, 1]*y[1, 2]+x[1, 2]*y[2, 2];
      z[2, 1] := x[2, 1]*y[1, 1]+x[2, 2]*y[2, 1];
      z[2, 2] := x[2, 1]*y[1, 2]+x[2, 2]*y[2, 2];
end
```

Даны матрицы

$$A=\begin{pmatrix} 2 & 1 \\ -1 & 3 \end{pmatrix} \quad \text{и} \quad B=\begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

Мы исследуем действие следующих инструкций:

1. *mult* (*A*, *B*, *C*) приводит к $C=\begin{pmatrix} 7 & 0 \\ 0 & 7 \end{pmatrix}$
2. *mult* (*A*, *B*, *A*) приводит к $A=\begin{pmatrix} 7 & 5 \\ 0 & 6 \end{pmatrix}$
3. *mult* (*A*, *B*, *B*) приводит к $B=\begin{pmatrix} 7 & 0 \\ 4 & 6 \end{pmatrix}$

Заметим, что правильные результаты (см. инструкцию 1) можно получить во всех трех случаях, если x и y специфицированы как параметры-значения (см. также упражнение 11.1).

12.4. Использование имени процедуры или функции в качестве фактического параметра

Процедура или функция F используется в качестве параметра процедуры или функции G, если F должна вычисляться во время выполнения G и если F представляет собой различные процедуры или функции в различных обращениях к G. Хорошо известными примерами являются алгоритмы вычисления интеграла G функции F.

Пример: Вычисление интеграла по формуле Симпсона
Для аппроксимации интеграла

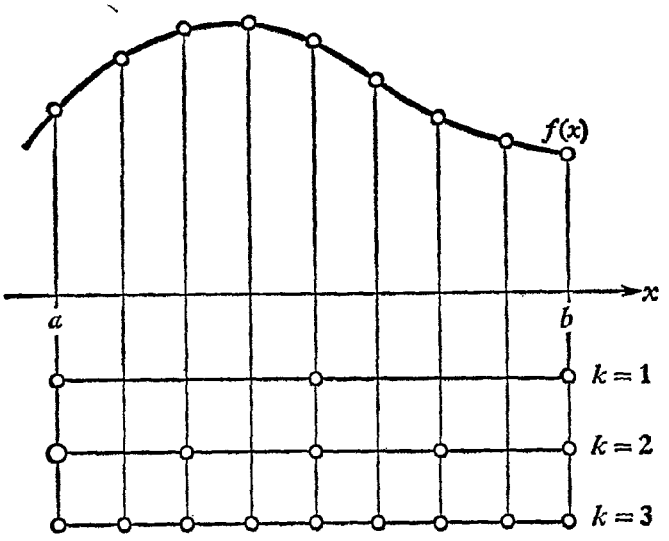
$$s=\int_a^b f(x)\,dx \tag{12.9}$$

используется сумма конечного числа узловых значений f_i.

$$s_k=\frac{h}{3}(f_0+4f_1+2f_2+4f_3+2f_4+\ldots+4f_{n-3}+2f_{n-2}+4f_{n-1}+f_n) \tag{12.10}$$

где $f_i=f(a+i*h)$, $h=(b-a)/n$ и $n=2^k$. Число узловых точек равно $n+1$, а h — расстояние между двумя соседними узловыми точками. Значение интеграла s аппроксимируется последовательностью s_1, s_2, s_3, ..., которая сходится, если функция ведет себя достаточно хорошо (гладкая) и если арифметические операции выполняются точно.

На каждом шаге число узловых точек удваивается. Конечно, хорошая программа не будет вычислять функцию $f(x)$ 2^k раз на каждом k-м шаге, поскольку можно использовать значения f_i,



(12.11)

полученные на предыдущих шагах. В сумме s_k три члена

$$s_k = s_k^{(1)} + s_k^{(2)} + s_k^{(4)} \tag{12.12}$$

которые обозначают суммы в узловых точках с весами 1, 2 и 4. Их можно определить с помощью рекуррентных соотношений (12.13) для $k>1$ и начальных значений (12.14).

$$\begin{aligned} s_k^{(1)} &= \frac{1}{2} s_{k-1}^{(1)} \\ s_k^{(2)} &= \frac{1}{2} s_{k-1}^{(2)} + \frac{1}{4} s_{k-1}^{(4)} \\ s_k^{(4)} &= \frac{4h}{3} (f(a+h) + f(a+3h) + \ldots + f(a+(n-1)h) \end{aligned} \tag{12.13}$$

$$\begin{aligned} s_1^{(1)} &= \frac{h}{3} (f(a) + f(b)) \\ s_1^{(2)} &= 0 \\ s_1^{(4)} &= \frac{4h}{3} f\left(\frac{a+b}{2}\right) \end{aligned} \tag{12.14}$$

После соответствующей подстановки в схему программы (9.23) эти соотношения приводят к программе интегрирования (12.15), в которой функция f играет роль параметра.

```
function Simpson (a, b: real; function f: real): real;
    var i, n: integer;
        s, ss, s1, s2, s4, h: real;
{f(x)—вещественная функция с одним параметром
типа real. Функция должна быть определена
в интервале a≤x≤b}
begin n := 2; h := (b—a)*0.5;
    s1 := h*(f(a)+f(b)); s2 := 0;
    s4 := 4*h*f(a+h); s := s1+s2+s4;                    (12.15)
    repeat ss := s; n := 2*n; h := h/2;
        s1 := 0.5*s1; s2 := 0.5*s2+0.25*s4;
        s4 := 0; i := 1;
        repeat s4 := s4+f(a+i*h); i := i+2
        until i > n;
        s4 := 4*h*s4; s := s1+s2+s4
    until abs(s—ss) < ε;
    Simpson := s/3
end
```

Теперь функцию *Simpson* можно использовать в качестве операнда в любом вещественном выражении. Например, инструкция

$$u := Simpson\ (0,\ \pi/2,\ sin) \tag{12.16}$$

обозначает присваивание

$$u = \int_0^{\pi/2} \sin(x)\,dx$$

Однако третьим фактическим параметром в обращении к функции *Simpson* может быть только имя функции. В большинстве языков программирования не разрешается использовать в этом качестве выражение. (Исключение составляет Алгол-60, в котором возможна так называемая подстановка по имени.) Например, для вычисления интеграла

$$u = \int_0^{\pi/2} \frac{dx}{(a^2\cos^2 x + b^2\sin^2 x)^{\frac{1}{2}}} \tag{12.17}$$

с помощью функции *Simpson* необходимо явно определить еще одну функцию *F*.

```
function F(x: real): real;                              (12.18)
begin F := 1/sqrt(sqr(a*cos(x))+sqr(b*sin(x))) end
```

Только в этом случае можно выразить (12.17) с помощью инструкции

$$u := Simpson(0,\ \pi/2,\ F) \qquad (12.19)$$

УПРАЖНЕНИЯ

12.1 Сформулируйте программы (7.18), (7.22), (9.17), (9.28), (10.21), (11.24) и (11.30) в виде процедур или функций с соответствующим образом выбранными параметрами.

12.2 Рассмотрите следующее описание функции:

```
function f (x, y: real): real;
begin if x ≥ y then f := (x+y)/2 else                    (12.20)
    f := f (f (x+2, y—1), f (x+1, y—2))
end
```

Каким будет значение f (1,10)?
Каким более простым способом можно представить и вычислить $f(a, b)$?

12.3 Выполните следующие три программы и определите значения фактических параметров в инструкциях *write*:

a)
```
var a, b, c: integer;
procedure P (x, y: integer; var z: integer);
begin z := x+y+z; write (x, y, z)
end;                                                     (12.21)
begin a := 5; b := 8; c := 3;
  P (a, b, c); P (7, a+b+c, a); P (a*b, a div b, c)
end.
```

b)
```
var i, j, k: integer;
procedure P (var i: integer);
begin i := i+1; write (i, j, k)
end {P};
procedure Q (h: integer; var j: integer);                (12.22)
    var i: integer;
    procedure R;
    begin i := i+1
    end {R};
  begin i := j;
    if h=0 then P (j) else if h=1 then P (i) else R;
    write (i, j, k)
  end {Q};
begin i := 0; j := 1; k := 2; Q (0, k); Q (1, i); Q (q, j)
end.
```

c)
```
procedure P (procedure R; b: Boolean);
    var x: integer;
    procedure Q;
    begin x := x+1
    end {Q};                                             (12.23)
  begin x := 0; if b then P (Q, false) else R;
    write (x)
  end {P};
begin P (P, true)
end.
```

12.4 Как показал *Гаусс*, эллиптический интеграл (см. 12.17)

$$I=\frac{2}{\pi}\int_0^{\pi/2}\frac{dx}{(a^2\cos^2 x+b^2\sin^2 x)^{\frac{1}{2}}}$$

равен пределу любой из двух сходящихся последовательностей

$$s_0,\ s_1,\ s_2,\ \ldots \text{ или } t_0,\ t_1,\ t_2,\ \ldots$$

которые определяются следующими рекуррентными соотношениями (для $i>0$):

$$s_i=(s_{i-1}+t_{i-1})/2$$

$$t_i=\sqrt{s_{i-1}*t_{i-1}}$$

и начальными значениями $s_0=a$, $t_0=b$. Этот способ вычисления двух последовательностей называется методом арифметическо-геометрического среднего. Дайте описание соответствующей функции.

12.5 Метод интегрирования, предложенный *Ромбергом*, основан на аппроксимации интеграла

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

последовательностью

$$t_{0,0},\ t_{1,0},\ t_{2,0},\ \ldots$$

которая сходится, если функция f достаточно гладкая. Члены последовательности определяются рекуррентным соотношением

$$t_{m,k}=\frac{1}{4^m-1}(4^m*t_{m-1,k+1}-t_{m-1,k})$$

при $m>0$ и начальными значениями

$$t_{0,k}=\frac{b-a}{n}\left(\frac{1}{2}f_0+f_1+\cdots+f_{n-1}+\frac{1}{2}f_n\right)$$

где $n=2^k$ и $f_i=f(a+i*(b-a)/n)$. Дайте описание функции с параметрами a, b и f, аппроксимирующей интеграл с заданной относительной точностью ε. *Указание*: В каждой узловой точке программа должна вычислять функцию f только один раз. На каждом шаге число узловых точек удваивается. Используйте массив T, такой, что после k-го шага

$$T[i]=t_{k-i,i}\quad i=0,\ \ldots,\ k$$

В случае плохой сходимости итерации следует прекратить не более чем через p шагов, т. е. $k=0,\ \ldots,\ p$.

12.6 Назовем значение x_0 «корнем» вещественной функции $f(x)$, если

$$(f(x_0-\varepsilon)<0)=(f(x_0+\varepsilon)>0)$$

при любом сколь угодно малом ε. Напишите программу-функцию с параметрами a, b и f, определяющую корень $f(x)$ в интервале $a\leqslant x\leqslant b$ при условии, что выполняется отношение

$$(f(a)<0)=(f(b)>0)$$

Указание: Используйте метод многократного деления пополам интервала, содержащего корень. Обратите внимание на сходство этого метода, называемого *делением пополам* [1], с бинарным поиском (см. 11.32). Определите, сколько раз требуется вычислить функцию f, если заданы a, b и ε?

[1] В оригинале *bisection*.— *Прим. ред.*

13. ПРЕОБРАЗОВАНИЕ ПРЕДСТАВЛЕНИЙ ЧИСЕЛ

Абстрактное понятие числа не зависит от его возможных представлений. Операции над числами можно точно определить некоторой совокупностью общих аксиом. Однако если эти операции выполняются над вполне определенными числами, то конкретное представление для них должно быть выбрано так, чтобы можно было узнать результат.

Желательно определить операции над числами с помощью общезначимых алгоритмов, не зависящих от конкретного представления. Тогда в любом процессоре, предназначенном для выполнения арифметических операций, можно выбрать представление, наиболее соответствующее его возможностям. В современных вычислительных машинах свобода выбора используется так, что предпочтение отдается представлению, в основу которого положен двоичный алфавит. Однако двоичное представление не удобно для человека, поскольку его с раннего детства учат пользоваться десятичной системой счисления. По этой причине устройства ввода-вывода в вычислительных машинах снабжены наборами литер, содержащими десятичные цифры, которым, разумеется, сопоставлены внутренние двоичные коды. При вводе и выводе числовых данных вычислительная машина переводит числа из внешнего, привычного для нас представления, во внутренний вид и наоборот. В общепринятом десятичном представлении

(a) используется позиционная запись цифр, расположенных по степеням так называемого основания B, а

(b) число 10 определяет количество различных цифр и, кроме того, является основанием системы счисления.

При обобщении алгоритмов преобразования представлений для систем с произвольным основанием $B\,(>1)$ не возникает каких-либо дополнительных трудностей. Однако мы ограничимся рассмотрением алгоритмов перевода для *позиционных систем счисления*. Таким образом, десятичная система будет частным случаем, когда $B=10$. Общее свойство всех позиционных систем состоит в том, что последовательность $\delta_n=d_1\ldots d_n$, состоящая из n цифр, обозначает число

$$\tilde{\delta}_n=\sum_{i=1}^{n}\tilde{d}_i * B^{n-i}=(\ldots(\tilde{d}_1 * B+\tilde{d}_2) * B+\cdots+\tilde{d}_{n-1}) * B+\tilde{d}_n \tag{13.1}$$

где $\tilde{d}_i$ есть числовое значение, представленное литерой (цифрой) d_i, а B — основание системы счисления. Любое число x ($0 \leqslant x < B^n$) однозначно представимо, если существует B различных цифр, которым соответствуют значения 0, 1, ..., B—1. Если записать первые i цифр представления числа в виде $\delta_i = d_1 \ldots d_i$, то из (13.1) и $\tilde{\delta}_0 = 0$ непосредственно следует рекуррентное соотношение

$$\tilde{\delta}_i = \tilde{\delta}_{i-1} * B + \tilde{d}_i \qquad 0 \leqslant \tilde{d}_i < B \qquad (13.2)$$

Из (13.2) также следует

$$\tilde{\delta}_{i-1} = \tilde{\delta}_i \textbf{ div } B \text{ и } \tilde{d}_i = \tilde{\delta}_i \textbf{ mod } B \text{ (см. также 7.23)} \qquad (13.3)$$

В последующих программах функция для вычисления $\tilde{d}$ будет обозначаться как *num* (*d*). Она определена над одиночными литерами (цифрами). Обратную функцию обозначим через *rep* (*x*);

$$rep\,(num\,(d)) = d \text{ и } num\,(rep\,(x)) = x \qquad (13.4)$$

Эти функции зависят от конкретного набора литер. Например, для набора литер ASCII их можно определить с помощью стандартных функций перевода *chr* и *ord*:

$$rep\,(x) \equiv chr\,(x + ord\,('0')) \text{ и } num\,(d) \equiv ord\,(d) - ord\,('0') \qquad (13.5)$$

Обратный алгоритм должен вычислять последовательность цифр $d_1 \ldots d_n = \delta_n$. Целью рассматриваемых программ является расширение элементарных функций перевода на случай, когда переводимые числа превышают B и когда имеется не одна цифра, а последовательность цифр.

13.1. Ввод и вывод неотрицательных целых чисел в позиционной форме

Программа нахождения числа, представленного последовательностью цифр в текстовом файле D, получается применением рекуррентного соотношения (13.2) к схеме программы (9.2) с использованием инструкции **repeat**, а не **while**. Программа оформлена как процедура (13.6), в которой предполагается, что при входе в процедуру значением буферной переменной $D\uparrow$ является первая цифра d_1. Кроме того, предполагается, что последовательность цифр оканчивается литерой, не являющейся цифрой, и поэтому чтение из файла D будет продолжаться до тех пор, пока эта литера не станет значением буферной переменной $D\uparrow$. Эта схема более всего соответствует той ситуации, когда читается файл ввода, в котором последовательности цифр включены в общий текст (например, в про-

грамму).

```
procedure read (var x: integer);
begin x := 0;
    repeat x := B*x+num(D↑); get(D)
    until ¬ digit(D↑)
end
```
(13.6)

Логическая функция *digit*(*d*), так же как и *num*(*d*), зависит от конкретного набора литер *char*. Для набора литер ASCII, например, эту функцию можно описать следующим образом:

$$digit(d) \equiv ('0' \leqslant d) \wedge (d \leqslant '9') \tag{13.7}$$

Обратный алгоритм нахождения цифр $d_1 \ldots d_n = \delta_n$, представляющих число δ_n в позиционной системе счисления с основанием B, получается применением соотношений (13.3) к схеме программы (9.2) опять-таки с использованием инструкции **repeat**, а не **while**. Если задано число δ_i, то последняя цифра в его представлении находится делением этого числа на B и применением элементарной функции перевода *rep*(*x*) к полученному остатку. Остальные цифры можно получить, применяя тот же алгоритм к частному от деления.

Если цифры $d_1, \ldots, d_n$ необходимо записать в файл вывода D, то мы обнаруживаем неудобства, поскольку наш алгоритм выдает цифры в обратном порядке (последняя цифра выдается первой). Очевидно, потребуется буферная память, которую целесообразнее всего представить в виде массива *d*. В результате мы получаем программу (13.8), которая также оформляется в виде процедуры

```
procedure write(x: integer);
var u: integer; i:0..n;
    d: array [1..n] of char;
begin {0 ⩽ x < B^n} u := x; i := n;
    repeat d[i] := rep(u mod B); u := u div B;
        i := i−1
    until i = 0;
    repeat {вывод цифр в обратном порядке}
        i := i+1; D↑ := d[i]; put(D)
    until i = n
end
```
(13.8)

В эту программу легко внести следующие два улучшения:

1. Чтобы подавить старшие нули, предшествующие первой значащей цифре, заменим условие окончания первого цикла $i=0$ на $u=0$. Заметим, что одна цифра появится в любом случае, если даже $x=0$.

2. Введем вспомогательную переменную v и заменим две инструкции, в которые входят довольно «дорогие» операции **div** и **mod**, на следующие инструкции

$$v := u \ \mathbf{div}\ B;\quad d[i] := rep(u - B * u);\ u := v$$

13.2. Вывод дробей в позиционной форме

В позиционной системе счисления вещественные числа представляются целой и дробной частями, отделенными друг от друга точкой. Поскольку в разд. 13.1 проблема преобразования натурального числа в последовательность цифр уже обсуждалась, мы можем ограничиться рассмотрением алгоритма перевода чистых дробей $x\ (0 \leqslant x < 1)$. По-прежнему будет использоваться система счисления с основанием B.

Пусть последовательность цифр $\delta = d_1 \ldots d_n$ обозначает дробь. Если точка расположена непосредственно слева от последовательности, то значение $\tilde{\delta}$ определяется следующим образом:

$$\tilde{\delta} = \sum_{i=1}^{n} \tilde{d}_i * B^{-i} = \frac{1}{B}\left(\tilde{d}_1 + \frac{1}{B}\left(\tilde{d}_2 + \cdots + \frac{1}{B}\tilde{d}_n\right)\ldots\right) \qquad (13.9)$$

где $\tilde{d}_i = num(d_i)$ и $0 \leqslant \tilde{d}_i < B$ для любого i. Если обозначить последовательность $d_i \ldots d_n$ через δ_i, то из (13.9) непосредственно следуют рекуррентные соотношения

$$\tilde{\delta}_i = \frac{1}{B}(\tilde{d}_i + \tilde{\delta}_{i+1}) \qquad 0 \leqslant \tilde{\delta}_i < 1$$

и

$$\begin{aligned} \tilde{d}_i &= trunc\ (B * \tilde{\delta}_i) \\ \tilde{\delta}_{i+1} &= B * \tilde{\delta}_i - \tilde{d}_i \end{aligned} \qquad (13.10)$$

Алгоритм, который получается подстановкой соотношений (13.10) в схему программы (9.2), представлен процедурой (13.11). Предполагается, что переменной u последовательно присваиваются значения $\tilde{\delta}_1, \tilde{\delta}_2, \ldots$. В качестве условия окончания цикла нельзя использовать $u = 0$, так как не приходится рассчитывать на абсолютно точное представление дроби $x = \tilde{\delta}$. Поэтому алгоритм завершается, когда получено заданное количество (n) цифр. На каждом шаге дробь умножается на основание B. Применяя элементарную функцию $rep(x)$ к целой части вычисленного произведения, мы получаем очередную цифру. Для получения остальных цифр применяется тот же самый алгоритм к дробной части произведения. К счастью, эта процедура генерирует цифры в должном порядке; поэтому не требуется промежуточного запоминания в буфере и

цифры можно непосредственно посылать в текстовый файл вывода D.

```
procedure write(x: real); {0 ≤ x < 1}
   var i: 0..n; u: real; v: integer; {n > 0}
begin u := x; D↑ := '.'; put(D); i := 0;
   repeat u := B * u; v := trunc(u);
          D↑ := rep(v); put(D);
          u := u−v; i := i+1
   until i = n
end
```
(13.11)

13.3. Преобразование представлений с плавающей точкой

Как уже говорилось в разд. 8.4, для вещественных чисел в вычислительных машинах часто используется так называемое представление с плавающей точкой. В этом представлении число x изображается парой масштабированных целых чисел $\langle m, e\rangle_B$ так, что

$$x = m * B^e \qquad \frac{1}{B} \leqslant m < 1 \tag{13.12}$$

где B — основание представления с плавающей точкой. Ниже приводятся примеры, из которых видно, почему возник термин «плавающая точка».

$$\begin{aligned} &\langle 0.34567, 2\rangle &&= 34.567 \\ &\langle 0.34567, 4\rangle &&= 3456.7 \quad B = 10 \\ &\langle 0.34567, -2\rangle &&= 0.0034567 \end{aligned} \tag{13.13}$$

В вычислительной машине в качестве основания представления предпочтительно выбрать небольшую степень 2 (например, 2^k). Тогда увеличение или уменьшение показателя степени e на 1 соответствует умножению или делению коэффициента m на 2^k, которые могут выполняться простым сдвигом m на k двоичных позиций влево или вправо соответственно. Однако, поскольку вне машины мы обычно пользуемся представлением с основанием $B = 10$, при вводе или выводе чисел с плавающей точкой необходимо преобразовывать (помимо преобразования основания системы счисления) еще и показатель степени.

Самый простой способ преобразования представления $\langle m_1, e_1\rangle_{B_1}$ в представление $\langle m_2, e_2\rangle_{B_2}$ состоит в умножении m на $B_1^{e_1}$ с последующей нормализацией, т. е. многократным делением (или умножением) произведения на B_2 до тех пор, пока не удовлетворится условие $1/B_2 \leqslant m < 1$. Число делений тогда будет равняться искомому показателю степени e_2. Но при таком способе промежуточные значения произведения могут оказаться слишком большими и превысить размеры ячейки памяти, поскольку m представлено как целое

число с масштабным коэффициентом. Поэтому мы накладываем ограничение

$$\frac{1}{B} \leqslant m < B \qquad B = B_1 * B_2 \tag{13.14}$$

которое должно выполняться в любой момент, т. е. после каждого шага вычисления. Следовательно, при переводе из системы с основанием B_1 в систему с основанием B_2 необходимо чередовать умножения m на B_1 с делениями на B_2. Окончательная программа для $B_2 > B_1$ представлена в (13.15). По очевидным причинам различаются случаи, когда $e_1 \geqslant 0$ и $e_1 < 0$. Отметим важность указанных инвариантов для верификации программы.

```
procedure convert (var m: real; var e1, e2: integer);
begin e2 := 0;                                                    (13.15)
    if e1 ⩾ 0 then
        while e1 ≠ 0 do
        begin {x = m * B1^e1 * B2^e2, 1/B2 ⩽ m < 1}
              m := B1 * m; e1 := e1 — 1;
              if m ⩾ 1 then begin m := m/B2; e2 := e2 + 1 end
        end
    else repeat {x = m * B1^e1 * B2^e2, 1/B2 ⩽ m < 1}
              m := m/B1; e1 := e1 + 1;
              if m < 1/B2 then begin m := B2 * m; e2 := e2 — 1 end
        until e1 = 0
end
```

Учитывая, что числа в реальных вычислительных машинах можно представить только конечным числом цифр, следует отметить неэффективность этого алгоритма для большого показателя степени e и его неточность из-за накопления ошибок округления в длинной последовательности арифметических операций. Этот алгоритм можно улучшить, если воспользоваться таблицей множителей B_1^v в виде пар масштабированных целых чисел $\langle u, v\rangle_k$ таких, что

$$u_k * B_2^{v_k} = B_1^k \qquad \frac{1}{B_2} \leqslant u_k < 1 \tag{13.16}$$

а вычисление заменить на

$$m := m * u_{e1}, \qquad e2 := v_{e1} \tag{13.17}$$

возможно с последующей нормализацией. Однако если диапазон изменения e_1 велик, то использование такой таблицы становится практически невозможным из-за ее огромных размеров. Возможен компромиссный вариант, для которого требуется немного больше

вычислений, но зато существенно меньшая таблица. В этом случае таблица содержит значения только для

$$u_k * B_2^{v_k} = B_1^{(2^k)} \qquad \frac{1}{B_2} \leqslant u_k < 1 \tag{13.18}$$

т. е. только для показателей степеней, равных степеням 2. Затем показатель степени $e1$ разлагается в ряд по степеням 2 аналогично тому, как это делается в алгоритме умножения (7.18). В результате m и $e2$ получаются в виде произведений и сумм соответствующих элементов таблицы (см. упражнение 13.3).

УПРАЖНЕНИЯ

13.1 Напишите две процедуры, первая из которых должна
(a) читать из текстового файла *input* десятичное число, представленное в виде знака, целой части, десятичной точки и дробной части, и присваивать его значение вещественной переменной — параметру v (см. (13.6)), а вторая процедура должна
(b) писать десятичное представление вещественного параметра x в файл *output* (см. (13.8) и (13.11)). Предполагается, что $|x| \leqslant max$ и что целая и дробная части числа x представляются соответственно m и n цифрами. Старшие нули, предшествующие первой значащей цифре, следует заменить пробелами.

13.2 Напишите две процедуры преобразования представлений с плавающей точкой
(a) из системы с основанием 2 в систему с основанием 10 и
(b) из системы с основанием 10 в систему с основанием 2.
Обе процедуры должны быть аналогичны программе (13.15).

13.3 Напишите процедуру преобразования чисел с плавающей точкой из системы с основанием 2 в систему с основанием 10, используя таблицу, состоящую из n пар множителей для показателей степеней в соответствии с (13.18). Предполагается, что

$$x = m * 2^{e1} \qquad 0 \leqslant e1 < 2^n$$

для заданного n (например, 10).

13.4 Постройте программу, которая генерирует точные десятичные представления дробей $1/n$ для $n=2, 3, \ldots, 50$, как это показано ниже, и записывает полученные последовательности цифр в файл *output*.
(a) Каждая последовательность цифр должна заканчиваться, как только обнаружен первый период десятичной дроби.
(b) Непосредственно перед первой цифрой периода нужно вставить пробел.
(c) В программе нельзя использовать переменные типа *real*.
При печати текстового файла должны получиться следующие значения:

```
.5 0
.3
.25 0
.2 0
.1 6
.142857
.......
.02 0
```

Указание: Сначала напишите программу, в которой игнорируется условие (b).

13.5 Напишите процедуру, которая выдает в текстовый файл *output* целочисленное значение параметра x в римской системе счисления. Предполагается, что $x>0$. ('I'=1, 'V'=5, 'X'=10, 'L'=50, 'C'=100, 'D'=500, 'M'=1000.)

14. ОБРАБОТКА ТЕКСТОВ С ИСПОЛЬЗОВАНИЕМ МАССИВОВ И ФАЙЛОВ

В этой главе рассматриваются некоторые задачи, имеющие одно общее свойство, а именно: все они имеют дело с данными, состоящими из литер, которые можно напечатать, т. е. с текстами. Выбранные примеры типичны для задач обработки текстов и могут служить также в качестве упражнений в применении рассмотренных ранее понятий и методов.

14.1. Регулирование длины строк в текстовом файле

Если текстовый файл f состоит из подпоследовательностей литер, ограниченных либо пробелами, либо управляющими литерами **eol**, то такие подпоследовательности, не содержащие внутри себя пробелов и литер **eol**, будем называть *словами*. Количество литер в слове назовем длиной слова w. Любая подпоследовательность, ограниченная литерами **eol** (и, следовательно, не содержащая внутри себя литер **eol**), называется *строкой*. Количество литер в строке назовем длиной строки L.

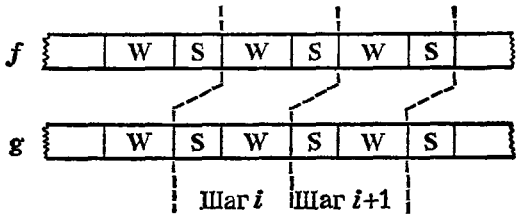
Напишем программу, которая читает файл f и генерирует файл g, содержащий те же самые нерасчлененные слова в том же порядке, но сгруппированные в строки, длина которых не превышает $Lmax$ [1]). Суммарное число литер в обоих файлах одинаково. Все слова в f имеют длину $w \leqslant wmax < Lmax$ при заданных $wmax$ и $Lmax$. Предположим, что файл f содержит по крайней мере одно слово и, следовательно, по крайней мере одну строку, которая оканчивается литерой **eol**.

Файл g получается путем копирования файла f и замены некоторых пробелов литерами **eol**, и наоборот. Поскольку слова расчленять нельзя и поскольку файлы можно генерировать только последовательно, слово можно добавить в конец файла g только тогда, когда уже известно, какую литеру (пробел или **eol**) следует поставить перед ним. Но это можно узнать только после того, как известна длина слова. Отсюда следует, что слова можно копировать только через промежуточный буфер. В качестве буфера вполне

[1]) P. Naur, Programming by Action Clusters, *BIT*, 9 (1969), 250—258.

подходит переменная-массив, имеющая *wmax* компонент (предполагается, что *wmax* сравнительно невелико).

Так как файл *f* содержит последовательность слов и литер-разделителей, программа, очевидно, должна иметь структуру повторяемой инструкции. При каждом повторении из файла *f* читается слово W и *следующий* за ним разделитель S, а затем то же самое слово и предшествующий ему разделитель записываются в файл *g* так, как это изображено ниже:



(14.1)

В том случае, когда после слова встречаются несколько разделителей (то есть кратные пробелы или несколько пустых строк), будем предполагать, что слова могут быть пустыми и что между каждыми двумя соседними разделителями находится пустое слово. Таким образом, можно условно считать, что слова и разделители строго чередуются между собой.

Повторяемая инструкция получается из схемы (10.13). Но прежде мы дадим неформальное определение двух вспомогательных процедур.

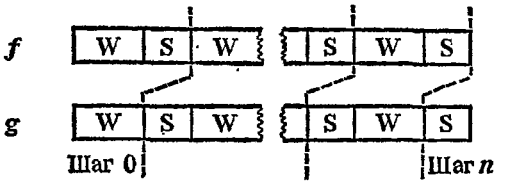
1. Процедура *readword* читает слово в буферную переменную *Z*. После выполнения процедуры *readword* переменная *ch* содержит разделитель, следующий за только что прочитанным и запомненным в *Z* словом. Переменной *w* целого типа присваивается значение, равное длине слова.
2. Процедура *writeword* записывает в конец файла *g* слово длины *w*, хранящееся в *Z*.

Поскольку длина генерируемой строки играет в нашем алгоритме важную роль, для представления текущей длины вводится переменная *L*. Тогда инструкцию, выполняемую на каждом шаге, можно сформулировать следующим образом:

```
begin readword;
    if L+w < Lmax then
        begin write (' '); L := L+1
        end else
        begin write (eol); L := 0
        end;
    writeword
end
```

(14.2)

Прежде чем приступить к дальнейшей детализации этих двух процедур, обратим внимание еще на одну проблему, а именно изучим ситуацию в начале и в конце обработки. Из рис. (14.3) ясно видно, что в начале и в конце программы необходимо написать некоторые инструкции, не согласующиеся по своему характеру с инструкцией (14.2).



(14.3)

В начале читается на одну литеру больше, чем пишется, а в конце пишется на одну литеру больше, чем читается. Окончательный вид программы представлен на рис. (14.4). Читатель может убедиться, что во вспомогательных процедурах пустые слова обрабатываются правильно. (В программе используются стандартные файлы *input* и *output* вместо *f* и *g*.)

```
    var w: 0..wmax; {длина слова}
        L: 0..Lmax; {длина строки}
        ch: char; {последняя прочитанная литера}
        Z: array [1..wmax] of char; {буфер}
    procedure readword;
    begin w :=0; read (ch);
        while (ch≠' ') ∧ (ch≠eol) do
        begin w := w+1; Z[w] :=ch; read(ch)
        end
    end;
    procedure writeword;
        var i: 1..wmax;                                    (14.4)
    begin for i := 1 to w do write(Z[i]); L := L+w
    end;
begin L := 0; readword; writeword;
    while ¬ eof(input) do
    begin readword;
        if L+w < Lmax then
            begin write(' '); L := L+1
            end else
            begin write(eol); L :=0
            end;
        writeword
    end;
    write(eol)
end.
```

Из этого примера можно извлечь следующие два урока:

1. Если последовательность информационных объектов преобразуется в другую последовательность объектов, то прежде всего необходимо сформулировать шаг преобразования для одного объекта. Причем этот шаг нужно представить в таком виде, чтобы его можно было «вложить» в инструкцию **repeat** или **while**, характеризующую условия повторения и окончания цикла. После этого рассматриваются граничные случаи и добавляются инструкции, которые выполняются в начале и в конце обработки.

2. Чтобы выделить логически связанные блоки операций, так называемые *кластеры* [1]) *действий*, необходимо ввести соответствующие процедуры, к детализации которых можно перейти позднее. Определение структуры данных для переменных, которые используются только внутри этих процедур, можно также отложить до того момента, пока не выяснятся всевозможные соображения, влияющие на выбор структуры.

14.2. Редактирование строки текста

Рассмотрим следующую задачу. Пусть даны строка текста z в виде последовательности литер

$$z=z_1z_2\ldots z_n \qquad n>0 \tag{14.5}$$

последовательность x (обычно встречающаяся в z)

$$x=x_1x_2\ldots x_k \qquad k>0$$

и последовательность y

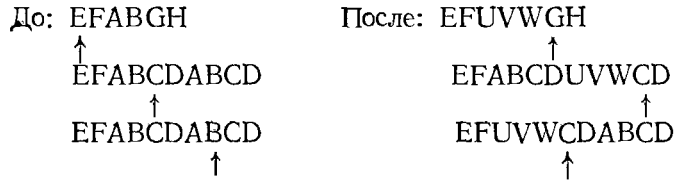
$$y=y_1y_2\ldots y_m \qquad m\geqslant 0$$

которую необходимо подставить вместо x. Логической переменной q должно быть присвоено значение *true* (истина), тогда и только тогда, когда x содержится в z и подстановка y возможна. Чтобы однозначно определить операцию подстановки в том случае, когда в тексте z последовательность x встречается несколько раз, мы введем понятие *позиции в строке* — p. Текст z просматривается всегда начиная с позиции p слева направо. Если между p и концом z последовательность x не будет найдена, то поиск будет продолжен с начала z и до позиции p, т. е. строку можно рассматривать как «закольцованную». Такой поиск называется циклическим. Например, если $x=AB$, а $y=UVW$, то текст z до и после подстановки

---

[1]) В оригинале cluster — скопление, группировка.— *Прим. ред.*

будет таким:



Подобные текстовые подстановки часто используются в системах текстового редактирования, осуществляемого с дистанционных терминалов, когда программы и данные постоянно хранятся в вычислительной системе в виде текстовых файлов.

Эту задачу можно разбить на две относительно независимые части:

(a) поиск заменяемого текста x, такого, что $z=\alpha x\beta$ и
(b) подстановка x вместо y, после чего $z=\alpha y\beta$.

Часть 1: Индекс i определяется так, что

$$x_j = A_{i+j-1} \quad \text{для} \quad j=1, \ldots, k \tag{14.6}$$

В программе (14.7) переменной q присваивается значение «текст x найден и в i нужный индекс».

$$\begin{array}{l} i := p; \\ \textbf{repeat } \{Q(i)\} \\ \quad q := x=(z_i \ \ldots \ z_{i+k-1}); \\ \quad i := i+1; \textbf{ if } i>n \textbf{ then } i := 1 \\ \textbf{until } q \vee (i=p) \end{array} \tag{14.7}$$

Утверждение $Q(i)$ состоит в следующем:

$$(x_1 \ \ldots \ x_k) \neq (z_j \ \ldots \ z_{j+k-1})$$

$$\text{для всех } j = \begin{cases} p, \ \ldots, \ i-1, & \text{если } i \geqslant p \\ p, \ \ldots, \ n \text{ и } 1, \ \ldots, \ i-1, & \text{если } i < p \end{cases}$$

Теперь мы превратим сравнение двух последовательностей литер в последовательность сравнений соответствующих пар отдельных литер.

$$\begin{array}{l} j := 1; \\ \textbf{repeat } \{P(j)\} \\ \quad q := (x[j]=z[i+j-1]); \ j := j+1 \\ \textbf{until } \neg q \vee (j>k) \end{array} \tag{14.8}$$

Утверждение $P(j)$ таково:

$$Q(i) \wedge (x_h = z_{i+h-1}) \text{ для всех } h=1, \ \ldots, \ j-1$$

Но эту программу нельзя считать законченной, так как в ней не учитывается следующая ситуация. Предположим, что

$$x_h = z_{i+h-1} \text{ для } h = 1, \ldots, h' \qquad i+h'-1 = n \qquad h' < k \tag{14.9}$$

Тогда следующими должны сравниваться литеры x_{h+1} и z_{n+1}. Но литера z_{n+1} не определена. Этот факт говорит о неполной формулировке задачи, поскольку в ней не содержится указаний о поведении программы в этой ситуации. Возможны два варианта: либо алгоритм должен закончиться и выдать результат $q=$*false*, либо текст $x_1 \ldots x_k$ следует рассматривать как закольцованную последовательность литер. Если z — строка, то второй вариант обычно не имеет смысла. Предположим, что в постановке задачи предусмотрена свобода выбора и мы останавливаемся на первой интерпретации. Тогда программа будет следующей:

```
j := 1;
repeat if i+j>n then q := false else                    (14.10)
    q := x[j]=z[i+j-1]; j := j+1
until ¬q ∨ (j > k)
```

Чтобы упростить эту программу, часто пользуются специальным приемом, с помощью которого исключается дополнительная проверка на конец строки. Для этой цели в конце строки ставится специальная литера (например, $z_{n+1}=$**eol**), которая не может встретиться ни в z, ни в x. В этом случае можно воспользоваться более простой программой (14.8). (См. также (11.6).)

Часть 2: Теперь литеры $z_i \ldots z_{i+k-1}$ заменяются на $y_1 \ldots y_m$. Эту операцию можно опять-таки разбить на два этапа: (a) превратить $z=\alpha x \beta$ в $z'=\alpha\beta$, сдвинув β на k позиций влево и (b) вставить y и получить $z''=\alpha y \beta$, предварительно сдвинув β вправо на m позиций. Обработку можно упростить, если сдвиг β делать только один раз. Но при этом нужно различать следующие случаи:

1. $m<k$ — новая последовательность короче старой; β сдвигается на $k-m$ позиций влево.
2. $m>k$ — новая последовательность длиннее старой; β сдвигается на $m-k$ позиций вправо.
3. $m=k$ — новая и старая последовательность имеют одинаковую длину и сдвига не требуется.

Теперь мы можем написать полную программу, которая осуществляет поиск и подстановку.

```
procedure Substitute;
   var i, j: 1..n+1; q: Boolean; d: integer;
begin {шаг 1: поиск x в строке z}
   i := p;
   repeat j := 1;
      repeat q := (x[j]=z[i+j-1]); j := j+1
      until ¬q ∨ (j>k);
      i := i+1; if i>n then i := 1
   until q ∨ (i=p);
   if q then
   begin {шаг 2: подстановка y вместо x}
      d := m-k; p := i;
      if d<0 then
      begin j := p+k;
         while j≤n do
         begin z[j+d] := z[j]; j := j+1
         end
      end else                                          (14 11)
      if d>0 then
      begin j := n;
         while j≥p+k do
         begin z[j+d] := z[j]; j := j-1
         end
      end;
      n := n+d; j := 1;
      while j≤m do
      begin z[p] := y[j]; p := p+1; j := j+1
      end
   end
end
```

14.3. Распознавание регулярных цепочек символов

Довольно часто приходится встречаться с задачей распознавания цепочек в тексте. Ее невозможно, например, обойти в программах, которые каким-либо образом интерпретируют или обрабатывают тексты. Обычно цепочка определяется некоторым набором правил композиции. Совокупная сложность программы, предназначенной для распознавания цепочек, очевидно, зависит от сложности и общности правил их порождения. Набор таких правил обычно называют *синтаксисом*, а задача распознавания образцов, сконструированных в соответствии с этими правилами, называется *синтаксическим анализом*.

Мы рассмотрим класс простых правил, т. е. схему правил. Каждое правило, составленное согласно этой схеме, определяет множество так называемых *предложений*, которые можно распознать

с помощью простого алгоритма анализа. Синтаксическим правилам, порождаемым с помощью этой схемы, свойственна определенная регулярность структуры, и поэтому их называют *регулярными выражениями*. Аналогично множество предложений, порождаемых с помощью регулярного выражения, называется *регулярным языком*.

Наша задача состоит в том, чтобы найти некоторые общие правила для систематического конструирования распознавателя, соответствующего данному синтаксическому правилу. Введем следующие обозначения:

1. Строчными латинскими буквами обозначаются символы основного словаря V. Все предложения являются последовательностями символов из V.
2. Прописными буквами обозначаются регулярные выражения или множество предложений (регулярный язык), определенных с помощью регулярного выражения.
3. Греческими буквами обозначаются последовательности символов, определенные над словарем V. Буквой ε обозначается пустая последовательность.
4. Множество последовательностей символов, получаемых конкатенацией предложений из A и предложений из B, называется произведением A и B.

$$AB=\{\alpha\beta;\ \alpha\in A \text{ и } \beta\in B\} \tag{14.12}$$

5. Объединение A и B обозначается как

$$A|B=\{\gamma;\ \gamma\in A \text{ или } \gamma\in B\} \tag{14.13}$$

6. Множество предложений, получаемых путем произвольного числа конкатенаций предложений из A, обозначается как A^*

$$A^*=\{\varepsilon|A|AA|AAA\ldots\} \tag{14.14}$$

Правила построения регулярных выражений следующие:

1. Любой основной символ $a\in V$ есть регулярное выражение.
2. Любое произведение двух регулярных выражений есть регулярное выражение.
3. Любое объединение двух регулярных выражений есть регулярное выражение.
4. Если A регулярное выражение, то A^* — регулярное выражение.
5. Регулярны только те выражения, которые получены с помощью правил 1—4.

Далее приводятся примеры регулярных выражений над словарем $V=\{a\ b\ c\ d\}$. Скобки имеют тот же смысл, что и в обычных выра-

жениях.

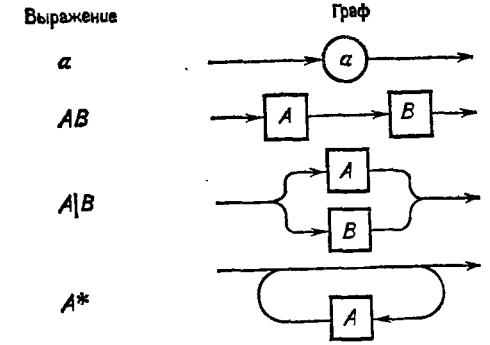
1. a
2. $(ab|bc)(d|a)$ (14.15)
3. ab^*c
4. $a((b|c)a)^*$

Множества предложений, определенные этими выражениями, будут следующими:

1. a
2. $abd\ \ aba\ \ bcd\ \ bca$
3. $ac\ \ abc\ \ abbc\ \ abbbc\ldots$ (14.16)
4. $a\ \ aba\ \ aca\ \ ababa\ \ abaca\ \ acaba\ \ acaca\ldots$

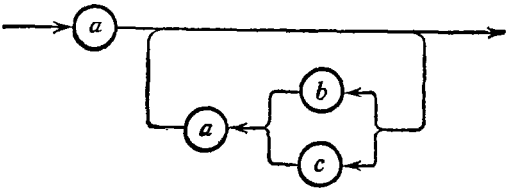
Теперь нашу задачу можно переформулировать так: по данному регулярному выражению A, определяющему множество предложений над словарем V, и любой данной последовательности α над словарем V построить алгоритм $\mathscr{P}(A)$, который определяет, является ли α предложением из A (т. е. $\alpha \in A$).

Очень просто и очевидным образом регулярные выражения представляются в виде графа. Каждое из элементарных правил построения регулярных выражений непосредственно преобразуется в одну из следующих графических диаграмм:



(14.17)

Например, если эти соответствия применить к выражению 4, то получим в результате следующий граф:



(14.18)

Графы немедленно подсказывают форму алгоритма распознавания предложений, если прохождение основного символа, заключенного

в кружок в графе, интерпретируется как распознавание символа в анализируемой последовательности α, а прохождение всего графа свидетельствует о том, что α является предложением соответствующего языка.

Предположим теперь, что последовательность α, которую требуется проанализировать, представлена текстовым файлом *input*, к которому имеется только последовательный доступ, и пусть основные символы — это литеры. Мы хотим построить алгоритм, в котором анализ предложений ведется *без возвратов* и не требуется буферизация (так сказать *опережение*) взамен возврата. Короче говоря, алгоритм должен быть таким, чтобы каждое решение (представленное разветвлением путей в графе) принималось в результате проверки только одного очередного символа.

Итак, мы описываем схему, следуя которой по заданному выражению можно конструировать алгоритм анализа. Но правила конструирования применимы лишь тогда, когда выражение *детерминированное*, т. е. если выражение не содержит составляющих вида

$$A|B \text{ и } A^*B, \tag{14.19}$$

таких, что A и B начинаются с одинаковых символов. Любой символ s, с которого начинается предложение из A, называется начальным символом предложения из A. Приведем несколько примеров регулярных выражений и их детерминированных эквивалентов (предполагая, что A и B не имеют общих начальных символов).

$$\begin{gathered} aA|aB=a(A|B) \\ (aA)^*aB=a(Aa)^*B \\ (aA)^*a|(aB)^*a=a((Aa)^*|(Ba)^*) \end{gathered} \tag{14.20}$$

Далее мы приводим правила конструирования, в которых $\mathcal{P}(X)$ обозначает схему программы, соответствующую регулярному выражению X. Предполагается, что переменная *ch* всегда содержит очередной сканируемый символ, а процедура *test*(x) осуществляет проверку равенства $ch=x$.

Правила конструирования

1. $\mathcal{P}(a) = test(a);\ read(ch)$
2. $\mathcal{P}(AB) = \mathcal{P}(A);\ \mathcal{P}(B)$
3. $\mathcal{P}(aA|B) =$ **if** $ch=a$ **then** **begin** $read(ch);\ \mathcal{P}(A)$ **end** **else** $\mathcal{P}(B)$ (14.21)
4. $\mathcal{P}((aA)^*) =$ **while** $ch=a$ **do** **begin** $read(ch);\ \mathcal{P}(A)$ **end**

Чтобы получить полную программу, соответствующую выражению A, за которым следует признак конца (например, точка), схему

$\mathcal{P}(A)$ нужно поместить в «рамку»:

```
var ch:char;
begin read(ch); P(A); {ch='.'}
end.
```
(14.22)

Приведем примеры программ анализа, полученные с помощью рассмотренных правил конструирования по заданным детерминированным регулярным выражениям.

1. $\mathcal{P}(ab^*c)=$
```
test (a); read (ch);
while ch=b do read (ch);
test (c); read (ch)
```
(14.23)

2. $\mathcal{P}(a(b(c|d))^*e)=$
```
test (a); read (ch);
while ch=b do
    begin read (ch);
        if ch=c then read (ch) else
        begin test (d); read (ch)
        end
    end;
test (e); read (ch)
```

3. $\mathcal{P}((ab^*c)^*d)=$
```
while ch=a do
begin read (ch);
    while ch=b do read (ch);
    test (c); read (ch)
end;
test (d); read (ch)
```

Процедура *test*(x) была предназначена для проверки равенства $ch=x$. Если отношение равенства не выполняется, то последовательность сканируемых литер не является предложением языка, определенного соответствующим регулярным выражением, и процесс анализа можно прекратить. Это можно сделать, например, с помощью инструкции перехода

```
goto L
```
(14.24)

где L — метка инструкции, к которой осуществляется переход. Перед любой инструкцией в программе может стоять метка и двоеточие. Инструкцией **goto** следует пользоваться в «особых» и «нестандартных» случаях, когда необходимо нарушить естественную структуру алгоритма (например, если обнаруживаются входные данные, не удовлетворяющие заданным спецификациям). Не рекомендуется пользоваться переходами при программировании регулярных итеративных процессов и условно выполняемых инструкций, так как с введением явных переходов структура текста про-

граммы уже более не отражает структуры выполняемых по этой программе вычислений. Более того, отсутствие соответствия между структурой текста и структурой вычислений чрезвычайно затрудняет понимание программы и существенно усложняет задачу верификации.

Алгоритмы анализа текстовых цепочек служат основой для программ обработки структурных текстов (например, для компиляторов, которые осуществляют трансляцию или интерпретацию программ). По существу, при конструировании таких программ достаточно включить дополнительные инструкции в программы, выполняющие синтаксический анализ. Эти дополнительные инструкции порождают новый текст из исходного анализируемого текста. С их добавлением простой процесс анализа (дающий в качестве результата логическое значение) становится процессом трансляции, для которого, вообще говоря, нельзя считать удовлетворительным заурядный переход в конец программы в случае, когда встречается неправильный исходный текст. Тем не менее алгоритм, играющий роль «скелета», который затем обрастает процедурами трансляции, значительно повышает надежность получаемого транслятора, поскольку последний конструируется с помощью стандартной методики согласно заданному набору правил.

УПРАЖНЕНИЯ

14.1 Определите, правильно ли решается в этих программах задача, рассмотренная в разд. 14.1.

(a)
```
var i, L, w: integer; ch: char;
    Z: array [1..wmax] of char;                    (14.25)
begin L := 0;
    repeat w := 0; read(ch);
        while (ch ≠ ' ') ∧ (ch ≠ eol) do
            begin w := w+1; Z[w] := ch; read(ch)
            end;
        if w > 0 then
            begin if L+w < Lmax then
                begin write (' '); L := L+1
                end else
                begin write (eol); L := 0
                end;
            for i := 1 to w do write (Z[i]); L := L+w
        end
    until eof(input)
end.
```

(b)
```
var i, L, w:integer; ch:char;
    Z:array [1..wmax] of char;
begin L := Lmax;
    repeat if (ch =' ') ∨ (ch=eol) then read(ch) else
        begin w := 0;
            repeat w := w+1; Z[w] := ch; read(ch)
            until (ch =' ') ∨ (ch=eol);
            if L+w < Lmax then
```

```
                    begin write (' '); L := L+1
                    end else
                    begin write (eol); L := 0
                    end;
                for i := 1 to w do write (Z[i]); L := L+w
            end
        until eof (input);
        write (eol)
    end.
```

14.2 В печатаемой строке длины L некоторые литеры помечаются маркерами. В качестве маркера используется определенная литера (например, '*'), которая печатается под помечаемой литерой. Номера (индексы) маркируемых литер задаются массивом

$$p_1, p_2, \ldots, p_n \qquad n \geqslant 0, \qquad 1 \leqslant p_i \leqslant L$$

где

$$p_i \leqslant p_j \text{ для всех } i<j$$

Пример: $p=(1, 5, 7, 14, 19)$, $n=5$
Строка текста: Ета сатача премитифна
Строка маркеров: * * * * *
Рассмотрите предложенные варианты. Какие из них правильные? Найдите условия, при которых проявятся ошибки.

(a)
```
k := 1;
for i := 1 to L do
begin if i=p[k] then
      begin write ('*'); k := k+1
      end
      else write (' ')
end
```

(b)
```
k := 1; p[n+1] := 0;
for i := 1 to L do
begin if i=p[k] then
      begin write ('*');
            repeat k := k+1 until i ≠ p[k]
      end
      else write (' ')
end
```

(c)
```
i := 1;
for k := 1 to n do
begin repeat write (' '); i := i+1
        until i=p[k];
  write ('*'); i:=i+1
end
```

(d)
```
i := 1; k := 1;
repeat
   while i < p[k] do
     begin write (' '); i := i+1
     end;
   if i=p[k] then
     begin write ('*'); i := i+1
     end;
   k := k+1
until k > n
```

(e)
```
i := 1; k := 1;
while k ≤ n do
begin while i < p[k] do
      begin write (' '); i := i+1
      end;
      while p[k]=i do k := k+1;
      write ('*')
end
```

14.3 Напишите программу, которая подсчитывает, сколько раз в файле f встречается каждая литера. Результат представьте в виде переменной N, которая описывается как

var N: **array** [*char*] **of** *integer*

чтобы в компоненте $N[c]$ можно было подсчитать количество литер c в файле f. Кроме того, переменная D должна указывать количество счетчиков, значения которых отличны от нуля.

14.4 Даны два массива A и B, которые содержат так называемые слова, т. е. последовательности букв. Иначе говоря, для $i=1, \ldots, M$

$$A_{i,1} \ldots A_{i,m_i} \text{ и } B_{i,1} \ldots B_{i,n_i} \quad m_i,\ n_i < N$$

являются буквами, а

$$A_{i,m_i+1} \ldots A_{i,N} \text{ и } B_{i,n_i+1} \ldots B_{i,N}$$

являются пробелами. Кроме того, дан файл *input*, содержащий слова (последовательности букв), разделенные пробелами и/или литерами, указывающими конец строки (eol). Напишите программу, которая читает файл *input* и генерирует файл *output*, заменяя при этом каждое слово, совпадающее с некоторым словом A_i массива A, на соответствующее ему слово B_i массива B.
Указание: В программе следует воспользоваться буфером, вмещающим не более N литер. Используйте процедуру *Transmitword*, которая сравнивает слово в буфере Z с A и отправляет в файл *output* B_i (если $Z=A_i$) или Z (если ни одно A_i не совпадает с Z). Будьте осторожны при применении следующей непроверенной процедуры (здесь $Z=Z_1 \ldots Z_k$):

```
procedure Transmitword;
    var i: 0..M; j: 0..N; f: Boolean;
begin i := 0; {Z[k+1]=' ', 0 < k < N}
    repeat i := i+1; j := 0;
        repeat j := j+1; f := Z[j] ≠ A[i, j]
        until f ∨ (i=k)
    until ¬f ∨ (i=M);
    if f then Emit (Z) else Emit (B[i])
end
```

14.5 Задан массив S, содержащий так называемые ключевые слова. Пусть $S_{i,1}, \ldots, S_{i,n_i}$—буквы и $1 \leqslant n_i < N$. Как и в упражнении 14.4, файл *input* состоит из слов (последовательностей букв), разделенных пробелами и/или *eol*-литерами. Каждая строка содержит не более L литер.
Напишите программу, которая генерирует выходной файл, содержащий те же самые строки входного файла, но циклически сдвинутые так, чтобы ключевые слова начинались с фиксированной k-й позиции. Если в строке

нет ключевых слов, то такая строка не передается в выходной файл, если строка содержит n ключевых слов, то эта строка выдается n раз, сдвинутая на разное количество позиций. (В данном случае необходима буферная переменная, вмещающая по крайней мере L литер.)

14.6 Напишите программу в соответствии с правилами конструирования (14.21) и (14.22) для анализа арифметических выражений. Словарь состоит из следующих символов

$$V=\{\lambda+-*/.\}$$

где λ обозначает произвольную букву. Анализируемые структуры задаются следующим регулярным выражением:

$$\lambda((*|/)\lambda)^*((+|-)\lambda((*|/)\lambda)^*)^*.$$

15. ПОШАГОВАЯ РАЗРАБОТКА ПРОГРАММ

Наши примеры в предыдущих главах убедительно показывают, что программирование, понимаемое как проектирование и формулирование алгоритмов, как правило, процесс сложный, требующий учета многочисленных деталей и владения специальной техникой. Очевидно также, что только в исключительных случаях будет существовать единственное во всех отношениях удовлетворительное решение. Обычно решений так много, что выбор оптимальной программы требует не только тщательного анализа существующих алгоритмов и вычислительных машин, но также и наиболее частых случаев использования программы. Если бы программирование было строго детерминированным процессом, подчиняющимся фиксированному набору правил, то оно давно было бы автоматизировано.

Как и в любом инженерном деле, конструирование изделия, в данном случае алгоритма, требует осмысливания, исследований и принятия решений. На ранних стадиях внимание обращено главным образом на глобальные проблемы, и в первом эскизном проекте упускаются из виду многие детали. По мере продвижения процесса проектирования задача разбивается на подзадачи, и постепенно все большее внимание уделяется подробному описанию проблемы и характеристикам имеющихся инструментов. В программировании благодаря абстрактной природе получаемого изделия появляются совершенно неожиданные возможности. В отличие от других областей инженерной деятельности в программировании изделия можно конструировать (и испытывать) без материальных затрат; они свободны от побочных физических эффектов, связанных со старением и применением материалов низкого качества. Поэтому программисты могут и должны разрабатывать много версий алгоритмов и тщательно сравнивать и анализировать их, прежде чем сделать окончательный выбор.

Вероятно, наиболее общая тактика программирования состоит в разложении процесса на отдельные действия, а соответствующих программ на отдельные инструкции. На каждом таком шаге декомпозиции нужно удостовериться, что

(a) решения частных задач приводят к решению общей задачи,
(b) выбранная последовательность индивидуальных действий разумна и
(c) выбранная декомпозиция позволяет получить инструкции, в каком-либо смысле более близкие к языку, на котором в конечном счете будет сформулирована программа.

Как раз последнее требование исключает возможность прямолинейного продвижения от первоначальной постановки задачи к программе, которая должна получиться в конечном итоге. Всякий этап декомпозиции сопровождается формулированием частных программ. В процессе этой работы может обнаружиться, что выбранная декомпозиция неудачна в том или ином смысле хотя бы потому, что подпрограммы неудобно выражать с помощью имеющихся средств. В этом случае один или несколько предыдущих шагов декомпозиции следует пересмотреть заново.

Если видеть в поэтапной декомпозиции и одновременном развитии и детализации программы постепенное продвижение вглубь, то такой метод при решении задач можно характеризовать как *нисходящий* (сверху вниз). И наоборот, возможен такой подход к решению задачи, когда программист сначала изучает имеющуюся в его распоряжении вычислительную машину и/или язык программирования, а затем собирает некоторые последовательности инструкций в элементарные процедуры или «кластеры действий», типичные для решаемой задачи. Элементарные процедуры затем используются на следующем уровне иерархии процедур. Такой метод (из глубины примитивных машинных команд к задаче, лежащей «на поверхности») называется *восходящим* (снизу вверх). На практике разработку программы никогда не удается провести строго в одном направлении (сверху вниз или снизу вверх). Однако при конструировании новых алгоритмов нисходящий метод обычно доминирует. С другой стороны, при адаптации программы к несколько измененным требованиям предпочтение зачастую отдается восходящему методу.

Оба метода позволяют разрабатывать программы, которым присуща структура,— свойство, отличающее их от аморфных линейных последовательностей инструкций или команд машины. И чрезвычайно важно, чтобы используемый язык в полной мере отражал эту структуру. Только тогда окончательный вид полученной программы позволит применить систематические методы верификации и проследить историю ее развития. Но в бесструктурной записи, например в массе двоичных цифр в памяти машины, в программе не хватает именно того, что позволяет человеку отделить полезную информацию от шума.

Если программа разбивается на подпрограммы, то для представления результатов и аргументов часто приходится вводить

новые переменные и таким образом устанавливать связь между подпрограммами. Такие переменные следует вводить и описывать на том этапе разработки, на котором они потребовались. Более того, детализация описания процесса может сопровождаться детализацией описания структуры используемых переменных. Следовательно, в языке должны быть средства для отражения иерархической структуры данных. Из сказанного видно, какую важную роль играет понятие процедуры, локальность объектов и структурирование данных в связи с пошаговой разработкой программы.

Следующие четыре примера призваны проиллюстрировать приведенные выше довольно абстрактные соображения. Ни один из примеров (за исключением, быть может, первого) нельзя отнести к типичным задачам обработки данных. Однако они очень хорошо подходят для применения и демонстрации метода пошаговой разработки программы, но не требуют конструирования чрезвычайно длинных и сложных программ. Тем не менее объем пояснений, которые потребовались для описания процесса разработки этих программ, служит серьезным напоминанием о том, что конструирование алгоритмов является далеко не простым делом.

15.1. Решение системы линейных уравнений

Предположим, что нам нужно написать программу для вычисления неизвестных значений $x_1, \ldots, x_n$ при решении системы линейных уравнений

$$\sum_{j=1}^{n} a_{ij} * x_j = b_i \qquad i = 1, \ldots, n \tag{15.1}$$

если все a_{ij} и b_i известны. Приведем пример системы линейных уравнений при $n=3$.

$$\begin{aligned} x_1 + 2*x_2 + 5*x_3 &= 4 \\ 3*x_1 + x_2 + 4*x_3 &= 11 \\ -2*x_1 + 5*x_2 + 9*x_3 &= -7 \end{aligned} \tag{15.2}$$

Из различных методов решения уравнений мы выберем получивший широкое распространение (еще до появления вычислительных машин) метод Гаусса — метод последовательного исключения неизвестных. На первом шаге мы, используя (15.1) при $i=1$, получаем выражение для x_1.

$$x_1 = \left(b_1 - \sum_{j=2}^{n} a_{1j} * x_j\right) \Big/ a_{11} \tag{15.3}$$

Затем x_1 подставляется в оставшиеся $n-1$ уравнений. В результате получается система $n-1$ уравнений с $n-1$ неизвестными $x_2, \ldots, x_n$. Этот процесс называется *исключением неизвестного*. Если

его повторить n—1 раз, то система сведется к одному уравнению с одним неизвестным, а такое уравнение решается совсем просто.

Чтобы получить программу, точно отражающую этот процесс, мы в общем виде сформулируем k-й шаг исключения неизвестного. На k-м шаге мы имеем дело с n—k+1 линейными уравнениями.

$$\sum_{j=k}^{n} a_{ij}^{(k)} * x_j = b_i^{(k)} \qquad i = k, \ \ldots, \ n \tag{15.4}$$

Новые коэффициенты $a_{ij}^{(k+1)}$ и $b_i^{(k+1)}$ вычисляются так, чтобы получилась система n—k уравнений:

$$\sum_{j=k+1}^{n} a_{ij}^{(k+1)} * x_j = b_i^{(k+1)} \qquad i = k+1, \ \ldots, \ n \tag{15.5}$$

Эти коэффициенты получаются как линейная комбинация коэффициентов k-го и i-го уравнений (строк)

$$\begin{aligned} a_{ij}^{(k+1)} &= a_{ij}^{(k)} - (a_{kj}^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} \\ b_i^{(k+1)} &= b_i^{(k)} - (b_k^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} \end{aligned} \tag{15.6}$$

для i, j=k+1, ..., n. Теперь k-е уравнение вычитается из i-го и затем умножается на величину, выбранную так, что для j=k и любого i

$$a_{ik}^{(k+1)} = a_{ik}^{(k)} - (a_{kk}^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} = 0 \tag{15.7}$$

Это значит, что в новой системе все коэффициенты при x_k равны нулю, и таким образом исключается x_k. Заметим, кстати, что нет необходимости вычислять коэффициенты $a_{ik}^{(k+1)}$.

После n—1 шагов система сведется к уравнению

$$a_{nn}^{(n)} * x_n = b_n^{(n)} \tag{15.8}$$

из которого непосредственно вычисляется x_n. Остальные неизвестные получаются подстановкой уже вычисленных неизвестных в полученные ранее уравнения. Например, для вычисления x_{n-1} достаточно подставить x_n в

$$a_{n-1,\,n-1}^{(n-1)} * x_{n-1} + a_{n-1,\,n}^{(n-1)} * x_n = b_{n-1}^{(n-1)} \tag{15.9}$$

Этот процесс называется *обратной подстановкой*. Обратную подстановку на k-м шаге можно выразить следующей общей формулой

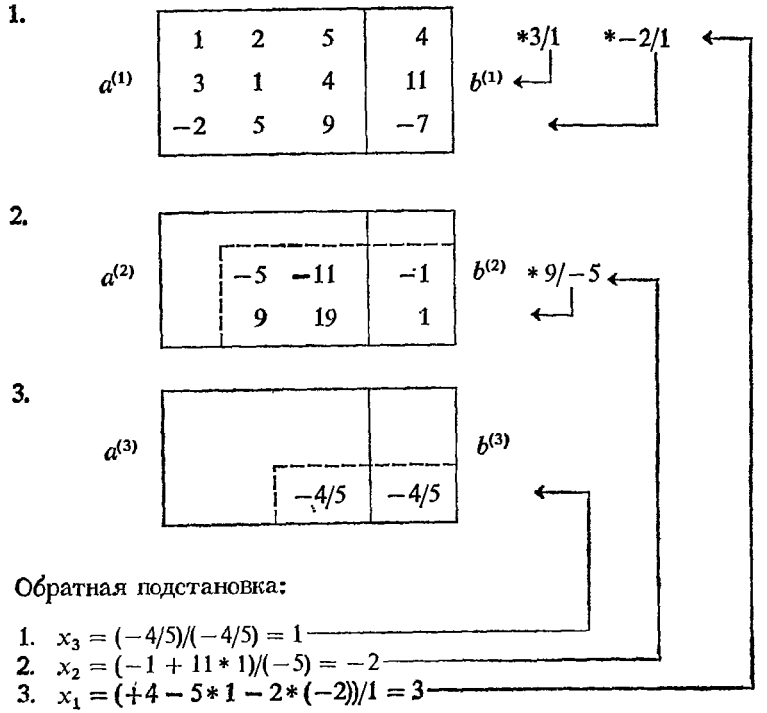
$$x_k = \left(b_i^{(k)} - \sum_{j=k+1}^{n} a_{ij}^{(k)} * x_j \right) \Big/ a_{ik}^{(k)} \tag{15.10}$$

для произвольного i, такого, что $k \leqslant i \leqslant n$. (Позднее мы поясним, почему обычно выбирается i=k.) Обратите внимание на последовательность, в которой выполняются обратные подстановки. Она

определяется тем, что для вычисления x_k должны быть известны значения $x_{k+1}, \ldots, x_n$.

Весь процесс решения демонстрируется на примере системы уравнений (15.2). Уравнения представляются своими коэффициентами, записанными в виде матрицы.

Исключение неизвестных:



При создании программы, соответствующей этому процессу, необходимо учитывать одно важное свойство описанного алгоритма. А именно: когда вычисляются коэффициенты $a^{(k+1)}$ и $b^{(k+1)}$, то нужны только коэффициенты $a^{(h)}$ и $b^{(h)}$ при $h=k$ и не нужны коэффициенты с $h<k$. Возникает вопрос, можно ли использовать лишь две переменные A и B для представления всех $a^{(k)}$ и $b^{(k)}$. Чтобы ответить на этот вопрос, необходимо исследовать, какие коэффициенты потребуются на последующих шагах обратной подстановки. Но мы уже видели, что можно воспользоваться любым из уравнений с индексом $i=k, \ldots, n$. Если для вычисления x_k выбирается k-е уравнение ($i=k$), то доступны как раз те строки A и B, которые содержат коэффициенты $a_{ij}^{(k)}$ и $b_i^{(k)}$ для $i=k+1, \ldots, n$. Таким образом, их можно заменить соответственно на $a_{ij}^{(k+1)}$ и $b_i^{(k+1)}$. Из

всего этого следует, что достаточно иметь по одному экземпляру переменных A и B, которые будут последовательно хранить коэффициенты $a_{ij}^{(1)}$, $a_{ij}^{(2)}$, ..., $a_{ij}^{(n)}$ и $b_i^{(1)}$, ..., $b_i^{(n)}$. Объем памяти, необходимый для хранения этих коэффициентов, резко сокращается. Так, вместо

$$n^2+(n-1)^2+\ldots+2^2+1^2=\frac{1}{6}(2n^3+3n^2+n)$$

ячеек для $a^{(1)}$, ..., $a^{(n)}$ и

$$n+(n-1)+\ldots+2+1=\frac{1}{2}(n^2+n)$$

ячеек для $b^{(1)}$, ..., $b^{(n)}$ расходуется только n^2+n ячеек, отведенных переменным A и B. Во многих случаях такое экономное использование памяти оказывается решающим фактором при выборе метода. Именно поэтому применение метода исключения Гаусса экономически оправдано на существующих вычислительных машинах. (Заметим, что по аналогичным соображениям можно было бы использовать общую память для x_j и b_i и не вводить переменную X.)

Учитывая сказанное, мы введем следующие описания переменных:

```
var A: array [1.. n, 1.. n] of real
    B, X: array [1.. n] of real                    (15.11)
```

и сформулируем программу

```
Вариант 1: begin „присвоить значения переменным A и B“;
             for k := 1 to n-1 do                  (15.12)
                 begin „вычислить a^(k+1) и b^(k+1), используя
                       a^(k) и b^(k), в соответствии с (15.6)“
                 end;
                 k := n;
                 repeat „вычислить x_k в соответствии с (15.10)“;
                        k := k-1
                 until k=0
           end
```

Вариант 2 получается в результате детализации инструкции „вычислить ... (15.6)“, которую можно записать следующим образом:

```
Вариант 2: for i := k+1 to n do
           begin „вычислить i-ю строку a^(k+1) и b^(k+1)
                 в соответствии с (15.6)“
           end                                     (15.13)
```

В этом месте следует обратить внимание на то, что при вычислении $a_{ij}^{(k+1)}$ (и $b_i^{(k+1)}$) по формуле (15.6) множитель $a_{ij}^{(k)}/a_{kk}^{(k)}$ (соответственно $b_k^{(k)}/a_{kk}^{(k)}$) не зависит от параметра цикла i. Следуя основному правилу, по которому нужно избегать повторного вычисления неменяющихся частей выражений, мы вынесем из инструкции **for** операцию деления на $a_{kk}^{(k)}$. Но возникает вопрос: где хранить результаты деления? Можно ли просто заменить $a_{kj}^{(k)}$ и $b_k^{(k)}$ соответственно на $a_{kj}^{(k)}/a_{kk}^{(k)}$ и на $b_k^{(k)}/a_{kk}^{(k)}$? Очевидно, нужно рассмотреть, как повлияет эта замена на процесс обратной подстановки. Но мы знаем, что умножение всех коэффициентов на один и тот же множитель не изменяет значения неизвестных. Кроме того, при делении на $a_{kk}^{(k)}$ значение самого $a_{kk}^{(k)}$ становится равным 1, и, следовательно, в процессе обратной подстановки деление становится излишним. Поэтому предлагаемая замена не только допустима, но и желательна. Это приводит к третьему варианту программы для фазы исключения неизвестных.

Вариант 3:
```
for k := 1 to n—1 do
begin p := 1/A[k, k];                                   (15.14)
      for j := k+1 to n do A[k, j] :=
      p*A[k, j]; B[k] := p*B[k];
      for i := k+1 to n do
      begin „вычислить a_i^(k+1) и b_i^(k+1) в соответствии с (15.6)“
      end
end
```

При детализации инструкции „вычислить ... (15.6)“ следует учитывать, что $A[k, j]$ имеет значение $a_{kj}^{(k)}/a_{kk}^{(k)}$ (на k-м шаге).

Вариант 4:
```
for i := k+1 to n do                                    (15.15)
begin for j := k+1 to n do A[i, j] :=
          A[i, j]—A[i, k]*A[k, j];
      B[i] := B[i]—A[i, k]*B[k]
end
```

Теперь тот же процесс постепенной детализации мы применяем к фазе обратной подстановки (15.12). Обратная подстановка реализуется как последовательность вычитаний по формуле (15.10). Заметим, что деление на $a_{kk}^{(k)}$ уже было выполнено во время исключения неизвестных.

Вариант 5:
```
begin t := B[k];                                        (15.16)
      for j := k+1 to n do t:=t—A[k, j]*X[j];
      X[k] := t
end
```

Обратите внимание, что здесь существенно используется то свойство инструкции **for**, в соответствии с которым не выполняется

никаких действий, если конечное значение меньше начального значения параметра цикла.

Окончательный вид программы решения системы линейных уравнений приводится в (15.17). Заметим, что теперь требуется n шагов исключения неизвестных (вместо $n-1$ шагов), так как на n-м шаге выполняется деление $b_n^{(n)}$ на $a_{nn}^{(n)}$.

```
{Решение системы n линейных уравнений методом исключения неизвестных по Гауссу.}
var i, j, k: 1..n;                                                    (15 17)
    p, t: real;
    A: array [1..n, 1..n] of real;
    B, X: array [1..n] of real;
begin {присвоить значения переменным A и B}
    for k := 1 to n do
    begin p := 1.0/A[k, k];
        for j := k+1 to n do A[k, j] := p*A[k, j];
        B[k] := p*B[k];
        for i := k+1 to n do
        begin for j := k+1 to n do
                  A[i, j] := A[i, j]-A[i, k]*A[k, j];
              B[i] := B[i]-A[i, k]*B[k]
        end
    end;
    k := n;
    repeat t := B[k];
        for j := k+1 to n do t := t-A[k, j]*X[j];
        X[k] := t; k := k-1
    until k=0
    {X[1]...X[n] являются решением системы уравнений}
end
```

Особое внимание нужно уделить операции деления, поскольку алгоритм перестает работать, если делитель окажется равным нулю. Это тем более важно, если арифметические операции выполняются с конечной точностью, так как даже близкий к нулю делитель может привести к ошибке или в лучшем случае к совершенно обескураживающим результатам. То, что перестановка строк (уравнений) и столбцов A и B не влияет на значения неизвестных, позволяет нам в такой последовательности производить исключение неизвестных, чтобы вообще избежать деления на нуль или близкое к нулю значение. Делители называются *осевыми элементами*. В качестве осевого элемента мы можем выбрать компоненту $a_{ij}^{(k)}$ с наибольшим абсолютным значением. Очевидно, алгоритм при этом усложняется, но без поиска осевого элемента невозможно, как правило, обойтись, если мы хотим иметь удовлетворительную

точность и надежность вычислений. Если на некотором шаге исключения не удается найти отличный от нуля осевой элемент, то такая система уравнений называется *сингулярной*, т. е. она имеет не единственное решение. Если же нет делителя, который существенно отличается от нуля, то такая система называется *плохо обусловленной* (см. упражнение 15.2).

Краткий анализ алгоритма показывает, что в самом внутреннем цикле операции

$$A[i, j] := A[i, j] - A[i, k] * A[k, j] \qquad (15.18)$$

выполняются

$$(n-1)^2 + (n-2)^2 + \ldots + 2^2 + 1^2 = \frac{1}{6}(2n^3 - 3n^2 + n) \qquad (15.19)$$

раз. Количество вычислений растет как третья степень числа n.

15.2. Нахождение минимального числа, равного двум суммам двух различных пар натуральных чисел, возведенных в третью степень

На этом примере демонстрируется метод пошаговой детализации программы, описывающей процесс перебора, критерии которого последовательно уточняются. Задача состоит в том, чтобы найти наименьшее число x, такое, что

$$x = a^3 + b^3 = c^3 + d^3 \qquad (15.20)$$

где a, b, c, d — натуральные числа, причем $a \neq c$ и $a \neq d$.

Если не привлекать глубокие сведения из теории чисел, то, по-видимому, разумно искать решение, просматривая возможных кандидатов в x (в порядке их возрастания), и прекратить процесс, как только встретятся подряд два равных кандидата. Кандидатами считаются всевозможные суммы двух натуральных чисел, возведенных в третью степень. Первый вариант такой программы можно записать следующим образом:

```
Вариант 1: x := 2; {2 = 1^3 + 1^3}
           repeat min := x;                          (15.21)
                  x := „следующий по порядку кандидат“
           until x = min
```

Задача тем самым сводится к детализации инструкции, определяющей поиск очередного кандидата. Чтобы найти ключ к организации такого поиска, рекомендуем вычислить «вручную» несколько первых членов этой последовательности. Для иллюстрации этого метода здесь приводятся суммы квадратов, а не кубов. В табл. (15.22) суммы не упорядочены и расположены так, что $S_{ij} = i^2 + j^2$.

i \ j	1	2	3	4	5	6	7	8	...
1	2								
2	5	8							
3	10	13	18						
4	17	20	25	32					
5	26	29	34	41	50				
6	37	40	45	52	61	72			
7	50	53	58	65	74	85	98		
8	65	68	73	80	89	100	113	128	
9	82	85	...						
...									

(15.22)

Из таблицы видно, что

$$50=1^2+7^2=5^2+5^2$$
$$65=1^2+8^2=4^2+7^2$$
$$85=2^2+9^2=6^2+7^2$$

и число 50 является искомым наименьшим числом. Главная задача теперь — найти метод, который позволит получать кандидатов в порядке возрастания их величин, т. е. в виде такой последовательности

2, 5, 8, 10, 13, 17, ..., 45, 50, 50

Следующие отношения существенно влияют на дальнейшую детализацию программы.

1. $S_{ij}>S_{ik}$ для любого i и любых $j>k$.
2. $S_{ij}>S_{kj}$ для любого j и любых $i>k$.
3. $S_{ij}=S_{ji}$, т. е. достаточно рассматривать S_{ij} только для $j\leqslant i$.

Из утверждения 1 следует, что нет необходимости постоянно хранить в памяти всю строку кандидатов. Достаточно продвигаться по каждой строке слева направо и помнить только последнего сгенерированного кандидата. Следовательно, таблицу можно представить переменной:

var S : **array** [1..?] **of** *integer* (15.23)

Чтобы упростить вычисление следующего кандидата в строке, необходима еще одна переменная

var j : **array** [1..?] **of** *integer* (15.24)

Значением k-го элемента переменной j будет индекс последнего сгенерированного кандидата k-й строки таблицы, т. е.

$$S[k]=k^3+j[k]^3 \qquad (15.25)$$

Если вместо самого кандидата x использовать индекс i последнего рассмотренного кандидата и ввести функцию $p(k)=k^3$, то второй вариант программы можно записать следующим образом:

Вариант 2:
```
i := 1;
for k := 1 to ? do
    begin j[k] := 1; S[k] := p(k)+1
    end;
repeat min := S[i];
    1: „j[i] увеличить на 1 и в S[i] подставить следующего кандидата в строке i“
    2: „определить новое значение i как индекс строки с наименьшим кандидатом“
until S[i] = min
```
(15.26)

Этот вариант программы нельзя считать окончательным, поскольку в нем не определено количество компонент $S[k]$, которым присваиваются начальные значения k^3+1^3, и, кроме того, в инструкции 2 неявно подразумевается поиск среди бесконечного числа кандидатов. Однако, учитывая утверждение 2, поиск (и установку начального значения) можно ограничить компонентами $S[k]$, для которых $k\leqslant ih$, где ih определяется как наименьший индекс, такой, что $j[ih]=1$ (если $j[k]>1$ для любого $k<ih$). Эти соображения приводят нас к следующему варианту программы:

Вариант 3:
```
i := 1; ih := 2;
j[1] := 1; S[1] := 2; j[2] := 1;
S[2] := p(2)+1;
repeat min := S[i];
    1: if j[i] = 1 then „увеличить ih на 1 и установить начальное значение S[ih]“
    2: „увеличить j[i] на 1 и подставить в S[i] следующего кандидата в строке i“
    3: „определить i, такое, что S[i] = min(S[1]...S[ih])“
until S[i] = min
```
(15.27)

При дальнейшей детализации трех помеченных инструкций следует иметь в виду, что генерирование кандидатов (в строке 2) следует прекратить, как только $j[i]=i$. С одной стороны, это полезно, поскольку симметричность таблицы позволяет ограничить поиск кандидата левым нижним треугольником, а с другой стороны — просто необходимо, иначе будут генерироваться и распознаваться как равные пары значений a^3+b^3 и b^3+a^3. Так как индексы $j[i]$ могут достигать предела (и тогда i-я строка исключается из рассмотрения), нам придется ввести нижнюю границу для индекса i, которую мы обозначим il. Выбор кандидатов таким образом ограничивается строками с индексами, лежащими между il и ih. Значение il увели-

чивается на 1 всякий раз, когда строка исключается из рассмотрения.

Вариант 4: $i := 1$; $il := 1$; $ih := 2$; ... (15.28)

```
repeat min := S[i];
    if j[i] = i then il := il + 1 else
        begin 1: if j[i] = 1 then „увеличить
                    ih на 1 и установить на-
                    чальное значение S[ih]“
                 2: „увеличить j[i] на 1 и под-
                    ставить в S[i] ...“
        end;
                 3: „определить i, такое, что
                    S[i] = min (S[il] ... S[ih])“
until S[i] = min
```

Теперь необходима детализация инструкций 1—3. Если в инструкции 3 минимум ищется простым линейным просмотром $S_{il}, \ldots, S_{ih}$ (см. 11.19), то мы получаем следующую часть программы:

```
3: i := il; k := i;
   while k < ih do                                   (15.29)
   begin {S[i] = min (S[il] ... S[k])} k := k + 1;
      if S[k] < S[i] then i := k
   end
```

Столь же просто и прямолинейно можно провести детализацию инструкций 2 и 1.

```
2: j[i] := j[i] + 1; S[i] := p(i) + p(j[i])          (15.30)
```

```
1: if j[i] = 1 then
   begin ih := ih + 1; j[ih] := 1;
         S[ih] := p(ih) + 1                          (15.31)
   end
```

Полученную таким образом программу можно улучшить, если исключить повторное обращение к функции $p(i)$ при вычислении сумм $S[i]$. Это усовершенствование легко реализовать, если каждое новое возведение в третью степень выполнять тогда и только тогда, когда увеличивается ih. Пусть значения третьих степеней имеют компоненты переменной

```
var p : array [1..?] of integer                      (15.32)
```

Тогда указатель функции $p(i)$ заменяется на $p[i]$. В существующей программе придется только добавить к инструкции 1 (15.31) следующее присваивание:

$$p[ih] := ih * ih * ih \qquad (15.33)$$

Итак, мы пришли к окончательной и полной версии программы (15.34). Эта программа определяет искомое наименьшее число

$$1729 = 10^3 + 9^3 = 12^3 + 1^3$$

в результате проверки 61 кандидата. Последние значения переменных *il* и *ih* равны соответственно 10 и 12. Проверка $S[k] < S[i]$ выполняется 107 раз.

```
var  i, il, ih, min, a, b, k; integer;                          (15.34)
     j, p, S: array [1..12] of integer;
     {p[k] = k^3, S[k] = p[k] + p[j[k]] для k = 1...ih}
begin i := 1; il := 1; ih := 2;
     j[1] := 1; p[1] := 1; S[1] := 2; j[2] := 1; p[2] := 8;
     S[2] := 9;
     repeat min := S[i]; a := i; b := j[i];
        if j[i] = i then il := il + 1 else
        begin if j[i] = 1 then
                 begin ih := ih + 1; p[ih] := ih*ih*ih;
                       j[ih] := 1; S[ih] := p[ih] + 1
                 end;
              j[i] := j[i] + 1; s[i] := p[i] + p[j[i]]
        end;
        i := il; k := i;
        while k < ih do
                 begin k := k + 1;
                       if S[k] < S[i] then i := k
                 end
     until S[i] = min;
     write (min, a, b, i, j[i], eol)
end.
```

Если изменить выражение (15.33) и некоторые начальные установки переменной S, то эту же программу можно использовать для получения наименьшего числа, равного двум различным суммам пар чисел в четвертых степенях. Однако количество необходимых вычислений при этом резко возрастает. В результате получается число

$$634318657 = 134^4 + 133^4 = 158^4 + 59^4$$

после просмотра 11660 кандидатов.

15.3. Получение первых *n* простых чисел

Как и в предыдущем разделе, программа получения первых n простых чисел должна просматривать множество натуральных чисел в порядке их возрастания и выбирать те из них, которые удовлетворяют определенному критерию. В данном случае условие завершения программы проще, и поэтому мы можем сразу предло-

жить следующий вариант программы:

Вариант 1:
```
var i, x : integer;
begin x := 1;
    for i := 1 to n do
        begin x := „следующее простое число“;
        write (x)
        end
end.
```
(15.35)

В этом варианте только одна инструкция $x :=$ „следующее простое число“ требует дальнейшей детализации. И только в этой инструкции упоминается существо задачи: получение простых (а не каких-либо других) чисел. Если ввести логическую переменную *prim*, то эту инструкцию можно выразить следующим образом:

Вариант 2:
```
repeat x := x+1;
    prim := „x — простое число“
until prim
```
(15.36)

Учитывая, что все простые числа (кроме 2) нечетные, мы можем сразу уменьшить затраты на вычисления вдвое, если будем x увеличивать с шагом 2. Само число 2 будем рассматривать, как особый случай. Теперь мы должны заняться детализацией инструкции

$$prim := \text{„}x\text{— простое число“}$$

Поэтому нам придется вспомнить определение простого числа. Число x называется простым тогда и только тогда, когда оно делится без остатка на 1 и на само себя, т. е. деление его на 2, 3, ..., x—1 всегда дает ненулевой остаток. Эта последовательная проверка требует еще одного итеративного процесса, и в результате получается структура вложенных циклов:

Вариант 3:
```
repeat x := x+2; k := 2;
    repeat {x не делится на 2, 3 ... k}
        k := k+1; prim := „x не делится на k“
    until ¬ prim ∨ (k ⩾ lim)
until prim
```
(15.37)

В качестве ограничения для k, очевидно, можно взять $lim=k-1$. Но выгоднее использовать $lim=\sqrt{x}$. Действительно, если x делится на некоторое $k>\sqrt{x}$, то x можно представить в виде $x=k*j$, и, следовательно, x также делится на j, причем $j<\sqrt{x}$.

Для дальнейшей разработки алгоритма и повышения его эффективности большое значение имеет тот факт, что можно удовлетвориться проверкой делимости x только на простые числа. В самом деле, если число x делится на непростое число k, то оно также делится на простые множители числа k. Поэтому имеет смысл все найденные простые числа сохранять в таблице p, где p_k есть k-е

простое число. Принимая во внимание только что высказанные соображения, получим

Вариант 4:
```
repeat x := x+2; k := 2; prim := true;
    while prim ∧ (k < lim) do                    (15.38)
    begin prim := „x не делится на p[k]“
        k := k+1
    end
until prim;
p[i] := x
```

Теперь мы должны определить значение *lim* как индекс наибольшего простого числа, которое участвует в проверке делимости x, т. е. так, что

$$p[lim] > \sqrt{x} \text{ и } p[lim-1] \leqslant \sqrt{x} \qquad (15.39)$$

До сих пор мы полагали, что значения $p_1, \ldots, p_{lim}$ уже известны, т. е. уже вычислены. Но это верно только в том случае, если проверяемое число x всегда меньше чем p^2_{lim}, т. е. всегда выполняется неравенство

$$p[i] < p[i-1]^2 \qquad (15.40)$$

К счастью, в теории чисел доказано, что это отношение справедливо для всех простых чисел. Заметим, что при каждом увеличении x значение *lim* должно определяться заново и увеличиваться всякий раз, когда $p^2_{lim} \leqslant x$. При этом к *lim* достаточно прибавить 1, поскольку по сравнению с предыдущей проверкой значение x увеличивается только на 2, тогда как $p^2_{i+1} > p^2_i + 2$ для любого i. Эти соображения приводят к варианту 5, представляющему полную программу, в которой учтены все отмеченные ранее факты.

Вариант 5:
```
type index = 1..n;
var x: integer;
    i, k, lim: index; prim: Boolean;
    p: array [index] of integer; {p[i] = i-е простое
    число}
begin p [1] := 2; write(2); x := 1; lim := 1;
    for i := 2 to n do
    begin
        repeat x := x+2;
            if sqr(p[lim]) ⩽ x then lim := lim+1;
            k := 2; prim := true;                (15.41)
            while prim ∧ (k < lim) do
                begin prim := „x не делится на p[k]“
                    k := k+1
                end
        until prim;
        p[i] := x; write(x)
    end
end.
```

Теперь необходимо подвергнуть дальнейшей детализации инструкцию

$$prim := \text{„}x \text{ не делится на } p[k]\text{“}$$

С помощью операторов, введенных в разд. 8.1, эту инструкцию можно записать одним из следующих способов:

$$prim := (x \textbf{ mod } p[k]) \neq 0 \qquad (15.42.1)$$

или

$$prim := (x \textbf{ div } p[k]) * p[k] \neq k \qquad (15.42.2)$$

На этом можно было бы и закончить конструирование программы нахождения простых чисел. Допустим, однако, что мы хотим написать программу, в которой не используется явный оператор деления. В этом случае мы вынуждены продолжить процесс детализации. Очевидно, деление можно заменить последовательностью вычитаний и написать еще одну циклическую инструкцию:

$$\begin{array}{l} r := x; \\ \textbf{repeat } r := r - p[k] \textbf{ until } r \leqslant 0; \\ prim := r < 0 \end{array} \qquad (15.43)$$

Но эта инструкция будет выполняться весьма часто, и, следовательно, процесс повторных вычитаний окажется довольно дорогостоящим. По-видимому, целесообразно поискать более экономное решение. Одно из приемлемых и в то же время простых решений требует организации таблицы не только для простых чисел $p_1, \ldots, p_{lim}$, но и для их произведений $V_k = m * p_k$, таких, что

$$x \leqslant V[k] < x + p[k] \quad \text{для } k = 2, \ldots, lim \qquad (15.44)$$

В этом случае делимость x на p_k можно проверить простым сравнением x с V_k. Принимая во внимание частое использование значения p_{lim}^2, мы введем вспомогательную переменную, которую назовем *square*:

$$square = p[lim]^2 \qquad (15.45)$$

В итоге получится следующий окончательный вариант программы [1]:

---

[1]) Dijkstra E. W., Structured programming, EWD249, T. H. Eindhoven, 1969. [См. также его статью «Заметки по структурному программированию» в книге У. Дал, Э. Дейкстра, К. Хоор, Структурное программирование, «Мир», М., 1975.— *Прим. ред.*]

Вариант 6:

```
type index = 1..n;
var x, square: integer;
    i, k, lim: index; prim: Boolean;
    p: array [index] of integer;
    V: array [1..⌊√n⌋] of integer;
begin p[1] := 2; write(2); x := 1; lim := 1;
      square := 4;
    for i := 2 to n do
    begin
        repeat x := x+2;
            if square ≤ x then                          (15.46)
                begin V[lim] := square;
                    lim := lim+1;
                    square := sqr(p[lim])
                end;
            k := 2; prim := true;
            while prim ∧ (k < lim) do
                begin if V[k] < x then
                        V[k] := V[k]+p[k];
                    prim := (x ≠ V[k]);
                    k := k+1
                end
        until prim;
        p[i] := x; write(x)
    end
end.
```

Этот пример ясно показывает, что программисту, который вынужден пользоваться более простым инструментом (например, машиной без встроенной операции деления), приходится искать и в конце концов находить решение более высокого качества. И нет ничего удивительного в том, что наличие очень мощных вычислительных машин с большой памятью расхолаживает программистов и ни в коей мере не стимулирует разработку более совершенного и экономичного варианта алгоритма.

Ниже мы приводим таблицу, показывающую частоту выполнения четырех разных инструкций программы (15.46) в зависимости от n — количества искомых простых чисел.

$n=$	10	20	50	500	1000
$x := x+2$	14	114	611	1785	3959
$lim := lim+1$	3	6	11	17	23
$prim := (x \neq V[k])$	13	268	2340	9099	25133
$V[k] := V[k]+p[k]$	8	156	1151	3848	9287

(15.47)

15.4. Эвристический алгоритм

Программа, рассматриваемая в этом разделе, является простым и типичным примером класса алгоритмов, в которых поиск решения ведется не прямо, а эвристически, т. е. путем проб и ошибок. Эвристический метод состоит в том, что по заданному правилу шаг за шагом генерируются возможные решения (кандидаты), каждое из которых затем подвергается проверке на соответствие критериям, характеризующим решение. Если предполагаемое решение оказывается неприемлемым, то генерируется другой кандидат; причем в этом случае возможно несколько предыдущих шагов придется аннулировать. Мы будем конструировать алгоритм для решения такой задачи:

Построить последовательность из N литер в алфавите, состоящем из трех элементов (например, 1, 2, 3), такую, что никакие две соседние подпоследовательности не совпадают друг с другом.

Например, последовательность длины $N=5$ с литерами «12321» приемлема, но последовательности «12323» и «12123» неприемлемы.

В задачах такого рода разумно строить последовательность из N литер постепенно, начиная с пустой последовательности и удлиняя ее на одну литеру на каждом шаге. Нет смысла продолжать последовательность, которая не удовлетворяет критерию. По этой причине на каждом шаге необходимо проверять последовательность и добавлять к ней новую литеру только тогда, когда удовлетворяется критерий; в противном случае последовательность необходимо изменить. В первом варианте программы мы введем две переменные, указывающие длину и качество построенной к данному моменту последовательности. Первая переменная типа *integer* обозначается как *m*. Вторую можно определить либо как переменную, принимающую два значения: *good* (хорошая) и *bad* (плохая), либо как переменную типа *Boolean* и дать ей соответствующее имя. Вот эти две возможности:

var *q* : (*good*, *bad*)

и

var *good* : *Boolean*

Мы выберем вторую возможность и будем считать, что

good = *true* : последовательность удовлетворяет критерию
good = *false*: последовательность не удовлетворяет критерию.

Теперь сформулируем первый вариант программы.

Вариант 1: **var** m:0..N; *good*:*Boolean*; S:*Sequence*; (15.48)
begin $m := 0$; *good* := *true*; {пустая последовательность считается хорошей}
repeat if *good* **then** „продолжить последовательность S“
else „изменить последовательность S“
good := „S — хорошая последовательность“
until *good* $\wedge$ ($m = N$);
print (S)
end.

Операция *изменить* означает замену или удаление некоторых компонент, но не увеличение длины последовательности. Чтобы гарантировать окончание процесса, необходимо так вносить изменения, чтобы никакая последовательность, однажды отмеченная как плохая, не генерировалась повторно. Это означает, что изменения должны подчиняться некоторому правилу и что среди возможных кандидатов существует некое упорядочение, которое соблюдается в процессе их генерации.

Выбор наименований для трех литер (1, 2, 3) уже подсказывает возможное упорядочение, а именно: если последовательность $S = s_1 s_2 s_3 \ldots$ рассматривать как десятичную дробь со значением $|S| = 0{,}s_1 s_2 s_3 \ldots$, то отношение порядка $\prec$ определяется как

$$S \prec S' \leftrightarrow |S| < |S'| \qquad (15.49)$$

Теперь алгоритмы изменения и продолжения последовательности по существу уже определены. При продолжении последовательности выбирается наименьший из возможных кандидатов, чтобы при последующих изменениях не был пропущен ни один из кандидатов. Если предположить, что ‘1’ $\prec$ ‘2’ $\prec$ ‘3’, то при каждом продолжении последовательности должна добавляться ‘1’.

Чтобы подробнее описать эти две операции, пользуясь инструкциями нашего языка программирования, необходимо более точно определить структуру переменной S. Поскольку компоненты этой переменной непрерывно проверяются и даже изменяются, то файловую структуру приходится отвергнуть как неприемлемую. Поэтому остановимся на таком описании:

var S : **array** [1..N] **of** *char* (15.50)

При детализации операции *изменить* необходимо помнить, что компоненту $S[m]$ нельзя слепо заменять следующим большим значением. Если $S[m]$ = ‘3’, то следующего большего значения (преемника) нет. Это как раз тот случай, когда последовательность нужно укоротить. Однако и предпоследняя компонента опять-таки может

иметь значение '3'. Действие по укорачиванию последовательности и вообще отказ от предполагаемого решения называется *возвратом* (или *отступлением*) [1]). Мы предлагаем читателю самому попытаться с помощью изложенного алгоритма генерировать кандидатов. Ниже перечислены первые десять кандидатов, причем те, которые приемлемы как решения, помечены знаком +.

```
+1
  11
+12
+121
  1211                    (15.51)
  1212
+1213
+12131
  121311
+121312
  ...
```

Если операции продолжения *(extend)*, изменения *(change)* и проверки *(check)* оформить в виде процедур, то мы получим следующую программу:

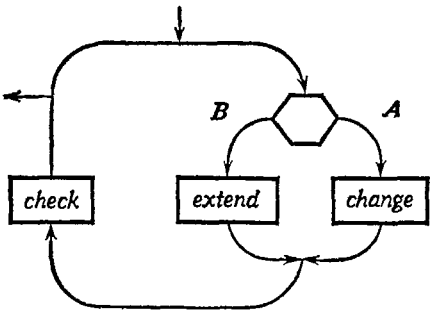
Вариант 2:

```
var S: array [1..N] of char;
    m:0..N; good: Boolean;                                  (15.52)
procedure extend;
    begin m := m+1; S[m] := '1' end;
procedure change;
    begin if S[m]<'3' then S[m] :=
          succ (S[m]) else
          begin m := m-1; {укоротить S}
                if m>0 then
                if S[m]<'3' then S[m] :=
                succ (S[m]) else
                begin m := m-1; {пустая
                последовательность}
                      if m>0 then
                      S[m] := succ (S[m])
                end
          end
    end;
begin m := 0; good := true;
    repeat if good then extend else change;
      check
    until good ∧ (m=N) ∨ (m=0);
    print (S)
end.
```

[1]) В оригинале backtracking.— *Прим. ред.*

В этой версии программы учитывается возможность получения после укорачивания S последовательности нулевой длины, поскольку в условие окончания цикла включается проверка $m=0$. Кроме того, заслуживает внимание то, что процедура *change* не может укоротить последовательность S более чем на 2 компоненты. Это обусловлено тем, что новые кандидаты всегда получаются добавлением одной литеры к *хорошей* последовательности, в которой не могут стоять подряд две одинаковые литеры.

Прежде чем продолжить процесс детализации программы, рассмотрим модификацию текущего варианта, которая будет слегка эффективнее. Прежде всего заметим, что операции *extend*, *change* и *check* всегда чередуются в соответствии со схемой (15.53).



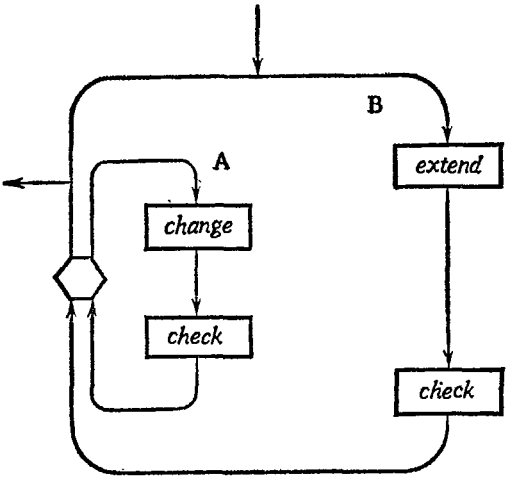
(15.53)

Легко видеть, что путь A будет выбираться значительно чаще, чем путь B. В варианте 2′ благодаря использованию вложенных инструкций несколько упрощается условие окончания цикла, в котором повторяется операция *change*.

Вариант 2′: **repeat** *extend*; *check*;
while $\neg$ *good* $\wedge$ $(m > 0)$ **do**
begin *change*; *check* **end**
until $(m = N) \vee (m = 0)$ (15.54)

Эта модификация варианта 2 соответствует схеме (15.55), в которой последовательность выполнения операций та же самая, что и на схеме (15.53).

Если посмотреть вариант 2 в целом, то прежде всего настораживает сложность операции *change* и условий окончания циклов. Что касается первого, то *change* можно упростить, прибегнув к одному довольно известному «приему». Операция *change* сложна потому, что в ней учитывается ситуация, которая бывает чрезвычайно редко (и бывает ли вообще?), а именно когда S сводится к пустой последовательности. В этом случае необходимо предотвратить присваивание несуществующему элементу $S[0]$ «Прием» заключается в том, что вводится дополнительный элемент $S[0]$, кото-



(15.55)

рому, вообще говоря, можно присвоить значение. Тогда процедуру *change* можно написать проще.

```
procedure change;
    begin while S[m]='3' do m := m—1;
        S[m] := succ(S[m])                    (15.56)
    end
```

Однако если известно (скажем, из комбинаторного анализа), что искомая последовательность длины N заведомо существует, то (15.56) можно использовать и без дополнительного элемента $S[0]$, а главную программу можно еще более упростить:

```
repeat extend; check;
    while ¬ good do                           (15.57)
        begin change; check end
until m = N
```

Теперь на нашем языке программирования программу можно сформулировать всю целиком, исключая операцию *check* [см. (15.48)], которая определялась как

good := «S — хорошая последовательность»

Примечателен тот факт, что вплоть до этого момента нам ни разу не потребовался критерий, характеризующий решение. (Единственное, что использовалось — это свойство, по которому решение нельзя получить, продолжая неприемлемые последовательности.) Но это означает, что разрабатываемая программа все еще обладает *значительной общностью*.

В поставленной задаче процедура *check* должна определять, существуют ли в S совпадающие соседние последовательности. Длина сравниваемых подпоследовательностей меняется от 1 до $m/2$. Так как для сравнения двух последовательностей длины L необходимо сравнить L отдельных литер, суммарное количество элементарных сравнений не превышает

$$N(m) = (m-1)*1 + (m-3)*2 + \ldots + 3*(m/2-1) + 1*(m/2) = \\ = \frac{1}{24}(m^3 + 3*m^2 + 2*m) \qquad (15.58)$$

для четного m и

$$N(m) = (m-1)*1 + (m-3)*2 + \ldots + 4*\left(\frac{m-3}{2}\right) + 2*\left(\frac{m-1}{2}\right) = \\ = \frac{1}{24}(m^3 + 3*m^2 - m - 3) \qquad (15.59)$$

для нечетного m. Для больших m число $N(m)$, очевидно, растет как третья степень m, и поэтому ставится под сомнение полезность такой программы. Однако при ближайшем рассмотрении оказывается, что ее эффективность можно значительно повысить. Каждый новый кандидат S получается добавлением одного элемента к последовательности, о которой известно, что она *хорошая*. Следовательно, достаточно сравнивать только те соседние подпоследовательности, в которые входит последняя добавленная литера, т. е. все пары

$$\langle S_{m-2L+1} \ldots S_{m-L},\ S_{m-L+1} \ldots S_m \rangle \qquad (15.60)$$

для $L = 1, \ldots, m/2$. Максимальное число элементарных сравнений при этом уменьшается до

$$N(m) = 1 + 2 + \ldots + \frac{m}{2} = \frac{1}{8}(m^2 + 2m) \qquad (15.61)$$

Учитывая эти соображения, процедуру *check* можно записать в следующем виде:

```
procedure check;
    var L: integer;
begin good := true;                                          (15.62)
    for L := 1 to (m div 2) do
    good := good ∧ ((S_{m-2L+1} ... S_{m-L}) ≠ (S_{m-L+1} ... S_m))
end
```

Программа станет эффективнее, если учесть, что сравнения можно прекратить, как только обнаружились две равные подпоследовательности. (Заметим, что, вообще говоря, схема программы (9.2) не может приниматься без изменений, если рекуррентное соотношение $v_i = f(v_{i-1})$ представляется в виде $v_i = (v_{i-1} \wedge q_i)$.) В улуч-

шенном варианте процедуры (15.63) число повторений заранее не известно, поэтому инструкция **for** заменяется на **while**.

```
procedure check;                                                    (15.63)
    var L, mhalf : integer;
begin good := true; L := 0; mhalf := m div 2;
    while good ∧ (L < mhalf) do
    begin L := L+1;
        good := (S_{m-2L+1} ... S_{m-L}) ≠ (S_{m-L+1} ... S_m)
    end
end
```

Так как сравниваемые подпоследовательности содержат по крайней мере по одной литере, мы можем воспользоваться инструкцией **repeat** и сравнение последовательностей литер представить в виде последовательности сравнений литер. В окончательном варианте процедуры *check* (15.64) все операции сформулированы в терминах принятого нами языка программирования.

```
procedure check;                                                    (15.64)
    var i, L, mhalf: integer;
begin good := true; L := 0; mhalf := m div 2;
    while good ∧ (L < mhalf) do
    begin L := L+1; i := 0;
        repeat good := S[m—i] ≠ S[m—L—i]; i := i+1
        until ¬ good ∨ (i = L)
    end
end
```

Теперь поставленная задача решена полностью, как это показано в (15.52), (15.56), (15.57) и (15.64).

В заключение мы еще раз прибегнем к задаче получения неповторяющихся подпоследовательностей, чтобы проиллюстрировать ситуацию, с которой довольно часто приходится встречаться на практике, когда первоначальная задача либо расширяется, либо слегка модифицируется. При этом существующая программа *адаптируется*, чтобы учесть новые изменившиеся условия.

В нашем случае расширенную задачу можно сформулировать хотя бы так:

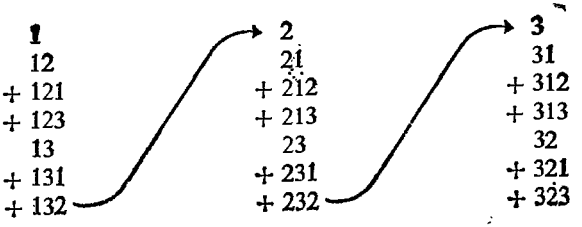
Вместо одной произвольной последовательности длины N найти все последовательности длины N, не содержащие одинаковых соседних подпоследовательностей.

К счастью, составленная нами программа имеет такую структуру, что многие ее отдельные части можно не менять. Обычно, чем четче и рациональнее структуризована программа, тем легче адаптировать ее для решения несколько измененных задач, поскольку проще выделить компоненты, которые нужно подвергнуть модификации. Мы с выгодой употребляем не только четкую декомпозицию про-

граммы, но и систематический способ генерации кандидатов. Следующие соображения позволяют сразу же получить ответ на поставленную задачу [см. (15.52)].

1. Если m достигает значения N, то S рассматривается как один из результатов и печатается. После чего необходимо изменить (но не продолжить) последовательность.
2. Условие завершения алгоритма можно упростить, поскольку отношение $m=N$ теперь излишне, и можно обойтись только проверкой условия $m=0$.

Мы настоятельно рекомендуем читателю самому убедиться в том, что предложенный алгоритм генерирует только приемлемые последовательности, а также в том, что он генерирует все возможные решения. В табл. (15.65) перечислены генерируемые кандидаты для случая $N=3$; решения помечены знаком +.



(15.65)

Из этой таблицы видно, что среди 12 решений есть несколько подобных друг другу в том смысле, что одно из другого можно получить циклической перестановкой основных литер.

(15.66)

В действительности есть только два существенно разных решения: «123» и «121». В программе, которая генерирует одного представителя из каждой группы, включающей шесть подобных решений, процесс генерации должен прекращаться, как только предпринимается попытка изменить $S[2]$. (Поэтому, разумеется, нет никакой необходимости в дополнительной фиктивной компоненте $S[0]$.) Итак, мы получаем окончательное решение более общей

задачи в виде полной программы.

```
var S: array [1..N] of char;
    m: integer; good: Boolean;
procedure extend;                                        (15.67)
    begin m := m+1; S[m] := '1' end;
procedure change;
    begin {см. (15.56)} end;
procedure check;
    begin {см. (15.64)} end;
procedure print;
    var i: integer;
    begin for i := 1 to N do write(S[i]);
          write(eol)
    end;
begin m := 2; S[1] := '1'; S[2] := '2'; good := true;
    repeat if good then
               if m=N then begin print; change end
               else extend
           else change;
           check
    until m=2
end.
```

В табл. (15.68) приводятся сведения о количестве решений K в зависимости от длины последовательности N. Учитывается тот факт, что существенно различных решений в 6 раз меньше.

N	$K(N)$	N	$K(N)$	N	$K(N)$
3	2	9	18	15	103
4	3	10	24	16	133
5	5	11	34	17	174
6	7	12	44	18	232
7	10	13	57	19	305
8	13	14	76	20	398

(15.68)

УПРАЖНЕНИЯ

15.1 Программу решения линейных уравнений (15.17) можно упростить, если переменную B включить в переменную A в виде дополнительного столбца, т. е.

$$A=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{pmatrix}$$

Напишите соответствующую программу, в которой учитываются все упрощения. (Заметим, что в обеих программах используется один и тот же алгоритм.)

15.2 Дополните программу решения системы линейных уравнений (15.17), включив в нее поиск *осевого элемента*.

Вариант 1: На k-м шаге исключения неизвестных в качестве осевого выбирается элемент $a_{hk}^{(k)}$ с наибольшим абсолютным значением среди элементов k-го столбца (осевого столбца), т. е.

$$|a_{hk}^{(k)}| \geqslant |a_{ik}^{(k)}| \quad \text{для} \quad i=k, \ldots, n$$

Указание: После того как осевой элемент $a_{hk}^{(k)}$ найден, поменяйте местами строки $a_h^{(k)}$ и $a_k^{(k)}$, а также элементы $b_h^{(k)}$ и $b_k^{(k)}$.

Вариант 2: На k-м шаге исключения неизвестных в качестве осевого элемента выбирается $a_{hm}^{(k)}$ с наибольшим абсолютным значением среди элементов k-й строки и k-го столбца (*тотальная осевая операция*), т. е

$$|a_{hm}^{(k)}| \geqslant |a_{ij}^{(k)}| \quad \text{для} \quad i, j=k, \ldots, n$$

Указание: После того как осевой элемент $a_{hm}^{(k)}$ найден, поменяйте местами строки $a_h^{(k)}$ и $a_k^{(k)}$ (а также $b_h^{(k)}$ и $b_k^{(k)}$). Затем поменяйте местами столбцы $a_{x_m}^{(k)}$ и $a_{x_k}^{(k)}$ и зафиксируйте перестановку столбцов $\langle m, k\rangle$, поскольку позднее, на этапе обратной подстановки, потребуется перестановка неизвестных x_m и x_k. Наконец, если нет $a_{ij}^{(k)}$, по абсолютной величине превосходящего ε (пусть, например, $\varepsilon=10^{-8}$), то система считается плохо обусловленной, и для этого случая нужно предусмотреть переход на некоторую предварительно определенную инструкцию [см. (14.24)].

15.3 Дополните программу решения системы линейных уравнений (15.17), включив в нее следующие операции.

Вариант 1: На k-м шаге исключения неизвестных все компоненты строки $a_i^{(k)}$ умножаются на один и тот же *масштабный множитель* s_i^k $(i=k, \ldots, n)$. Заметим, что значения неизвестных при этом не меняются, но точность вычисленных корней x_j может улучшиться. Выберем следующие масштабные множители:

$$s_i^{(k)} = 1 \Big/ \sum_{j=k}^{n} a_{ij}^{(k)} \quad \text{для} \quad i=k, \ldots, n$$

Вариант 2: На k-м шаге исключения неизвестных все элементы j-го столбца умножаются на один и тот же *масштабный множитель* $s_j^{(k)}$ $(j=k, \ldots, n)$. Полученные результаты x_j на этапе обратной подстановки необходимо соответствующим образом изменить. Выберем

$$s^{(k)}{}_j = 1 \Big/ \sum_{i=k}^{n} a_{ij}^{(k)} \quad \text{для} \quad j=k, \ldots, n$$

15.4 Напишите программу решения следующей системы n линейных уравнений с коэффициентами a_{ij} и b_i

$$a_{11} * x_1 + a_{12} * x_2 = b_1$$
$$a_{k,\,k-1} * x_{k-1} + a_{k,k} * x_k + a_{k,k+1} * x_{k+1} = b_k \quad \text{для } k=2, \ldots, n-1$$
$$a_{n,\,n-1} * x_{n-1} + a_{n,\,n} * x_n = b_n$$

Если из коэффициентов a_{ij} образовать матрицу, то она окажется *трехдиагональной*

$$\begin{pmatrix} a_{11} & a_{12} & & & & & 0 \\ a_{21} & a_{22} & a_{23} & & & & \\ & a_{32} & a_{33} & a_{34} & & & \\ & & \cdot & \cdot & \cdot & & \\ 0 & & & \cdot & \cdot & \cdot & \\ & & & & a_{n,n-1} & a_{n,n} & \end{pmatrix}$$

Указание: На k-м шаге исключения неизвестных необходимо вычислять только следующие два коэффициента:

$$a_{k+1,k+1}^{(k+1)} = a_{k+1,k+1}^{(k)} - (a_{k+1,k}^{(k)} * a_{k,k+1}^{(k)})/a_{kk}^{(k)}$$

и

$$b_{k+1}^{(k+1)} = b_{k+1}^{(k)} - (a_{k+1,k}^{(k)} * b^{(k)})/a_{kk}^{(k)}$$

Все остальные коэффициенты не меняются. Трехдиагональная матрица представляется массивом

var *A*:**array** [1..*n*, —1..1] **of** *real*

где коэффициенты $a_{ij}^{(k)}$ обозначаются как $A[i, j-i]$. В этом случае нулевые коэффициенты не занимают памяти.

15.5 Напишите программу решения следующей системы n линейных уравнений:

$$\sum_{j=1}^{i} a_{ij} * x_j = b_i \quad \text{для } i=1, \ldots, n$$

В этом случае ненулевые коэффициенты a_{ij} образуют треугольную матрицу

$$A = \begin{pmatrix} a_{11} & & & & 0 \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \cdot & \cdot & \cdot & \cdot & \\ a_{n1} & \cdots & & a_{nn} & \end{pmatrix}$$

Заметим, что алгоритм Гаусса можно значительно упростить, а в памяти можно хранить только $\frac{1}{2}n(n+1)$ коэффициентов. Матрицу a_{ij}, следовательно, можно представить в виде массива

$$A : \textbf{array } [1..m] \textbf{ of } real \qquad m = \frac{1}{2}n(n+1)$$

а коэффициенты a_{ij} обозначить как $A[i * (i-1) \textbf{ div } 2+j]$.

15.6 Измените программу (15.34) так, чтобы она генерировала не одно, а 10 наименьших чисел, каждое из которых равняется двум разным суммам пар натуральных чисел, возведенных в третью степень. Из десяти пар сумм $\langle a_i, b_i \rangle$ выберите такие, у которых слагаемые линейно независимы, т. е. не

должно быть множителя n, такого, что

$$a_i = n * a_j \quad \text{и} \quad b_i = n * b_j \qquad i \neq j$$

15.7 Рассмотрите следующую программу для задачи из разд. 15.2:

```
var i, j, m, n, x: integer; p: array [0..13] of integer;
begin m := 0; p[0] := 0;
   while m < 13 do
   begin m := m+1; p[m] := m*m*m; n := 0;                    (15.69)
     while n < m do
     begin n := n+1; x := p[m]+p[n]; i := m-1;
           {x—следующий кандидат}
       while 2*p[i] > x do
       begin j := i-1;
         while p[i]+p[j] > x do j := j-1;
         if p[i]+p[j]=x then go to 99 else i := i-1
       end
     end
   end;
99:write (x, m, n, i, j)
end.
```

Выясните историю ее разработки и определите условия верификации, которые положены в основу этой программы.

Программа получает правильный результат 1729. Однако не исключается возможность, что автор воспользовался условием, которое трудно доказать и которое он считал истинным. (Что это за условие?) Таким образом, это пример получения правильного результата на основе неправильных умозаключений.

Сравните количество вычислений в этой программе и в программе (15.34), если обе программы модифицировать для случая четвертых степеней ($x^4 = a^4+b^4=c^4+d^4$).

15.8 Напишите программу, вычисляющую 10 наименьших чисел x_i, третья степень которых представляется как сумма кубов трех натуральных чисел

$$x_i^3 = a_i^3 + b_i^3 + c_i^3 \quad \text{для} \quad i=1, \ \ldots, \ 10$$

Слагаемые должны быть линейно независимы, т. е. не должно существовать множителя n, такого, что для $i \neq j$, $a_i = n * a_j$, $b_i = n * b_i$ и $c_i = n * c_j$.

15.9 Исследуйте аналитически (т. е. без помощи вычислительной машины) последствия замены условия

$$\textbf{if } square \leqslant x \textbf{ then} \qquad (15.70)$$

на

$$\textbf{if } square < x \textbf{ then}$$

в программе (15.46).

15.10 Если к изменению (15.70) в программе (15.46) добавить замену условия

$$\textbf{while } n < lim \textbf{ do}$$

на

$$\textbf{while } n \leqslant lim \textbf{ do} \qquad (15.71)$$

то программа станет неверной. Какой из инвариантов при этом нарушается? Если оставить эти два изменения, то каким образом можно легко исправить программу? Сравните эффективность полученной программы и программы (15.46)?

15.11 Вывод

$$x \neq a \leftrightarrow \text{«}x \text{ не делится на } p\text{»}$$

в программе (15.46) справедлив только тогда, когда выполняются следующие два условия:

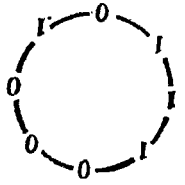
$$x \leqslant a < x + p \text{ и } a = n * p$$

Проверьте, вставив соответствующие утверждения, что эти условия являются инвариантами в следующем фрагменте программы, взятом из (15.46), если p — натуральное число, большее 2:

```
x := 1; a := 0;
repeat x := x+2;
    if a < x then a := a+p;                    (15.72)
    {x ≤ a < x+p, a = n * p}
    until P
```

15.12 Напишите программу для генерации в возрастающем порядке первых 100 чисел множества M, которое определяется следующим образом:
(a) число 1 принадлежит M;
(b) если x принадлежит M, то $y = 2 * x + 1$ и $z = 3 * x + 1$ также принадлежат M;
(c) никакие другие числа не принадлежат M. (M = {1, 3, 4, 7, 9, 10, ...}.)

15.13 На рис. (15.73) изображено кольцо, содержащее 2^3 нулей и единиц. В этом кольце каждая из 2^3 возможных подпоследовательностей, состоящих из 3-х двоичных цифр, встречается ровно один раз.



(15.73)

Постройте алгоритм, генерирующий аналогичное «кольцо» из 2^n цифр, в котором каждая подпоследовательность из n цифр встречается только один раз. Придерживайтесь принципа пошаговой детализации программы.

15.14 [1]) Задана матрица отношений R размера $(n \times n)$. Постройте алгоритм, вычисляющий натуральные числа x_i и y_i $(i = 1, \ldots, n)$, такие, что

(a) $x_i < y_j$, если R_{ij} «меньше»,
(b) $x_i = y_j$, если R_{ij} «равно»,
(c) $x_i > y_j$, если R_{ij} «больше».

Если таких чисел не существует, то логической переменной q следует присвоить значение *false*.

---

[1]) По существу это фрагмент задачи построения функций предшествования. Более полную и подробную постановку задачи можно найти в книге Д. Гриса, Конструирование компиляторов для цифровых вычислительных машин, «Мир», М., 1975.— *Прим. ред.*

ПРИЛОЖЕНИЕ А.
ЯЗЫК ПРОГРАММИРОВАНИЯ ПАСКАЛЬ[1])

Основные символы

A...*Z*, *a*...*z*	буквы
0 1 2 3 4 5 6 7 8 9	цифры
+ — * /**div mod**	арифметические операторы
∨ ∧ ¬	логические операторы
=≠<≤≥> **in**	операторы отношения
( )	скобки
[ ]	индексные скобки
{ }	скобки для комментария
begin end	инструктивные скобки
: =	оператор присваивания
'	апостроф
. , : ;	разделители
↑	символ указателя
if then else case of with while do repeat until for to	разделители инструкций
const type var procedure function	спецификаторы класса объекта
array file record set	спецификаторы класса структуры
nil	пустой указатель
goto label	оператор перехода, описатель метки

Стандартные идентификаторы

Константы:	*false*, *true*	(8.1)
	eol (≡'**eol**')	(9.3)
Типы:	*Boolean*, *integer*, *char*, *real*, *text*	(8.1—8.4)
Переменные:	*input*, *output*	(10.4)

[1]) Пересмотренная версия по сравнению с определением языка в *Acta Informatica*, (1971), 35—63.

Функции:	*abs*, *sqr*, *odd*	
	succ, *pred*	(8)
	ord, *chr*	(8.3)
	trunc, *eof*	(8.4,10.3)
	sin, *cos*, *exp*, *ln*, *sqrt*, *arctan*	
Процедуры:	*get*, *put*, *reset*, *rewrite*	(10.2—10.3)
	read, *write*	(10.4)

Операторы

Операторы отношения (имеют наименьший приоритет):

$= \neq$	операнды произвольные, результат *Boolean*	(8.3)
$< \leqslant \geqslant >$	операнды-скаляры, результат *Boolean*	

Аддитивные операторы:

$+ -$	сложение, вычитание	(8.2, 8.4)
$\vee$	логическое объединение (OR)	(8.1)

Мультипликативные операторы:

*	умножение	(8.2, 8.4)
/	деление, результат *real*	(8.4)
div	деление, результат *integer*	(8.2)
mod	остаток от целочисленного деления	(8.2)
$\wedge$	логическая дизъюнкция (AND)	(8.1)

Одноместный оператор:

$\neg$	логическое отрицание (NOT)	(8.1)

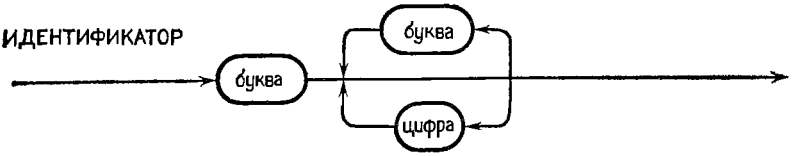
Стандартное представление программ на Паскале при ограниченном наборе литер в коде ASCII

1. Используются только заглавные буквы.
2. Пробелы внутри идентификаторов, слов-разделителей и чисел не допускаются.
3. Основные символы, представленные английскими словами (так называемые *слова-разделители*), не обрамляются никакими специальными литерами, играющими роль авторегистров. Никакое слово-разделитель нельзя использовать в качестве идентификатора, и если перед ним или после него встречается другое слово-разделитель или идентификатор, то между ними должен быть по крайней мере один пробел.
4. Символы языка Паскаль, которых нет в ограниченном наборе ASCII, представляются следующим образом:

Символ в языке Паскаль	Соответствующее представление литерами
$\vee \quad \wedge \quad \neg$	OR AND NOT
$\neq \quad \leqslant \quad \geqslant$	<> <= >=
$\uparrow$	@
{ }	/* */

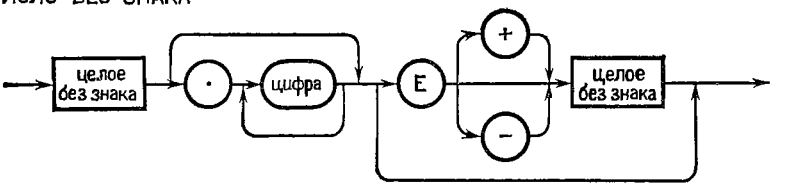
СИНТАКСИС

ИДЕНТИФИКАТОР

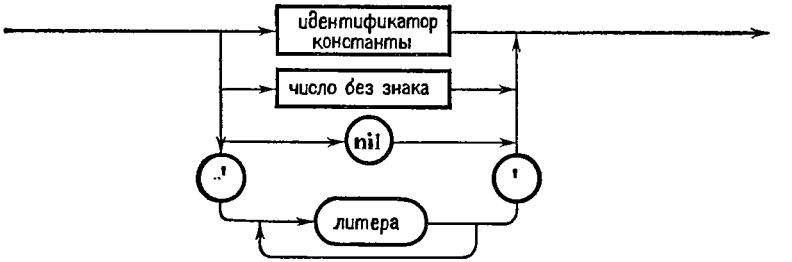


ЦЕЛОЕ БЕЗ ЗНАКА

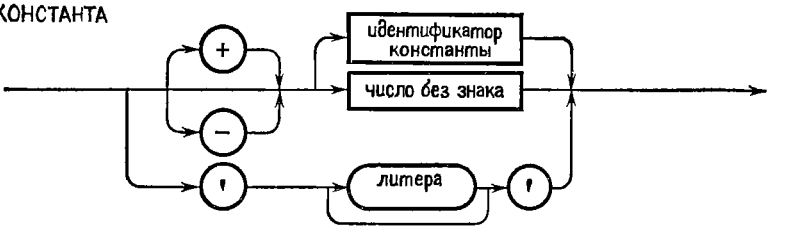
ЧИСЛО БЕЗ ЗНАКА



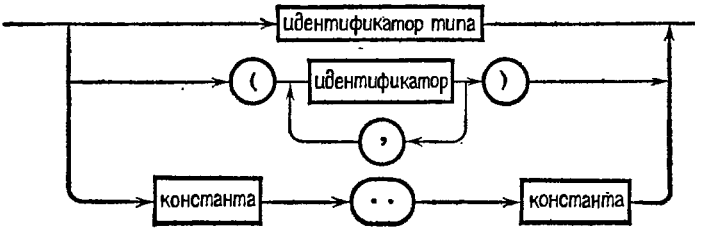
КОНСТАНТА БЕЗ ЗНАКА



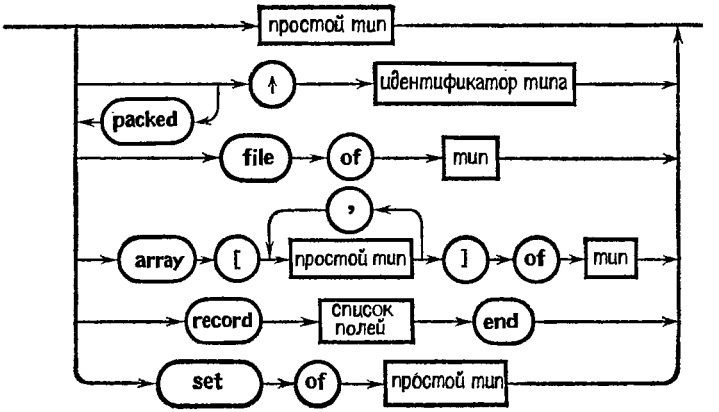
КОНСТАНТА



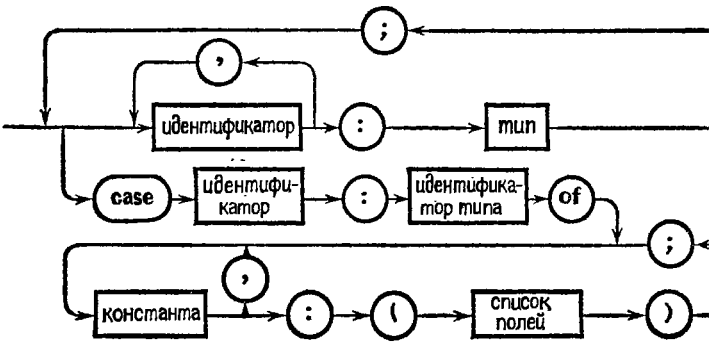
ПРОСТОЙ ТИП



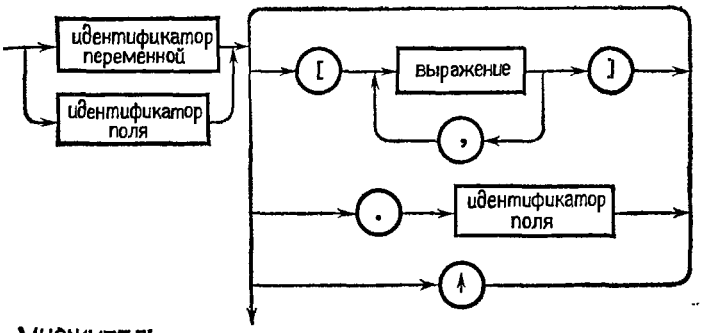
ТИП



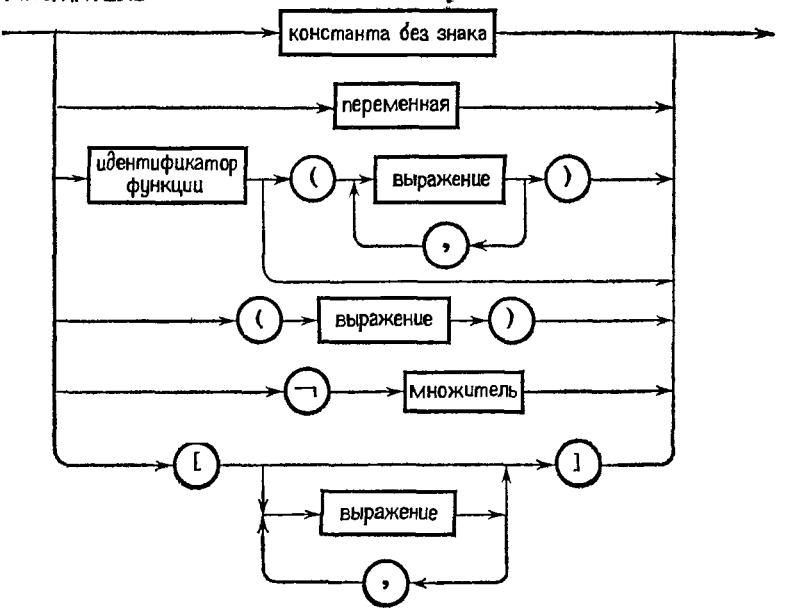
СПИСОК ПОЛЕЙ



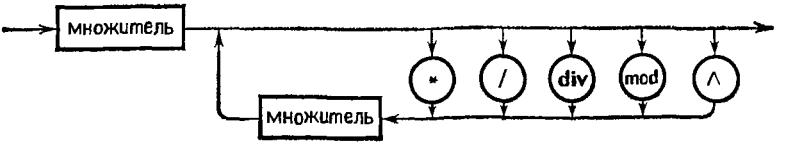
ПЕРЕМЕННАЯ



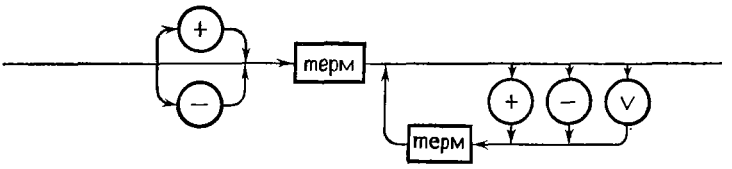
МНОЖИТЕЛЬ



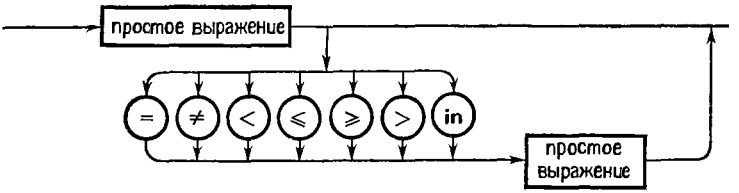
ТЕРМ



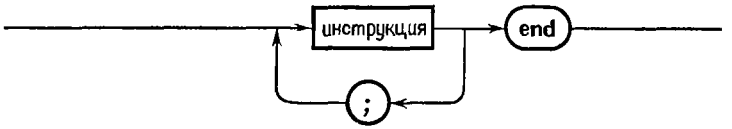
ПРОСТОЕ ВЫРАЖЕНИЕ



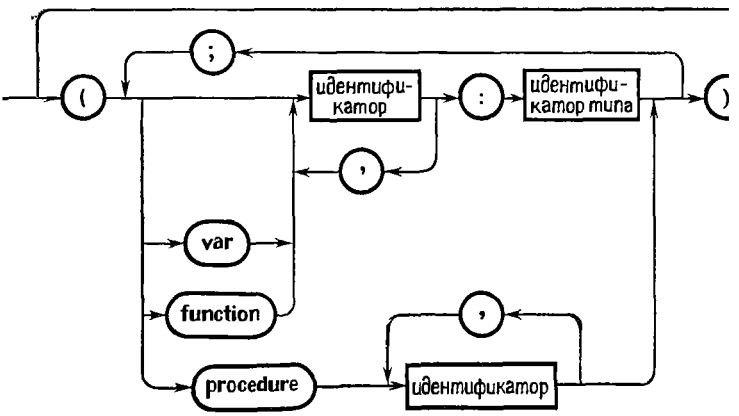
ВЫРАЖЕНИЕ



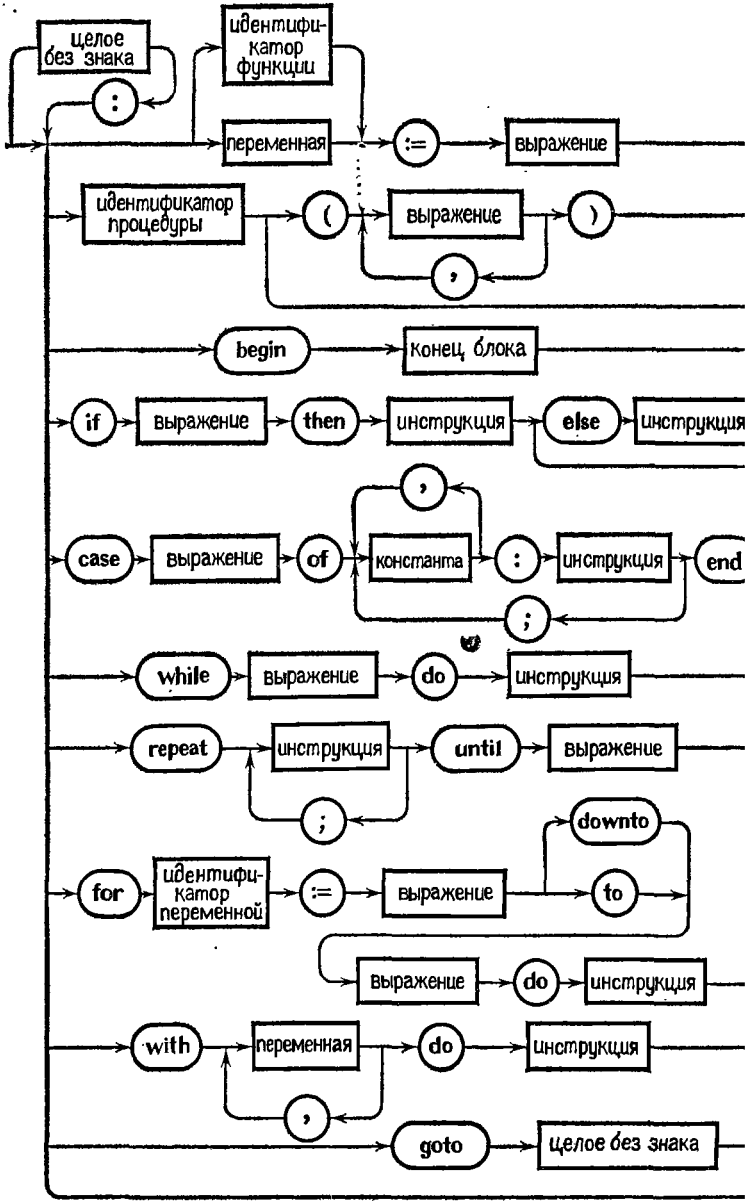
КОНЕЦ БЛОКА



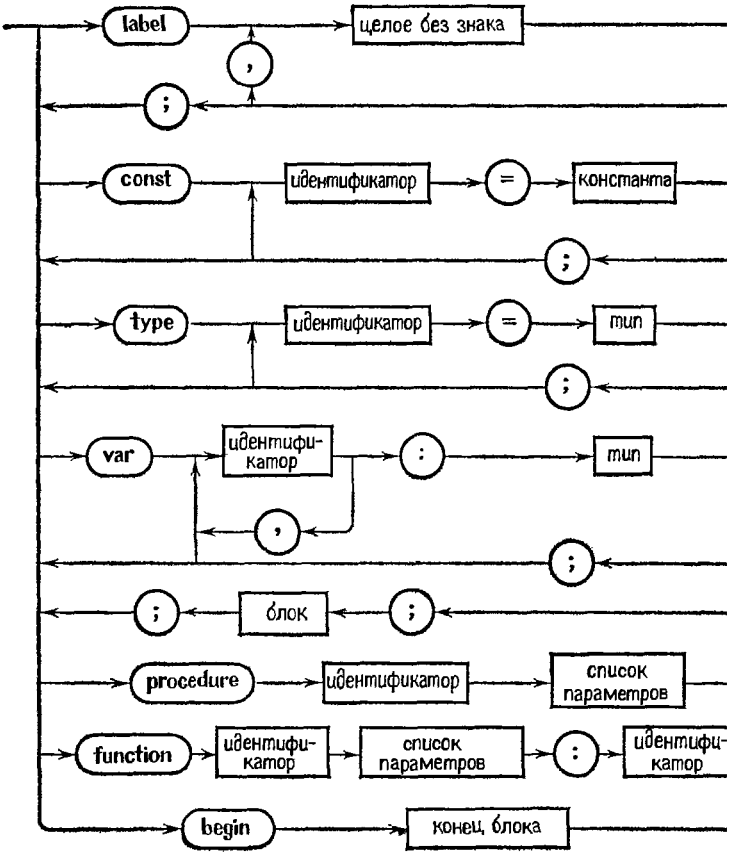
СПИСОК ПАРАМЕТРОВ



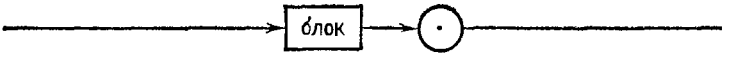
ИНСТРУКЦИЯ


целое без знака
:
идентификатор функции
переменная
:=
выражение
идентификатор процедуры
(
выражение
)
,
begin
конец блока
if
выражение
then
инструкция
else
инструкция
case
выражение
of
константа
:
инструкция
end
;
while
выражение
do
инструкция
repeat
инструкция
until
выражение
for
идентификатор переменной
:=
выражение
downto
to
выражение
do
инструкция
with
переменная
do
инструкция
goto
целое без знака

БЛОК



ПРОГРАММА



ПРИЛОЖЕНИЕ В.
ЛИТЕРЫ В КОДЕ ASCII

b_6	0	0	0	0	1	1	1	1
b_5	0	0	1	1	0	0	1	1
b_4	0	1	0	1	0	1	0	1
b_3–b_0								
0000	nul	dle		0	@	P		p
0001	soh	dcl	!	1	A	Q	a	q
0010	stx	dc2	"	2	B	R	b	r
0011	etx	dc3	#	3	C	S	c	s
0100	eot	dc4	$	4	D	T	d	t
0101	enq	nak	%	5	E	U	e	u
0110	ack	syn	&	6	F	V	f	v
0111	bel	etb	'	7	G	W	g	w
1000	bs	can	(	8	H	X	h	x
1001	ht	em	)	9	I	Y	i	y
1010	lf	sub	*	:	J	Z	j	z
1011	vt	esc	+	;	K	[[1]	k	{[1]
1100	ff	fs	,	<	L	\	l	|
1101	cr	qs	-	=	M	]	m	}
1110	so	rs	.	>	N	^ `	n	¬
1111	si	us	/	?	O	_	o	del

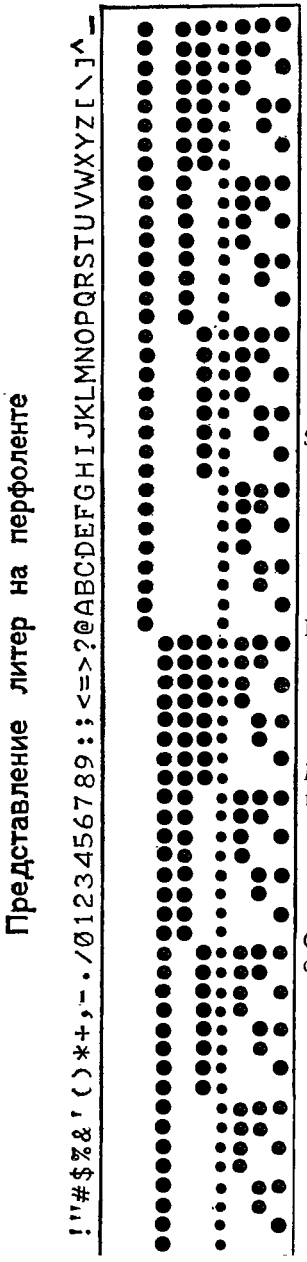
управляющие литеры

графические литеры

Ограниченный набор литер в коде ASCII

[1]) Эти символы в стандарте ISO не определены. В различных национальных версиях они различаются. Здесь приводится версия ASCI1.

Смысл управляющих литер для передачи даиных



Layout characters:	Литеры, управляющие расположением материала:
bs backspace	вш возврат на шаг
ht horizontal tabulation	гт горизонтальная табуляция
lf line feed	пс перевод строки
vt vertical tabulation	вт вертикальная табуляция
ff form feed	пф перевод формата
cr carriage return	вк возврат каретки
Ignore characters:	Литеры отмены:
nul null characters	пус пусто
can cancel	ан аннулирование
sub substitute	зм замена
del delete	зб забой
Separator characters:	Разделители информации:
fs file separator	рф разделитель файлов
gs group separator	рг разделитель групп
rs record separator	рз разделитель записей
us unit separator	рэ разделитель элементов
Escape characters:	Регистровые литеры:
so shift-out	лат латинский регистр
si shift-in	рус русский регистр
esc escape	пре префикс (авторегистр)
Medium control characters:	Литеры управления носителем:
bel ring bell	зв звонок
dc1-dc4 device control	су1—су4 символ управления
em end of medium	кн конец носителя
Communication control characters:	Литеры связи:
soh start of heading	нз начало заголовка
stx start of text	нт начало текста
etx end of text	кт конец текста
eot end of transmission	кп конец передачи
enq enquiry	ктм кто там?
ack acknowledgment	да подтверждение
nak negative acknowledgment	нет отрицание
dle data link escape	ар1 авторегистр 1
syn synchronous idle	син синхронизация
etb end of transmission block	кб конец блока

УКАЗАТЕЛЬ

СОДЕРЖАНИЕ