

22.18 /
396
БИБЛИОТЕЧКА
ПРОГРАММИСТА

Н.И. ВЬЮКОВА
В.А. ГАЛАТЕНКО
А.Б. ХОДУЛЕВ

Систематический подход к программированию



18
96

БИБЛИОТЕЧКА ПРОГРАММИСТА

Н. И. ВЬЮКОВА
В. А. ГАЛАТЕНКО
А. Б. ХОДУЛЕВ

СИСТЕМАТИЧЕСКИЙ ПОДХОД К ПРОГРАММИРОВАНИЮ

Под редакцией
Ю. М. БАЯКОВСКОГО



МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1988

ББК 22.18

В 96

УДК 519.6

Вьюкова Н. И., Галатенко В. А., Ходулев А. Б.
Систематический подход к программированию/Под ред. Ю. М. Бая-
ковского.— М.: Наука. Гл. ред. физ.-мат. лит., 1988.— 208 с.—
(Библиотечка программиста). ISBN 5-02-013771-5

Содержит систематическое введение в программирование. Глав-
ная цель — научить методам разработки программ. Основные ком-
поненты систематического подхода — математическое исследование
задачи, пошаговая детализация и обоснование правильности про-
грамм, анализ алгоритмов. Для записи программ используется язык
паскаль, получивший широкое признание как в педагогической
практике, так и в профессиональной деятельности.

Для начинающих программистов, студентов вузов и преподава-
телей школ.

Табл. 4. Ил. 1. Библиогр. 17 назв.

Рецензент

член-корреспондент АН СССР Л. Н. Королев

АБОНЕМЕНТ



БИБЛИОТЕКА №

Инд. №

Г. Самарканд

В 1702070000—104 11-88
053(02)-88

ISBN 5-02-013771-5

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

429636

ОГЛАВЛЕНИЕ

Предисловие редактора	5
1. Элементы теории алгоритмов	7
1.1. Понятие алгоритма	7
1.2. Машина Поста	9
1.3. Предложение Поста	12
1.4. Алгоритмически неразрешимые проблемы	15
2. Основные элементы языка паскаль	19
2.1. Паскаль-машина	19
2.2. Паскаль-программа	20
2.3. Инструкции присваивания	21
2.4. Инструкции ввода-вывода	23
2.5. Условные инструкции	27
2.6. Комментарии	29
2.7. Инструкции цикла	30
2.8. Обработка последовательностей	32
2.9. Тип INTEGER	37
2.10. Об инвариантах циклов	38
2.11. Определение констант	41
2.12. Определение новых типов. Отрезочные типы	42
2.13. Массивы	43
2.14. Тип BOOLEAN	46
2.15. Процедуры и функции	48
3. Методы и приемы программирования	56
3.1. Пошаговая разработка программ	56
3.2. Анализ алгоритмов	63
3.3. Обработка рекуррентных последовательностей	72
3.4. Упрощение циклов	77
3.5. Тип CHAR	79
3.6. Понижение размерности массивов	81
3.7. Несколько примеров программ	85
3.8. Тестирование и отладка программ	94
3.9. Переборные задачи	100
4. Более сложные элементы языка паскаль	111
4.1. Типы данных, заданные перечислением	111
4.2. Инструкции выбора	113
4.3. Множества	114
4.4. Упакованные структуры данных	115

4.5. Файловая структура данных	117
4.6. Записи	122
4.7. Точность машинных вычислений	129
4.8. Динамические переменные и указатели	134
4.9. Бинарные деревья	144
4.10. Рекурсивные процедуры	152
4.11. Процедуры в качестве параметров	159
Задачи для самостоятельного решения	163
Заключение	170
Приложение 1. Синтаксические диаграммы языка паскаль	175
Приложение 2. Пример программы	181
Приложение 3. Работа с трансляторами	194
Терминологический словарь	200
Список литературы	203
Предметный указатель	205

ПРЕДИСЛОВИЕ РЕДАКТОРА

Интерес к информатике, компьютерам, программированию велик и многообразен. Электронные игры, системы продажи железнодорожных и авиационных билетов не требуют от человека, взаимодействующего с ЭВМ, специальных знаний в информатике. Сталкиваясь всего лишь с внешними проявлениями сложных систем, достаточно приобрести некоторые навыки работы с клавиатурой и экраном дисплея, освоить азы компьютерной грамотности. Специалисты, разрабатывающие системы, должны позаботиться, чтобы диалог с ЭВМ был удобным для человека, а поведение системы, как говорят, «дружелюбным». Тем же, кому необходимо решать с помощью ЭВМ нестандартные задачи, кому приходится создавать сложные системы, нужны глубокие специальные знания.

Содержащийся в этой книге учебный курс программирования следует рассматривать как начальный этап на пути к профессиональному овладению программированием. Профессиональные знания и навыки в программировании полезны многим и в первую очередь будущим математикам, физикам, инженерам (и, конечно же, программистам).

Программирование — конструктивная дисциплина, и поэтому неизбежно предполагается использование ЭВМ в процессе обучения. Коль скоро такой объект не абстрактный объект, как программа, создан, он должен быть проверен, и, более того, он должен быть употреблен для получения практических результатов.

Книга начинается с краткого изложения теоретических основ программирования (гл. 1): вводится понятие алгоритма, рассматривается его уточнение (машина Поста) и основные идеи доказательства алгоритмической неразрешимости проблем.

Курс построен таким образом, чтобы как можно быстрее ввести повятит необходимые для написания интересных и достаточно содержательных программ. Требуемые для этого сведения о языке паскаль излагаются в гл. 2. После ознакомления с основными возможностями паскаль-машины и изучения раздела «Обработка последовательностей» можно браться за решение разнообразных задач вычисления сумм, произведений, минимумов и т. п.

Главная цель курса — научить методам систематической разработки программ. Этой теме посвящена гл. 3. Основные компоненты систематического подхода — математическое исследование задачи, анализ алгоритмов, пошаговая разработка программ, обоснование правильности разрабатываемых программ. При таком подходе от языка программирования, используемого для преподавания, требуется, чтобы он отражал основные понятия программирования в наиболее естественной форме. Именно поэтому выбран язык пас-

каль. Впрочем, большую часть излагаемого материала можно проиллюстрировать и с помощью других языков, например фортрана или бейсика, если акцентировать внимание на принципиальной стороне дела, а не на частных особенностях языка. Стоит отметить тот факт, что знание языка паскаль помогает быстро освоить другой язык. В гл. 3 также разбираются приемы программирования, полезные при решении определенных классов задач. После изучения гл. 3 можно приступить к созданию программы в несколько сотен строк. Возможные темы таких заданий приведены в конце книги.

ЭВМ в настоящее время используются преимущественно не для манипуляций с числами, а в таких областях, как моделирование, аналитические преобразования, обработка текстов, анализ и конструирование изображений и т. п. Материал, необходимый для решения таких нечисленных задач, излагается в гл. 4.

В приложениях содержится справочная информация (синтаксические диаграммы языка паскаль) и приводится пример достаточно большой программы.

1. ЭЛЕМЕНТЫ ТЕОРИИ АЛГОРИТМОВ

1.1. Понятие алгоритма

Алгоритм — одно из основных понятий математики, которое невозможно строго определить, а можно лишь пояснить с помощью других слов. Близкими по смыслу являются слова «рецепт», «метод», «программа» и др. Несколько упрощая формулировку, приведенную в [2], можно сказать, что алгоритм — это точное предписание, которое задает вычислительный процесс, начинающийся с произвольного исходного данного (из некоторой совокупности возможных для этого алгоритма исходных данных) и направленный на получение полностью определяемого этим исходным данным результата.

Одним из древнейших и известнейших является алгоритм Евклида для нахождения наибольшего общего делителя двух натуральных чисел. Приведем одну из возможных формулировок этого алгоритма.

1. Присвоить переменным X и Y значения, наибольший общий делитель которых ищется.

2. Если $X > Y$, перейти на шаг 5.

3. Если $X < Y$, перейти на шаг 6.

4. (Здесь $X = Y$.) Выдать X в качестве результата. Конец работы.

5. Заменить пару (X, Y) парой $(X - Y, Y)$ и вернуться на шаг 2.

6. Заменить пару (X, Y) , парой $(X, Y - X)$ и вернуться на шаг 2.

Давайте проследим, как будет развиваться вычислительный процесс, задаваемый алгоритмом Евклида. Будем считать, что каждый шаг алгоритма выполняется за единицу времени. Составим таблицу (табл. 1.1), показывающую, как с течением времени после выполнения очередного шага меняются значения X и Y . В качестве исходного данного возьмем пару чисел (100, 80).

Обычно требуется, чтобы алгоритмы обладали следующими пятью свойствами:

Работа алгоритма Евклида

Момент времени	Шаг алгоритма	X	Y	$X > Y?$	$X < Y?$
1	1	100	80		
2	2	100	80	Да	
3	5	20	80		
4	2	20	80	Нет	
5	3	20	80		Да
6	6	20	60		
7	2	20	60	Нет	
8	3	20	60		Да
9	6	20	40		
10	2	20	40	Нет	
11	3	20	40		Да
12	6	20	20		
13	2	20	20	Нет	
14	3	20	20		Нет
15	4	Конец работы. Результат: 20			

1. **Конечность.** Работа алгоритма должна заканчиваться за конечное число шагов. Алгоритм Евклида обладает этим свойством, поскольку максимумы из X и Y образуют убывающую последовательность натуральных чисел, а такая последовательность не может быть бесконечной.

2. **Определенность.** Все предписания алгоритма должны допускать однозначную трактовку и быть понятны тому, кто будет выполнять алгоритм, — исполнителю. Понятие исполнителя очень важно. Свойства исполнителя решающим образом влияют на алгоритм, призванный дать решение той или иной задачи. Например, если в микрокалькуляторе отсутствует блок вычисления элементарных функций, задача вычисления синуса угла становится гораздо более сложной, чем при наличии такого блока. Предписания, понятные одному исполнителю, могут оказаться непонятными для другого. Поэтому, составляя алгоритм, нужно помнить об исполнителе, для которого алгоритм предназначается. Алгоритм Евклида предназначен для исполнителя, умеющего сравнивать и вычитать натуральные числа.

3. **Ввод.** Алгоритм должен обладать свойством массовости, т. е. давать решение целой группы задач, отличающихся исходными данными. Алгоритм Евклида позволяет найти наибольший общий делитель любой пары натуральных чисел, являющихся исходными данными.

4. Вывод. Алгоритм должен давать некоторый результат (результаты). В случае алгоритма Евклида это наибольший общий делитель.

5. Эффективность. Все шаги алгоритма должны быть такими, чтобы исполнитель мог выполнить их за конечное время. Кроме того, общее время работы алгоритма должно не просто быть конечным, но и лежать в некоторых разумных пределах. Например, хотя методом полного перебора вариантов можно выяснить, чем кончится шахматная партия при сильнейшей игре обеих сторон, однако время работы этого метода делает его практически бесполезным.

В следующем разделе изучается внешне простой, но методологически очень важный исполнитель.

1.2. Машина Поста

Свою конструкцию американский математик Эмил Пост предложил в 1936 г. для формализации нестрогого понятия алгоритма. Читателям, желающим подробнее ознакомиться с машинами Поста, мы рекомендуем работу [15]. Правда, избранный нами способ изложения несколько отличается от принятого в [15].

Машина Поста состоит из *бесконечной ленты* и *головки*. Лента разделена на клетки. В каждой клетке может быть записан один символ из алфавита, фиксированного для каждой данной машины Поста. В любой момент времени головка обзора ровно одну клетку и может работать только с ней.

Машины Поста работают под управлением программ. Программы состоят из команд. Запись команд для машин Поста имеет три поля: номер команды, операцию и отсылку. Команды нумеруются, начиная с 1. Выполнение программы начинается с команды номер 1. Отсылка показывает, какую команду нужно выполнять после данной. Что касается видов операций, то их бывает пять:

—> — движение головки на одну клетку вправо;

<— — движение головки на одну клетку влево;

S (где S — произвольный символ из алфавита машины Поста) — запись символа S в обозреваемую клетку, при этом прежнее содержание клетки пропадает;

СТОП — завершение работы машины Поста, в поле отсылки не указывается;

$$? \begin{cases} S_1 M_1 \\ S_2 M_2 \\ \dots \\ S_n M_n \end{cases}$$

— команда разветвления. Здесь $S_1, S_2, \dots$ — некоторые символы из алфавита машины Поста, $M_1, M_2, \dots$ — отсылки. Все символы

S_i должны быть различными, однако не требуется, чтобы в команде разветвления был употреблен весь алфавит. Эта команда выполняется следующим образом. Анализируется символ из обозреваемой клетки. Если он совпадает с S_i , следующей будет выполняться команда с номером M_i . При этом головка никуда не перемещается и на ленту ничего не записывается. Если в клетке стоит символ, не упомянутый в данной команде, происходит так называемая безрезультатная остановка машины Поста («машина ломается»).

Другие возможные исходы выполнения программы на машине Поста — бесконечная работа и так называемая результативная остановка, которая вызывается выполнением команды СТОП. Последний исход является нормальным; именно так и должна заканчиваться работа машины Поста.

Приведем несколько примеров программ для машин Поста. Пусть алфавит машины Поста состоит из двух символов: I и пустого символа (пробела). Программа

1. $\rightarrow 2$

2. I 1

Момент времени	Номер команды	Состояние машины
0		
1	1	
2	2	
3	1	
4	2	
5	1	
6	2	
...		

(Символ $\wedge$ обозначает положение головки.)

состоит всего из двух команд, однако за счет того, что отсылка во второй команде равна 1, обе команды будут выполняться многократно, а сама программа сделает очень многое — она будет заполнять ленту символами I справа от клетки, которую головка обозревала в начальный момент. Работа этой программы никогда не завершится.

Давайте проследим, как будет меняться состояние машины Поста. Предположим, что на выполнение каждой команды требуется единица времени.

Составив соответствующие программы, машину Поста можно научить арифметическим операциям. Договоримся, что числа на ленте будем записывать в единичной системе счисления. Например, четверка запишется как IIII. Программа

1. <-2
2. I 3
3. СТОП

увеличит на единицу записанное на ленте число в предположении, что в начальный момент головка обозревает левую единицу числа. Предположим теперь, что головка обозревает произвольную единицу числа. Напишем программу прибавления единицы в этом случае:

1. <-2
2. $\begin{cases} \text{I} & 1 \\ & 3 \end{cases}$
3. I 4
4. СТОП

Идея этой программы проста: головка смещается влево, пока не найдет первую пустую клетку, а затем записывает в нее символ I.

В предыдущих примерах предполагалось, что алфавит машины Поста содержит лишь пробел и символ I. Пусть теперь в алфавит входит еще и символ 0. Решим следующую задачу. На ленте записана последовательность символов I и 0. Головка обозревает самый левый элемент последовательности. Требуется заменить все символы I на 0, а 0 на I.

1. $\begin{cases} \text{I} & 2 \\ & 0 & 3 \\ & & 5 \end{cases}$
2. 0 4
3. I 4
4. $->1$
5. СТОП

Проследите за работой программы, изображая состояние ленты и положение головки после выполнения очередного шага.

Наконец, рассмотрим следующую задачу. Сдвинуть число, записанное в единичной системе счисления, на одну позицию влево. В начальный момент головка обозревает левую единицу числа.

1. $< - 2$
2. I 3
3. $- > 4$
4. $\left. \begin{array}{l} I \\ ? \end{array} \right\} \begin{array}{l} 3 \\ 5 \end{array}$
5. $< - 6$
6. 7
7. СТОП

1.3. Предложение Поста

Машины Поста были предложены в качестве формализации нестрогого понятия алгоритма, рассмотренного в разд. 1.1. Формализация понятия алгоритма позволяет строго доказывать теоремы о несуществовании алгоритмов для решения каких-либо задач (теоремы об алгоритмической неразрешимости). Пост предположил, что любой алгоритм может быть записан в виде программы для машины Поста (это утверждение называется предложением Поста). Сделаем несколько оговорок.

Алгоритм понимается здесь в обобщенном смысле: от алгоритма не требуется конечности (разд. 1.1, свойство 1). Требование конечности, очень важное в практических приложениях, оказывается неудобным для теории алгоритмов. Таким образом, работа алгоритма над некоторыми исходными данными может никогда не закончиться. В этом случае будем говорить, что алгоритм неприменим к этим исходным данным. Алгоритм может быть неприменим и по другим причинам, например в случае машины Поста по причине безрезультатной остановки. Результат применения алгоритма P к исходным данным X будем обозначать $P(X)$; будем считать, что это выражение не определено, если P неприменим к X . Определим понятие эквивалентности двух алгоритмов. Алгоритм P называется эквивалентным алгоритму Q , если для любых исходных данных X выполняется равенство $P(X) = Q(X)$. Здесь (и в дальнейших аналогичных выражениях) равенство означает, что либо оба выражения не определены, либо оба определены и равны.

Исходные данные для машины Поста задаются состоянием ленты и положением головки. Чтобы избавиться от проблемы поиска исходных данных на ленте, договоримся, что в начальный момент головка обозревает самый левый непустой символ исходных данных (если он есть). Кроме того, потребуем, чтобы среди символов

исходных данных не было пробелов. Тогда исходные данные для машины Поста можно будет записывать в виде конечной последовательности непустых символов из некоторого конечного набора — входного алфавита машины. Конечные последовательности символов из алфавита (включая пустую последовательность) называются словами в этом алфавите. Так, допустимым начальным состоянием может быть

$$\begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & \\ \hline \end{array}$$

^

что записывается с помощью слова 100 в алфавите 1, 0. Наложив те же требования на заключительное состояние ленты (после результирующей остановки машины), мы можем сказать, что рассматриваемые нами машины Поста перерабатывают слова в некоторые другие слова.

Теперь сформулируем предложение Поста более строго.

Предложение Поста. Для всякого алгоритма, исходными данными и результатами которого являются слова, существует эквивалентная ему программа для машины Поста.

Будем называть алгоритмы, эквивалентные программам для машины Поста, **постовыми алгоритмами**. Тогда предложение Поста кратко формулируется так: все алгоритмы, работающие со словами, **постовы**.

Заметим, что ограничение лишь словами на самом деле несколько нас не стесняет: все математические объекты, с которыми могут работать алгоритмы, можно закодировать в виде слов. Мы уже изображали в виде слов натуральные числа. Нетрудно придумать способ изображения произвольных целых и рациональных чисел. Так, число минус две трети можно записать, как слово —II/III. Для изображения последовательностей и матриц следует ввести какие-либо символы-разделители строк и выписывать все элементы подряд, разделяя их этими символами. Сложнее обстоит дело с изображением вещественных чисел. Строгое изучение алгоритмов над произвольными вещественными числами составляет предмет конструктивного математического анализа — довольно обширной ветви теории алгоритмов. Заметим, однако, что применяемые на практике алгоритмы вычислений с вещественными числами работают всегда с приближенными числами, представленными, например, конечными десятичными дробями, т. е. рациональными числами.

Предложение Поста не может быть доказано математически, поскольку оно включает нестрогое понятие алгоритма. Оно может быть лишь подтверждено большим количеством примеров постовых алгоритмов, отсутствием опровергающих примеров и некоторыми общими рассуждениями, т. е. примерно так же, как устанавливаются законы природы в естественных науках. Такими подтвержда-

ющими примерами являются, например, программы, составленные при изучении машины Поста.

Еще важнее то, что известные общие приемы построения новых алгоритмов из уже имеющихся не способны вывести нас за пределы класса постовых алгоритмов. Это означает, что, если мы возьмем некоторое количество постовых алгоритмов и начнем строить из них новые посредством применения (возможно, неоднократного) определенных приемов, мы сможем получить огромное количество новых алгоритмов, но все они будут заведомо постовыми. Ниже рассмотрены наиболее употребительные способы построения новых алгоритмов.

1. **Композиция алгоритмов.** Пусть даны алгоритмы A и B . Их композиция P , обозначаемая как $B \circ A$, определяется предписанием: применить алгоритм A и к его результату применить алгоритм B . Таким образом, результат алгоритма P записывается формулой

$$P(X) = B(A(X)).$$

Легко показать, как построить требуемую программу, если A и B выражены программами для машины Поста. Для этого нужно приписать программу B к программе A , увеличив в программе B все номера команд и отсылки на длину программы A , и заменить в программе A все отсылки на команду СТОП отсылками на первую команду программы B .

2. **Разветвление.** Даны три алгоритма — A , B и C . Их разветвление Q работает следующим образом. Сначала к исходным данным X применяется алгоритм A . Если его результатом окажется пустое слово (слово, не содержащее букв, т. е. пустая лента), то к исходным данным следует применить алгоритм B , в противном случае к исходным данным следует применить алгоритм C . Разумеется, если A неприменим к X , то и разветвление следует считать неприменимым к X . Таким образом,

$$\begin{aligned} \text{если } A(X) = \langle \text{пустое слово} \rangle, \text{ то } Q(X) &= B(X), \\ \text{если } A(X) = \langle \text{непустое слово} \rangle, \text{ то } Q(X) &= C(X). \end{aligned}$$

3. **Повторение.** Здесь снова объединяются два алгоритма A и B . Повторение P определяется формулой

$$\begin{aligned} \text{если } A(X) = \langle \text{пустое слово} \rangle, \text{ то } P(X) &= X, \text{ иначе} \\ \text{если } A(B(X)) = \langle \text{пустое слово} \rangle, \text{ то } P(X) &= B(X), \text{ иначе} \\ \text{если } A(B(B(X))) = \langle \text{пустое слово} \rangle, \text{ то } P(X) &= B(B(X)), \text{ иначе} \end{aligned}$$

Реализующий эту формулу алгоритм предписывает многократно применять алгоритм B до тех пор, пока алгоритм A не скажет, что пора кончать (выработав пустое слово).

Доказательства реализуемости на машине Поста разветвления и повторения более сложны, чем в случае композиции, и мы не будем их рассматривать.

С помощью указанных способов можно сконструировать весьма сложные программы, не выписывая их явно. Одним из показательных примеров является так называемая универсальная программа U . Она реализует следующий алгоритм. Взять заданную произвольную программу для машины Поста и применить ее к заданному слову. Исходными данными для универсальной программы должна быть запись программы в паре с исходным словом. Записать программу в виде слова не составляет труда; запись программы P будем обозначать $\tilde{P}$. Тогда исходные данные для U могут иметь, например, вид $\tilde{P} * X$, где предполагается, что знак $*$ не используется в программе P . Теперь можно определить работу универсальной программы следующей формулой:

$$U(\tilde{P} * X) = P(X)$$

Существование такой программы U может быть строго доказано.

Последним, и весьма сильным, подтверждением предложения Поста является то, что различные формализации понятия алгоритма, независимо предложенные разными авторами — Тьюрингом, Черчем, Марковым и др., оказались в точности эквивалентными.

1.4. Алгоритмически неразрешимые проблемы

Объектом изучения в теории алгоритмической разрешимости являются так называемые массовые проблемы, т. е. задачи, зависящие от какого-нибудь параметра. Так, бессмысленно спрашивать об алгоритмической разрешимости единичной проблемы вроде Великой теоремы Ферма, ответом на которую является одно слово «да» или «нет». Однако ту же теорему Ферма можно превратить в массовую проблему, если анализировать наличие корней уравнения $x^n + y^n = z^n$ при заданном значении n . Задача алгоритмической разрешимости формулируется тогда следующим образом: существует ли алгоритм, который по заданному натуральному n определяет, имеет ли уравнение $x^n + y^n = z^n$ решение в целых числах? (Ответ на этот вопрос, как и на вопрос о справедливости теоремы Ферма, неизвестен.)

Мы докажем алгоритмическую неразрешимость проблемы самоприменимости. Эта проблема заключается в том, чтобы по заданной программе P для машины Поста узнать, применима ли она к своей записи, т. е. определено ли $P(\tilde{P})$. Например, программа, заменяющая символы I на 0 , а 0 на I , применима к любому слову, в частности и к своей записи (если только мы ухитримся придумать спо-

соб записи программ в двухсимвольном алфавите, что в принципе возможно), поэтому она самоприменима, а следующая программа B

1. ? {2

2. СТОП

применима только к пустому слову и поэтому несамоприменима. (Если головка в начальный момент обозревала клетку, в которой записан не пробел, выполнение первой команды приведет к безрезультатной остановке.) Задача состоит в отыскании алгоритма, позволяющего для любой программы P определить ее самоприменимость. Мы докажем, что не существует программы для машины Поста, распознающей самоприменимость. В соответствии с принципом Поста отсюда будет следовать, что не существует вообще никакого алгоритма для распознавания самоприменимости программ для машины Поста.

Допустим, что требуемая программа существует. Она должна быть применима к записи любой программы P и перерабатывать записи несамоприменимых программ, скажем, в пустое слово, а записи самоприменимых в непустое. Обозначим искомую программу через S . Тогда

если P несамоприменима, то $S(\tilde{P}) = \langle \text{пустое слово} \rangle$,

если P самоприменима, то $S(\tilde{P}) = \langle \text{непустое слово} \rangle$.

Для наших целей нужна несколько иная программа R , которая работала бы так:

если P несамоприменима, то $R(\tilde{P}) = \langle \text{пустое слово} \rangle$,

если P самоприменима, то $R(\tilde{P})$ не определено

(R должна быть неприменима к записям самоприменимых программ). Такую программу R легко сконструировать как композицию S и приведенной выше программы B , применимой только к пустым словам: $R = B \circ S$. Требуемые свойства программы R проверяются непосредственно. Выясним теперь, самоприменима ли R , т. е. определено ли $R(\tilde{R})$. Приведенные выше формулы для результата R должны быть справедливы для любой программы P , подставляемой в качестве исходных данных, в частности для самой R . Подставляя R вместо P в эти формулы, получим

если R несамоприменима, то $R(\tilde{R}) = \langle \text{пустое слово} \rangle$,

если R самоприменима, то $R(\tilde{R})$ не определено.

Иными словами, если R несамоприменима, то $R(\tilde{R})$ определено, т. е. R самоприменима; если же R самоприменима, то $R(\tilde{R})$ не опре-

делено, т. е. R несамоприменима. Получили противоречие, которое доказывает невозможность такой программы R и, следовательно, алгоритмическую неразрешимость проблемы самоприменимости.

Мы рассуждали здесь примерно так же, как в известном парадоксе парикмахера.

В некотором городе жил парикмахер, который обязался брить всех тех и только тех жителей города, кто не бреет себя сам. Спрашивается, бреет ли этот парикмахер сам себя. Если нет, то он принадлежит к категории жителей, не бреющих себя, и поэтому должен брить себя. Противоположное предположение о том, что он бреет себя, тоже, как легко видеть, приводит к противоречию.

Решение этого парадокса несложно — принятое парикмахером обязательство невыполнимо. В случае проблемы самоприменимости программ для машины Поста мы не просто постулировали свойства программы R , мы ее сконструировали. Поэтому полученное противоречие доказывает невыполнимость нашей исходной предпосылки о существовании программы S , распознающей самоприменимость.

Проблема самоприменимости — частный случай более важной проблемы применимости: по заданной программе и исходным данным узнать, применима ли программа к данным. Эта проблема, таким образом, тоже алгоритмически неразрешима. Неразрешимость проблем самоприменимости и применимости имеет место не только для машины Поста, но вообще для любого формального описания понятия алгоритма, в частности для программ на языке паскаль.

В различных областях математики — в алгебре, теории чисел, топологии и др. — были найдены многие алгоритмически неразрешимые проблемы. Наиболее известный результат такого рода — алгоритмическая неразрешимость 10-й проблемы Гильберта, установленная в 1970 г. советским математиком Ю. В. Матиясевичем. Проблема состоит в том, чтобы по заданному произвольному уравнению вида $P(x_1, x_2, \dots, x_n) = 0$, где P — многочлен с целыми коэффициентами, установить, имеет ли оно целочисленное решение.

Задачи

1. На ленте записаны два положительных целых числа в единичной системе счисления, разделенные одним пробелом. Головка в начальный момент расположена у левого конца левого числа. Написать программу для машины Поста, которая после результативной остановки оставит на ленте запись одного числа, являющегося суммой двух исходных.

У к а з а н и е. Необходимо лишь перенести одну единицу с левого края в разделяющий числа пробел.

2. В условиях задачи 1 написать программу для вычисления абсолютной величины разности двух чисел.



3. В условиях задачи 1 написать программу для вычисления произведения двух чисел.

4. Написать программу нахождения целой части корня квадратного из числа, записанного на ленте машины Поста в единичной системе счисления.

5. Докажите алгоритмическую неразрешимость проблемы эквивалентности программ для машины Поста. Проблема состоит в том, чтобы по записи двух программ для машины Поста узнать, эквивалентны ли они.

У к а з а н и е. Сведите к этой проблеме проблему применимости.



2. ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА ПАСКАЛЬ

2.1. Паскаль-машина

Ниже мы познакомимся еще с одним исполнителем — паскаль-машиной. Паскаль-машина не существует в виде реальной машины, однако она может быть промоделирована на универсальных ЭВМ, и такие модели, называемые реализациями языка паскаль, действительно существуют на многих ЭВМ. Язык паскаль — это средство записи программ для паскаль-машины.

Паскаль-машина состоит из памяти и устройства управления. Память разделена на секции, называемые *переменными*. Каждая переменная имеет жестко закрепленное за ней имя и может принимать одно из конечного множества значений. Значения переменных могут меняться в ходе работы паскаль-машины. Переменные бывают разных *типов* в зависимости от допустимого множества значений. Пока мы познакомимся с переменными типа REAL, значениями которых являются вещественные числа. Числа хранятся в переменных с ограниченной точностью и в ограниченном диапазоне. Конкретный диапазон и точность зависят от реальной ЭВМ, на которой реализована паскаль-машина. Так, в случае ЕС ЭВМ вещественные числа принадлежат (по модулю) диапазону примерно от 10^{-78} до 10^{76} , их точность — примерно 16—17 десятичных цифр.

Устройство управления паскаль-машины выполняет программу на языке паскаль (точнее, исполнительную часть программы; см. разд. 2.2). Программа представляет собой последовательность *инструкций* (иногда говорят — операторов или команд). Устройство управления выполняет инструкции программы последовательно в том порядке, как они записаны, если только какая-либо инструкция не предписывает изменить обычный порядок выполнения. Устройство управления может изменять состояние памяти и анализировать его, кроме того, оно может общаться с внешним миром, принимая (например, от человека) какую-либо информацию или посылая ее. Для каждого из этих действий имеется свой тип инструкций.

Чтобы иметь возможность проследить за работой паскаль-машины, нужно четко представлять себе, как описывается ее состояние. Оно задается состоянием памяти и состоянием устройства управления. Состояние памяти — это значения всех переменных. Под состоянием устройства управления будем пока понимать место в программе, т. е. инструкцию, которая в данный момент выполняется или только что выполнялась.

2.2. Паскаль-программа

Программа на языке паскаль состоит из двух частей. В первой, описательной части задается необходимая нам паскаль-машина. Различные паскаль-машины различаются, в частности, набором переменных, из которых состоит их память. В описательной части перечисляются все переменные данной машины. Вторая часть, исполнительная, содержит собственно программу, т. е. инструкции, которые будут выполняться устройством управления.

Паскаль-программа начинается заголовком следующего вида:

```
PROGRAM <имя программы> (...);
```

Здесь слово PROGRAM должно записываться именно в таком виде, это одно из *ключевых слов* языка паскаль. Каждое ключевое слово исполняет в программе вполне определенную роль и не может использоваться ни для каких других целей; например, ключевые слова нельзя использовать в качестве имен.

Имя программы следует заменить на какое-либо конкретное имя, например РАЗБОР или РЕШЕНИЕ-УРАВНЕНИЯ. Условимся, что в угловые скобки < > будут заключаться названия понятий. В конкретных программах конструкцию в угловых скобках нужно заменять на конкретный текст, соответствующий этому понятию.

Имя программы, как и всякое другое имя, может содержать только буквы, цифры и знак подчеркивания (-). Имя должно начинаться с буквы. В остальном имена можно выбирать произвольно, однако нужно учитывать технические ограничения реализаций паскаль-машины.

1. Во многих реализациях не различаются прописные и строчные буквы.

2. В большинстве реализаций в именах анализируется только несколько первых символов (на БЭСМ-6 — восемь символов). Это означает, что длинные имена, имеющие длинное одинаковое начало, будут восприняты как совпадающие.

3. В некоторых реализациях в именах нельзя использовать русские буквы и знак подчеркивания.

Примеры правильных имен:

A A1 ВРЕМЯ T2P
ПРОДОЛЖИТЕЛЬНОСТЬ
ПРОДОЛЖИТЕЛЬНОСТЬ-ДВИЖЕНИЯ

(последние два имени, скорее всего, не будут различаться реальной паскаль-машиной).

Примеры неправильных имен:

9Б КТО-ТО X5.2

В скобках после имени программы указываются средства для общения с внешним миром; об этом см. разд. 2.4.

Заголовок отделяется от остальной части программы точкой с запятой.

После заголовка располагается описание переменных:

VAR <имя₁>, <имя₂>, ..., <имя_n> : <тип>;

Здесь VAR — ключевое слово, с которого начинается раздел описания переменных. Затем через запятую перечисляются имена всех используемых паскаль-машинной переменных и после двоеточия указывается их тип. Пока вместо <тип> мы будем писать REAL, что означает вещественный тип переменных. Пример описания переменных:

VAR G, ВЫСОТА, ВРЕМЯ : REAL;

Исполнительная часть программы начинается ключевым словом BEGIN и заканчивается ключевым словом END, за которым стоит точка. Между BEGIN и END записываются инструкции программы, разделяемые точками с запятой. В следующих разделах мы познакомимся с инструкциями различных видов.

2.3. Инструкции присваивания

Инструкции присваивания служат для изменения состояния памяти. Каждая инструкция присваивания предписывает изменить значение одной переменной и записывается в виде

<имя изменяемой переменной> := <выражение>

Выполнение этой инструкции приведет к изменению значения переменной, имя которой указано слева от знака присваивания :=. Справа от := записывается выражение, определяющее новое значение переменной. Это выражение строится по правилам, близким к правилам записи математических выражений. Выражение составляется из чисел и имен переменных, связанных знаками арифметических операций: + (сложение), — (вычитание), * (умножение), / (де-

ление). При вычислений выражения операции выполняются в обычном порядке — сначала умножения и деления, затем сложения и вычитания. Рядом стоящие умножения и деления выполняются слева направо, равно как и рядом стоящие сложения и вычитания. Для изменения естественного порядка операций можно воспользоваться скобками по обычным правилам. Таким образом, выражение $A*B/C*D$ означает то же, что и $(A*B/C)*D$, поскольку операции выполняются слева направо. Чтобы $C*D$ было в знаменателе, нужно написать $A*B/(C*D)$. Использование в выражении имени переменной означает, что при вычислении выражения будет использовано текущее значение этой переменной. Примеры правильных выражений:

$$1.5+2/4 \quad X+1E-5 \quad A-(B+C)/D$$

Чтобы отделить целую часть числа от дробной, вместо десятичной запятой используется точка. В записи $1E-5$ буква E означает «умножить на 10 в степени», а все число читается как «1 умножить на 10 в степени -5 ».

В выражениях можно также использовать некоторые математические функции, например $\sin$ и $\cos$. Для исключения неоднозначностей аргумент функций заключается в скобки. Так, вместо математического обозначения $\sin x$ следует писать $SIN(X)$.

Имеются следующие стандартные функции

ABS — абсолютная величина,
 SQR — возведение в квадрат,
 SQRT — квадратный корень,
 EXP — экспонента, $EXP(X)=e^X$,
 LN — натуральный логарифм,

SIN, COS, ARCTAN — тригонометрические функции (углы выражаются в радианах).

Пример выражения с использованием стандартных функций:

$$SQRT(2*N/G)$$

Это — время падения тела с высоты N в поле тяжести с ускорением G .

Важно уяснить, что инструкции присваивания — это не тождества или уравнения, как формулы в математических текстах. Любая инструкция есть некоторое предписание, которое нужно выполнить. В случае инструкции присваивания исполнителю предписывается вычислить выражение в правой части (после знака $=$), подставив вместо имен переменных их значения, и сделать полученный результат новым значением переменной, указанной слева от

знака $:=$. При этом старое значение этой переменной теряется (однако при вычислении правой части его еще можно использовать).

Пример. После выполнения последовательности инструкций

$X:=3.5; \quad Y:=2*X; \quad X:=X+1; \quad Z:=2*X$

переменные получают следующие значения:

$X=4.5 \quad Y=7 \quad Z=9$

Теперь воспользуемся полученными знаниями и составим полную программу для решения квадратного уравнения $Ax^2+Bx+C=0$. Будем считать, что уравнение имеет два вещественных корня. Первый вариант программы будет выглядеть так:

```
PROGRAM КОРНИ( );
VAR  A,B,C,D,X1,X2:REAL;
BEGIN  D:=SQRT(SQR(B)-4*A*C);
       X1:=(-B-D)/(2*A);  X2:=(-B+D)/(2*A)
END.
```

З а м е ч а н и е 1. Обратим внимание на использование вспомогательной переменной D в целях экономии длины программы. Присвоив переменной D значение корня квадратного из дискриминанта уравнения, мы избежали необходимости двукратной записи этого выражения в формулах для X1 и X2.

З а м е ч а н и е 2. Разбиение программы на строки несущественно для паскаль-машины и выбирается из соображений удобства чтения программы человеком. Можно в одной строке записывать несколько инструкций или разбивать одну инструкцию на несколько строк. Однако нельзя переносить с одной строки на другую имена и числа.

З а м е ч а н и е 3. Эта программа на самом деле неправильная, поскольку в ней не заданы (но используются) значения переменных A, B и C, которые являются исходными данными программы. В отличие от машины Поста мы не можем задавать начальное состояние памяти паскаль-машины. Чтобы программа могла получить исходные данные, необходимо воспользоваться инструкциями связи с внешним миром (инструкциями ввода-вывода).

2.4. Инструкции ввода-вывода

Инструкции ввода позволяют получить информацию из внешнего мира и поместить ее в память паскаль-машины. Программе (исполнителю) безразлично, откуда берется эта информация — от человека, от другой программы или же с автоматического измерительного прибора. Инструкции вывода позволяют передать информацию во внешний мир, скажем напечатать что-либо на печатаю-

щем устройстве. Инструкции вывода — это единственное средство сделать результаты программы доступными для обозрения человеком.

Форма записи инструкции ввода:

READ($\langle$ имя переменной $_1$ $\rangle$, ..., $\langle$ имя переменной $_n$ $\rangle$)

Выполнение этой инструкции заключается в чтении (вводе) n значений и присваивании их перечисленным в скобках переменным. Таким образом, эта инструкция работает как последовательность инструкций присваивания

$\langle$ имя переменной $_1$ $\rangle := \langle$ первое введенное значение $\rangle$;

$\langle$ имя переменной $_2$ $\rangle := \langle$ второе введенное значение $\rangle$;

...

с той только разницей, что присваиваемые значения не задаются и не вычисляются в программе, и если выполнять программу несколько раз, то эти значения могут быть разными. Если программа использует инструкции ввода, то в заголовке программы после ее имени в скобках следует записать слово INPUT.

Заметим, что сама исходная информация, которую получает паскаль-машина по инструкции ввода, содержит лишь вводимые числа; каким переменным они будут присвоены, знает только инструкция ввода.

Инструкция вывода имеет вид

WRITE($\langle$ выражение $_1$ $\rangle$, ..., $\langle$ выражение $_n$ $\rangle$)

Инструкция предписывает вычислить значения выражений и вывести (напечатать) их. Если программа использует инструкции вывода, то в заголовке программы после ее имени в скобках следует записать слово OUTPUT.

Переделаем теперь программу решения квадратных уравнений, с тем чтобы она вводила коэффициенты уравнения и печатала корни:

```
PROGRAM КОРНИ(INPUT,OUTPUT);
VAR  A,B,C,D,X1,X2:REAL;
BEGIN  READ(A,B,C);
        D:=SQRT(SQR(B)-4*A*C);
        X1:=(-B-D)/(2*A);  X2:=(-B+D)/(2*A);
        WRITE(X1,X2)
END.
```

Исходные данные к этой программе могут выглядеть, например, так:

1 -1 -1

что соответствует уравнению

$$X^2 - X - 1 = 0.$$

Эта программа уже может быть выполнена, однако она имеет существенный изъян. Подумаем, как будет выглядеть выдаваемый на печать результат работы этой программы (так называемая выдача). Только по выдаче мы можем судить о работе программы, состояние памяти паскаль-машины нам недоступно. Эта программа напечатает что-нибудь вроде

—0.61803399 1.61803399

Зачастую выдачи программ приходится рассматривать отдельно от самих программ, а по такой выдаче невозможно удостовериться в правильности программы, поскольку неизвестно, какое уравнение решается. Более того, из этой выдачи не видно, для чего вообще предназначена программа. Хорошо составленная программа должна удовлетворять следующим требованиям:

- 1) программа должна печатать исходные данные;
- 2) выдача должна быть прокомментирована так, чтобы был ясен смысл напечатанных чисел.

Для включения в выдачу комментариев необходимо уметь печатать не только числа, но и произвольные тексты. Чтобы напечатать заданный текст, следует в качестве одного из выражений в инструкции вывода записать этот текст, заключенный в кавычки. Например, инструкция `WRITE('M=', M, ' КГ')` напечатает текст `M=`, затем значение переменной `M` и затем текст `КГ`, в результате чего в выдаче появится, например, `M=5.5 КГ`. Обратите внимание на пробел в начале `' КГ'`. Он служит для того, чтобы `КГ` не печаталось вплотную за предшествующим числом.

Переделаем нашу программу с учетом этих требований:

```
PROGRAM КОРНИ(INPUT,OUTPUT);
VAR  A,B,C,D:REAL;
BEGIN  READ(A,B,C);
  WRITE(' КОЭФФИЦИЕНТЫ КВАДРАТНОГО',
  ' УРАВНЕНИЯ A=' ,A, ' B=' ,B, ' C=' ,C);
  D:=SQRT(SQR(B)—4*A*C);
  WRITE('КОРНИ X1=' ,(-B—D)/(2*A),
  ' X2=' ,(-B+D)/(2*A))
END.
```

З а м е ч а н и е 1. Инструкции вывода занимают по две строки. При переносе следует учитывать, что тексты в кавычках нельзя разрывать. Чтобы напечатать длинный текст, его следует разбить на несколько коротких кусков, записав каждый кусок в кавычках и перечислив их через запятую.

З а м е ч а н и е 2. Исходные данные следует печатать в самом начале программы, сразу же после их ввода, как сделано в при-

мере. Если отложить печать исходных данных, мы рискуем не получить вообще никакой информации о работе программы, поскольку при выполнении программы, содержащей ошибки, исполнитель может и не дойти до инструкций вывода.

З а м е ч а н и е 3. Исходные данные не есть часть программы. Типичная ситуация, когда человек, сидя за дисплеем, запускает программу и в ответ на ее запрос набирает на клавиатуре исходные данные.

З а м е ч а н и е 4. Инструкция вывода результата в этой программе записана несколько иначе, чем в предыдущей, без использования вспомогательных переменных X1 и X2. Это сделано специально, для того чтобы продемонстрировать, что можно поступать и тем и другим способом.

Вернемся к программе КОРНИ. Теперь мы получим, например, такую выдачу:

```
КОЭФФИЦИЕНТЫ КВАДРАТНОГО УРАВНЕНИЯ A=1.0000000000 B=-1.0000000000 C=-1.0000000000 КОРНИ X1=-0.61803399 X2=1.61803399
```

Что-то можно разобрать, но выглядит неважно.

Для получения красивых выдaч нужно научиться располагать выдачу по строкам желаемым образом. Для этого следует разобратся в работе печатающего устройства паскаль-машины. Будем представлять его как пишущую машинку, в которой печать осуществляется символ за символом, после чего печатающая головка останавливается справа от напечатанного текста. С этого места начнется печать по следующей инструкции вывода. Как только текущая строка заполнится, печать продолжится на следующей строке. Специальная инструкция вывода WRITELN (состоит из одного слова) передвигает печатающую головку в начало следующей строки. Можно записывать инструкцию WRITELN с параметрами, как инструкцию WRITE. В этом случае WRITELN напечатает указанные значения точно так же, как WRITE, а потом переведет печатающую головку на новую строку. Таким образом, инструкция WRITELN(E1,E2,...,EN) эквивалентна двум инструкциям

WRITE(E1,E2,...,EN); WRITELN

П р и м е р: Последовательность инструкций

WRITE(1); WRITELN(2); WRITE(3)

напечатает

```
1 2
3
```

Аналогично, существует инструкция READLN, выполняющая переход к началу новой строки в исходных данных (исходные да

ные можно представлять себе записанными на бумаге в несколько строчек). С ее помощью можно пропустить исходные данные, еще остающиеся в текущей строке.

Сделаем красивую печать в нашей программе:

```
PROGRAM КОРНИ (INPUT, OUTPUT);
VAR A, B, C, D, X1, X2: REAL;
BEGIN READ (A, B, C);
  WRITELN(' КОЭФФИЦИЕНТЫ КВАДРАТНОГО',
    ' УРАВНЕНИЯ');
  WRITELN(' A=', A, ' B=', B, ' C=', C);
  D:=SQRT(SQR(B)-4*A*C);
  X1:=(-B-D)/(2*A); X2:=(-B+D)/(2*A);
  WRITELN; WRITE(' КОРНИ');
  WRITELN(' X1=', X1, ' X2=', X2);
END.
```

Результатом будет выдача

```
КОЭФФИЦИЕНТЫ КВАДРАТНОГО УРАВНЕНИЯ
A=1.00000000 B=-1.00000000 C=-1.00000000
КОРНИ X1=-0.6180399 X2=1.6180399
```

З а м е ч а н и е 1. Символ, выдаваемый в самом начале каждой строки, является управляющим. Он не воспроизводится на бумаге, а вызывает различные специальные действия печатающего устройства, например переход к следующей странице. Чтобы избежать неожиданных эффектов на выдаче, печать строки следует начинать с пробела.

З а м е ч а н и е 2. Точка с запятой после последней инструкции (перед END.) не нужна, однако ее употребление не является ошибкой.

2.5. Условные инструкции

Машина Поста может анализировать состояние ленты посредством инструкций разветвления. Ясно, что и для паскаль-машины нужны средства анализа обстановки и выбора варианта действия в зависимости от результатов анализа. Указанной цели служат условные инструкции. Форма записи условных инструкций такова:

```
IF (условие) THEN BEGIN
  (инструкция1);
  ...
  (инструкцияn)
END
ELSE BEGIN
```

```

{инструкцияn+1};
...
{инструкцияn+m}
END

```

Под условием пока будем понимать два выражения, соединенных знаками = (равно), <> (не равно), < (меньше), <= (меньше или равно), > (больше), >= (больше или равно).

Условная инструкция выполняется следующим образом. Сначала проверяется условие. Если оно истинно, выполняется группа инструкций {инструкция₁}—{инструкция_n}, если оно ложно, выполняются инструкции {инструкция_{n+1}}—{инструкция_{n+m}}.

В ряде случаев условная инструкция может записываться проще. Если при невыполнении условия делать ничего не нужно, ELSE-часть (от слова ELSE до конца инструкции) может отсутствовать. Если $n=1$ или $m=1$, соответствующая пара BEGIN — END также может отсутствовать. Обратим внимание на то, что перед словом ELSE ставить точку с запятой нельзя, поскольку она будет воспринята как конец условной инструкции без ELSE-части и слово ELSE окажется употребленным вне условной инструкции, что недопустимо.

Приведем несколько примеров условных инструкций. Пусть нужно вычислить модуль значения переменной X, не пользуясь стандартной функцией ABS:

```
IF X >= 0 THEN ABSX := X ELSE ABSX := -X;
```

Можно поступать иначе, заранее присвоив ABSX некоторое значение, а затем изменить его, если это необходимо:

```
ABSX := X; IF X < 0 THEN ABSX := -X;
```

Дополним программу решения квадратных уравнений анализом знака дискриминанта. Если дискриминант отрицателен, следует сообщить, что корней нет, в противном случае будем действовать, как в предыдущей версии программы:

```

PROGRAM KORNI(INPUT,OUTPUT);
VAR A,B,C,D,X1,X2:REAL;
BEGIN READ (A,B,C);
  WRITELN(' КОЭФФИЦИЕНТЫ КВАДРАТНОГО',
    ' УРАВНЕНИЯ');
  WRITELN(' A=', A, ' B=', B, ' C=', C);
  IF SQR(B)-4*A*C < 0 THEN
    WRITELN(' ВЕЩЕСТВЕННЫХ КОРНЕЙ НЕТ')
  ELSE BEGIN
    D := SQR(SQR(B)-4*A*C);
    X1 := (-B-D)/(2*A); X2 := (-B+D)/(2*A);
    WRITELN; WRITE(' KORNI');
  END

```

```

WRITELN(' X1=', X1, ' X2=', X2);
END
END.

```

Если не окружить инструкции, принадлежащие ELSE-части, парой BEGIN — END, в ELSE-часть попадет лишь одна инструкция — извлечения корня из дискриминанта. В результате при отрицательном дискриминанте после печати текста ВЕЩЕСТВЕННЫХ КОРНЕЙ НЕТ начнут выполняться инструкции присваивания X1 и X2 с неопределенным значением переменной D. Скорее всего, паскаль-машина заметит ошибочность подобных действий, напечатает сообщение НЕОПРЕДЕЛЕННОЕ ЗНАЧЕНИЕ В ВЫРАЖЕНИИ и прекратит выполнение программы, указав последнюю выполненную инструкцию. Если же реализация паскаль-машины такова, что действия с неопределенными значениями не фиксируются, результат работы программы будет непредсказуем.

2.6. Комментарии

При составлении программ нужно помнить, что они предназначены не только для выполнения на вычислительных машинах. Их читают, пытаются понять, переделывают люди. Поэтому программы должны быть удобны как для ЭВМ, так и для человека. Чтобы разъяснить смысл и обосновать правильность, их снабжают пояснениями — комментариями. *Комментарий* — это любой текст, заключенный в скобки (* и *). Этот текст не влияет на функционирование паскаль-машины, он важен лишь для человека, читающего программу. Сделать комментарии полезными и соответствующими программе — дело чести программиста.

Рассмотрим следующую задачу. Требуется прочитать в переменные A, B и C три числа и поменять местами значения этих переменных так, чтобы выполнялось соотношение $A \leq B \leq C$. Для этого используем приводимую ниже программу:

```

PROGRAM СОРТИРОВКА3(INPUT,OUTPUT);
VAR A,B,C,D: REAL;
BEGIN
  WRITELN(' СОРТИРОВКА ТРЕХ ЧИСЕЛ ПО',
    ' ВОЗРАСТАНИЮ');
  READ(A,B,C);
  WRITELN(' ИСХОДНЫЕ ЧИСЛА: ',A,' ',B,' ',C);
  IF A>B THEN BEGIN D:=A; A:=B; B:=D END;
  (*ТЕПЕРЬ A<= B*)
  IF A>C THEN BEGIN D:=A; A:=C; C:=D END;
  (*A<= B, A<= C*)
  IF B>C THEN BEGIN D:=B; B:=C; C:=D END;

```



```
(* A <= B <= C*)
WRITELN(' РЕЗУЛЬТАТ СОРТИРОВКИ: ', A, ' ', B, ' ', C)
END.
```

В данном примере комментарии содержат утверждения, истинность которых обеспечивается выполнением предыдущих инструкций. Комментарии помогают убедиться в правильности составленной программы.

2.7. Инструкции цикла

Вычислительные машины способны выполнить за секунду миллионы действий. Но для того, чтобы заставить машину выполнить эти действия, не записывая программу из миллионов инструкций (составление программ в таком случае потеряло бы всякий смысл), нужно иметь возможность многократно выполнять некоторые фрагменты программы. Другое соображение состоит в том, что при составлении программы не всегда известно, сколько раз нужно повторять то или иное действие (это может зависеть от исходных данных программы), и поэтому мы лишены возможности просто выписать инструкции нужное число раз. На машине Поста *цикл*, т. е. многократно выполняемый фрагмент программы, организуется с помощью отсылок, а выход из цикла (прекращение повторений) определяется работой команды разветвления. Паскаль-машина снабжена более удобными средствами организации циклов. В настоящем разделе мы изучим две разновидности инструкций цикла — WHILE—DO и REPEAT—UNTIL. Первая из них записывается следующим образом:

```
WHILE (условие) DO BEGIN
  (инструкция1);
  ...
  (инструкцияn)
END
```

Инструкция WHILE—DO выполняется следующим образом. Сначала проверяется условие. Если оно истинно, выполняются (инструкция₁), ..., (инструкция_n). Затем происходит возврат к проверке условия. Если оно вновь оказалось истинным, опять выполняются (инструкция₁), ..., (инструкция_n) и т. д. Если же условие ложно, выполнение инструкции цикла считается законченным, паскаль-машина переходит к обработке последующих инструкций. Таким образом, если условие с самого начала оказалось ложным, (инструкция₁), ..., (инструкция_n) не будут выполнены ни разу. В цикле WHILE—DO (условие) — это условие выполнения цикла: пока оно истинно, паскаль-машина из цикла

выйдет. Как и в случае условных инструкций, при $n=1$ пара BEGIN — END может отсутствовать. При $n>1$ она обязательна, так как иначе повторяться будет только одна инструкция, стоящая сразу после слова DO.

Несколько иначе выглядит цикл REPEAT — UNTIL:

```
REPEAT
  (инструкция1);
  ...
  (инструкцияn)
UNTIL (условие)
```

Паскаль-машина следующим образом обрабатывает цикл REPEAT — UNTIL. Сначала выполняются инструкции (инструкция₁), ..., (инструкция_n). Затем проверяется условие. Если оно ложно, происходит возврат к выполнению инструкций (инструкция₁) — (инструкция_n). Если условие истинно, цикл считается выполненным и паскаль-машина переходит к обработке последующих инструкций. Таким образом, в цикле REPEAT — UNTIL (условие) — это условие завершения цикла: цикл повторяется, пока оно ложно.

Приведем примеры использования инструкций цикла. Пусть вводится последовательность положительных чисел. Замыкает последовательность число 0. Требуется отпечатать все вводимые числа, включая 0:

```
PROGRAM КОПИЯ (INPUT,OUTPUT);
VAR ОЧЕРЕДНОЕ-ЧИСЛО: REAL;
BEGIN
  WRITELN(' КОПИЯ ИСХОДНЫХ ДАННЫХ');
  REPEAT
    READ(ОЧЕРЕДНОЕ-ЧИСЛО);
    WRITE(' ',ОЧЕРЕДНОЕ-ЧИСЛО)
  UNTIL ОЧЕРЕДНОЕ-ЧИСЛО=0;
  WRITELN
END.
```

Теперь несколько изменим условие задачи. Пусть завершающий последовательность 0 печатать не нужно. В этом случае целесообразно воспользоваться циклом WHILE—DO:

```
PROGRAM КОПИЯ-БЕЗ-ПОСЛЕДНЕГО (INPUT,OUTPUT);
VAR ОЧЕРЕДНОЕ-ЧИСЛО: REAL;
BEGIN
  WRITELN(' КОПИЯ ИСХОДНЫХ ДАННЫХ БЕЗ',
    ' ЗАВЕРШАЮЩЕГО 0');
  READ(ОЧЕРЕДНОЕ-ЧИСЛО);
```

```

WHILE ОЧЕРЕДНОЕ-ЧИСЛО <> 0 DO BEGIN
  WRITE(' ', ОЧЕРЕДНОЕ-ЧИСЛО);
  READ(ОЧЕРЕДНОЕ-ЧИСЛО)
END;
WRITELN
END.

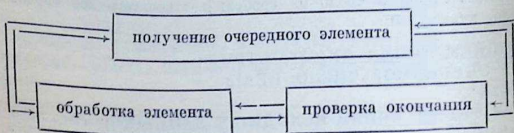
```

Предложенная программа правильно сработает и в том случае, когда 0 является первым (и единственным) элементом последовательности.

З а м е ч а н и е. Следует иметь в виду, что выполнение циклов WHILE—DO и REPEAT — UNTIL может никогда не закончиться (например, если инструкции, входящие в цикл, не изменяют значений переменных, входящих в условие). Поэтому необходимо обращать внимание на обеспечение конечности числа повторений цикла.

2.8. Обработка последовательностей

Обработка последовательностей — весьма распространенная задача в программировании. В подобных задачах требуется чередовать три действия:



Так, в задаче копирования исходных данных (см. разд. 2.7) получение очередного элемента сводилось к инструкции READ, обработка — к инструкции WRITE, а проверка окончания состояла в сравнении с нулем.

Перечисленные действия должны выполняться циклически, т. е. по кругу. Круг этот можно проходить в двух направлениях: по ходу часовой стрелки и против хода часовой стрелки. Выбор направления обхода зависит от того, можно ли произвести проверку окончания до обработки элемента. Если можно, то круг следует проходить по ходу часовой стрелки. Программа печати копии исходных данных без завершающего нуля принадлежит этой категории, поскольку повторения следовало закончить, как только встретится число 0, а сам 0 обработке не подлежал.

Рассмотрим теперь вопрос, с какого места круга начинать ход. Мы изучили две инструкции цикла: WHILE—DO и REPEAT

UNTIL. Если воспользоваться первой из них, обход по часовой стрелке запишется в виде

```
WHILE (условие продолжения) DO BEGIN  
  (обработка элемента);  
  (получение очередного элемента)  
END
```

В случае REPEAT — UNTIL имеем:

```
REPEAT  
  (обработка элемента);  
  (получение очередного элемента)  
UNTIL (условие завершения)
```

Чтобы обеспечить правильную работу цикла, нужно до входа в цикл должным образом присвоить начальные значения переменным, т. е. произвести начальную установку. Неправильная начальная установка — одна из распространенных ошибок. Если мы проходим круг из трех действий по часовой стрелке, инструкции обработки предшествуют получению очередного элемента. Значит, первый элемент следует получить особым образом, до входа в цикл, что и сделано в программе КОПИЯ-БЕЗ-ПОСЛЕДНЕГО.

Рассмотрим еще один пример. Будем считать, что исходные данные по-прежнему представляют собой последовательность чисел, оканчивающуюся нулем. Требуется напечатать минимальное из вводимых ненулевых чисел. Предполагается, что последовательность не состоит из одного нуля. Как и раньше, получение очередного элемента сводится к его вводу, проверка окончания — к сравнению с нулем, а вот обработка элемента записывается чуть сложнее. Пусть переменная МИНИМУМ хранит минимальное из введенных чисел. Тогда каждое новое число нужно напечатать (чтобы убедиться в правильности программы; если эта программа будет использоваться для поиска минимального элемента длинной последовательности, печать можно убрать), а затем сравнить с минимумом и, если оно оказалось меньше, сделать его значением переменной МИНИМУМ.

ОБРАБОТКА ЭЛЕМЕНТА *):

```
WRITE(' ', ОЧЕРЕДНОЕ_ЧИСЛО);  
IF МИНИМУМ > ОЧЕРЕДНОЕ_ЧИСЛО THEN  
  МИНИМУМ := ОЧЕРЕДНОЕ_ЧИСЛО
```

Получаем следующий набросок программы:

```
PROGRAM НАБРОСОК-ПОИСКА-МИНИМУМА (INPUT,  
OUTPUT);
```

*) Эта строка не есть часть программы, она служит для именования фрагмента программы. Подобная запись используется и в дальнейшем.

```

VAR ОЧЕРЕДНОЕ_ЧИСЛО, МИНИМУМ: REAL;
BEGIN
  WRITELN(' МИНИМАЛЬНОЕ ИЗ ЧИСЕЛ');
  READ(ОЧЕРЕДНОЕ_ЧИСЛО);
  WHILE ОЧЕРЕДНОЕ_ЧИСЛО <> 0 DO BEGIN
    (* ОБРАБОТКА ЭЛЕМЕНТА *)
    WRITE(' ', ОЧЕРЕДНОЕ_ЧИСЛО);
    IF МИНИМУМ > ОЧЕРЕДНОЕ_ЧИСЛО THEN
      МИНИМУМ:=ОЧЕРЕДНОЕ_ЧИСЛО;
    (*ПОЛУЧЕНИЕ ОЧЕРЕДНОГО ЭЛЕМЕНТА*)
    READ(ОЧЕРЕДНОЕ_ЧИСЛО)
  END;
  WRITELN; WRITELN(' РАВНО ', МИНИМУМ)
END.

```

Набросок содержит типичную ошибку — мы не приняли мер для нормальной обработки первого элемента (бросились в воду прямо в одежде). Переменная МИНИМУМ не получила начального значения, и первое же сравнение

МИНИМУМ > ОЧЕРЕДНОЕ_ЧИСЛО

будет незаконным, поскольку содержит неопределенную величину.

Очевидный способ исправления отмеченной ошибки — по-особому организовать не только получение, но и обработку первого элемента. Ясно, что минимум из одного числа равен самому числу без каких-либо «если». Получаем следующую программу:

```

PROGRAM ПРЯМОЛИНЕЙНЫЙ-ПОИСК-МИНИМУМА
(INPUT,OUTPUT);
VAR ОЧЕРЕДНОЕ_ЧИСЛО,МИНИМУМ: REAL;
BEGIN
  WRITELN(' МИНИМАЛЬНОЕ ИЗ ЧИСЕЛ');
  READ(ОЧЕРЕДНОЕ_ЧИСЛО);
  (* ОБРАБОТКА ПЕРВОГО ЭЛЕМЕНТА *)
  WRITE(' ',ОЧЕРЕДНОЕ_ЧИСЛО);
  МИНИМУМ:=ОЧЕРЕДНОЕ_ЧИСЛО;
  (* ПОЛУЧЕНИЕ ВТОРОГО ЭЛЕМЕНТА *)
  READ(ОЧЕРЕДНОЕ_ЧИСЛО);
  WHILE ОЧЕРЕДНОЕ_ЧИСЛО <> 0 DO BEGIN
    (* ОБРАБОТКА ЭЛЕМЕНТА *)
    WRITE(' ', ОЧЕРЕДНОЕ_ЧИСЛО);
    IF МИНИМУМ > ОЧЕРЕДНОЕ_ЧИСЛО THEN
      МИНИМУМ:=ОЧЕРЕДНОЕ_ЧИСЛО;
    (* ПОЛУЧЕНИЕ ОЧЕРЕДНОГО ЭЛЕМЕНТА *)
    READ(ОЧЕРЕДНОЕ_ЧИСЛО)
  END;

```

```
WRITELN; WRITELN(' РАВНО ',МИНИМУМ)
END.
```

Обратим внимание, что если первый элемент обрабатывается до входа в цикл, то второй элемент следует получать также до цикла.

Посмотрим теперь на предложенную программу внимательнее. Нетрудно сообразить, что начальная установка

```
МИНИМУМ:=ОЧЕРЕДНОЕ-ЧИСЛО
```

позволяет обработать первый элемент наравне с остальными. Другой возможный вариант начальной установки — присвоить переменной МИНИМУМ самое большое число, представимое в данной реализации паскаль-машины. В обоих случаях после обработки первого элемента условием инструкции

```
IF МИНИМУМ>ОЧЕРЕДНОЕ-ЧИСЛО THEN
```

```
МИНИМУМ:=ОЧЕРЕДНОЕ-ЧИСЛО
```

значение переменной МИНИМУМ окажется равным этому элементу. Мы воспользуемся первой из предложенных начальных установок. Получаем программу:

```
PROGRAM ПОИСК-МИНИМУМА(INPUT,OUTPUT);
VAR ОЧЕРЕДНОЕ-ЧИСЛО,МИНИМУМ: REAL;
BEGIN
```

```
(* НАЧАЛЬНАЯ УСТАНОВКА *)
```

```
WRITELN(' МИНИМАЛЬНОЕ ИЗ ЧИСЕЛ');
```

```
READ(ОЧЕРЕДНОЕ-ЧИСЛО);
```

```
МИНИМУМ:=ОЧЕРЕДНОЕ-ЧИСЛО;
```

```
WHILE ОЧЕРЕДНОЕ-ЧИСЛО <> 0 DO BEGIN
```

```
(* ОБРАБОТКА ЭЛЕМЕНТА *)
```

```
WRITE(' ', ОЧЕРЕДНОЕ-ЧИСЛО);
```

```
IF МИНИМУМ > ОЧЕРЕДНОЕ-ЧИСЛО THEN
```

```
МИНИМУМ :=ОЧЕРЕДНОЕ-ЧИСЛО;
```

```
(* ПОЛУЧЕНИЕ ОЧЕРЕДНОГО ЭЛЕМЕНТА *)
```

```
READ(ОЧЕРЕДНОЕ-ЧИСЛО)
```

```
END;
```

```
WRITELN; WRITELN(' РАВНО ',МИНИМУМ)
```

```
END.
```

Итак, решение задач обработки последовательностей распадается на два этапа — программирование цикла и выбор начальных установок. Текстуально начальные установки стоят до цикла, но продумываются они, как правило, когда цикл уже написан. Впрочем, в некоторых задачах можно дать общие рекомендации. Если требуется просуммировать числа, начальное значение суммы полагается равным 0, если числа перемножаются, начальным значением

произведения становится 1. После такой начальной установки первое число можно обрабатывать наравне с остальными.

Разберем теперь случай, когда проверку окончания можно организовать только после обработки очередного элемента. Тогда круг из трех действий следует проходить против часовой стрелки, а сам цикл запишется на языке паскаль в виде

```
WHILE <условие продолжения> DO BEGIN
    <получение очередного элемента>;
    <обработка элемента>
END
```

или в виде

```
REPEAT
    <получение очередного элемента>;
    <обработка элемента>
UNTIL <условие завершения>
```

Опять-таки начальную установку следует выполнять так, чтобы при входе в цикл условие продолжения не содержало переменных с неопределенными значениями и все было готово для получения очередного элемента.

Программа печати копии исходных данных служит примером обхода против часовой стрелки. Рассмотрим еще одну программу. Пусть требуется читать исходные данные, пока сумма прочитанных чисел не превысит 100. Полученную сумму следует напечатать. Ясно, что, не обработав очередной элемент, нельзя решить, пора ли выходить из цикла.

```
PROGRAM ПРЕВЫШЕНИЕ(INPUT,OUTPUT);
VAR ОЧЕРЕДНОЕ-ЧИСЛО,СУММА: REAL;
BEGIN
    (* НАЧАЛЬНАЯ УСТАНОВКА *)
    WRITELN(' СУММА ЧИСЕЛ');
    СУММА:=0;
    REPEAT
        (* ПОЛУЧЕНИЕ ОЧЕРЕДНОГО ЭЛЕМЕНТА *)
        READ(ОЧЕРЕДНОЕ-ЧИСЛО);
        (* ОБРАБОТКА ЭЛЕМЕНТА *)
        WRITE(' ',ОЧЕРЕДНОЕ-ЧИСЛО);
        СУММА:=СУММА+ОЧЕРЕДНОЕ-ЧИСЛО
    (* ПРОВЕРКА ОКОНЧАНИЯ *)
    UNTIL СУММА >100;
    WRITELN; WRITELN(' ВПЕРВЫЕ ПРЕВЫСИЛА 100');
    WRITELN(' И СОСТАВИЛА ',СУММА)
END.
```

2.9. Тип INTEGER

Этот тип представляет множество целых чисел. В каждой реализации паскаль-машины диапазон представимых целых чисел ограничен. На небольших ЭВМ целые числа обязаны лежать в диапазоне от $-32\,768$ до $32\,767$; для ЕС ЭВМ и БЭСМ-6 этот диапазон значительно шире.

Переменные типа INTEGER описываются следующим образом:

```
VAR <переменная1>, . . ., <переменнаяn> :INTEGER;
```

Если в программе описываются и вещественные, и целые переменные, слово VAR ставится только в начале описания переменных, например:

```
VAR A,B :REAL; M,N:INTEGER;
```

Над целыми числами определены операции:

$+$, $-$, $*$,

$/$ (дает вещественный результат),

DIV (целочисленное деление),

MOD (остаток от целочисленного деления).

П р и м е р.

```
5/2=2.5    5 DIV 2=2    5 MOD 2=1
```

Операции DIV и MOD следует отделять от операндов пробелами, иначе вместо операндов и операции получится одно длинное имя. Функции преобразования вещественных значений в целые:

ROUND — округление до ближайшего целого,

TRUNC — отбрасывание дробной части.

П р и м е р.

```
ROUND(3.6)=4    TRUNC(3.6)=3
```

В качестве примера действий с целыми числами рассмотрим программу вычисления факториала натурального числа, являющегося единственным исходным данным. Эту задачу можно трактовать как задачу обработки последовательности. Обозначим через N число, факториал которого вычисляется. Следует перемножить все числа от 2 до N . Таким образом, для получения очередного элемента нужно увеличить ранее обрабатывавшийся элемент на 1. Обработка состоит в умножении элемента на уже накопленное произведение, а для проверки окончания нужно сравнить очередной элемент с N . Проверку окончания можно производить и до, и после обработки элемента. Пойдем сначала по первому пути. Получаем программу:

```
PROGRAM ФАКТОРИАЛ(INPUT,OUTPUT);  
VAR N,K,FACT:INTEGER;
```

```
(* N — ИСХОДНОЕ ЧИСЛО, K — ТЕКУЩИЙ МНОЖИТЕЛЬ,
  FACT — НАКОПЛЕННОЕ ПРОИЗВЕДЕНИЕ *)
BEGIN READ(N); WRITE(' ', N, ' ФАКТОРИАЛ = ');
  K:=2; FACT:=1;
  WHILE K <= N DO BEGIN
    FACT:=FACT*K; (* В ЭТОТ МОМЕНТ FACT = K! *)
    K:=K+1
  END; (* FACT = N! *)
  WRITELN(FACT)
END.
```

Если производить проверку после обработки, начальные установки и цикл запишутся так:

```
K:=1; FACT:=1;
WHILE K < N DO BEGIN
  K:=K+1;
  FACT:=FACT*K (* В ЭТОТ МОМЕНТ FACT = K! *)
END; (* FACT = N! *)
```

Задачи

В последующих задачах исходные данные — последовательность вещественных чисел, оканчивающаяся нулем. Нуль обработки не подлежит.

1. Напечатать сумму, среднее арифметическое, сумму квадратов и произведение вводимых чисел.

2. Напечатать отдельно суммы положительных и отрицательных элементов последовательности.

3. Пусть все элементы последовательности различны. Напечатать среднее арифметическое элементов без учета самого большого и самого маленького числа.

4. Напечатать второй по величине элемент последовательности предполагая, что все элементы различны.

2.10. Об инвариантах циклов

Инвариант — то, что не меняется. Инвариант цикла — это соотношение между значениями переменных, которое остается справедливым при любом прохождении цикла. В последующем изложении нам хотелось бы подчеркнуть ту мысль, что инварианты циклов помогают не только доказывать правильность программ, но и разрабатывать программы. Если думать над доказательством правильности уже при разработке программ, они получатся яснее, изящнее и зачастую эффективнее, чем программы, написанные бессистемно.

Рассмотрим в качестве примера процесс разработки программ

возведения целого числа в натуральную степень. Через M обозначим основание степени, через N — показатель, через P — переменную, в которой будем накапливать результат. В конце программы требуется обеспечить истинность соотношения

$$P = M^N \quad (1)$$

Мы должны стремиться к этой цели посредством некоторого цикла. Для выбора соотношения, остающегося истинным в цикле, требуется придумать обобщение соотношения (1). Такое «придумывание» — процесс творческий, тут трудно дать готовый рецепт. Полезный прием — введение в соотношение типа (1) новых переменных. Помогает выбору обобщения тот факт, что справедливость обобщенного соотношения должна обеспечиваться и до входа в цикл. Простота начальных установок, обеспечивающих истинность инварианта, — веское соображение в пользу того, что инвариант выбран правильно. В рассматриваемой задаче будем поддерживать истинность соотношения

$$P \cdot M1^{N1} = M^N \quad (2)$$

Начальные установки, делающие соотношение (2) истинным, записываются очевидным образом:

$$P := 1; M1 := M; N1 := N$$

Чтобы из (2) следовало (1), второй сомножитель в (2) должен равняться единице. Этого можно добиться, сделав $N1$ равным нулю. Таким образом, из соображений правильности получаем условие выхода из цикла и следующий набросок программы:

```
P:=1; M1:=M; N1:=N;
WHILE N1<>0 DO BEGIN
  ТЕЛО ЦИКЛА
  (* P*(M1 В СТЕПЕНИ N1) = (M В СТЕПЕНИ N) *)
END
(*P = (M В СТЕПЕНИ N) *)
```

Чтобы условие $N1 \neq 0$ стало ложным, в теле цикла $N1$ надо уменьшать. При этом множитель $(M1 \text{ в степени } N1)$ будет уменьшаться. Чтобы соотношение (2) оставалось истинным, P надо во столько же раз увеличивать. В результате получаем следующую программу:

```
PROGRAM СТЕПЕНЬ(INPUT,OUTPUT);
VAR M,N,M1,N1,P :INTEGER;
BEGIN
  READ(M,N); WRITE(' ',M,' В СТЕПЕНИ ',N,' = ');
  P:=1; M1:=M; N1:=N;
  WHILE N1<>0 DO BEGIN
```

```

N1 := N1 - 1; P := P * M1
(* P * (M1 В СТЕПЕНИ N1) = (M В СТЕПЕНИ N) *)
END;
(* P = (M В СТЕПЕНИ N) *)
WRITELN(P)
END.

```

Чтобы доказать, что соотношение (2) является инвариантом цикла, проще всего воспользоваться методом математической индукции. Основанием индукции служит состояние перед входом в цикл индуктивный шаг очевиден.

Для доказательства того, что работа программы завершится после конечного числа повторений цикла, воспользуемся фактом который примем на веру: убывающая последовательность натуральных чисел не может быть бесконечной. В нашей программе значения N1 уменьшается на единицу при каждом повторении цикла, поэтому условие $N1 < > 0$ в конце концов станет ложным.

Подчеркнем, что мы доказали два утверждения (в предположении корректности исходных данных и отсутствия переполнения).

1. Если работа программы завершится, значением переменной P станет M в степени N.

2. Работа программы завершится после конечного числа повторений цикла.

С математической точки зрения наша программа правильна. Полезно, однако, посмотреть на нее с другой, инженерной точки зрения, проанализировать характеристики созданного изделия. Оценим число умножений, которое требуется программе. В данном случае легко видеть, что умножений будет N. Число умножений можно уменьшить, если выбрать более эффективный, чем вычитание единицы, способ уменьшения N1. Попытаемся изменить форму представления величины $M1^{N1}$, воспользовавшись соотношением

$$A^2B = (A \cdot A)^B \quad (3)$$

Применение соотношения (3) сразу уменьшит показатель степени в два раза. Ясно, что это эффективнее, чем вычитание единицы.

Соотношение (3) можно применять, пока значение N1 четно. Вставим в нашу программу перед инструкцией $N1 := N1 - 1$ цикл уменьшения N1 в два раза с одновременным возведением M1 в квадрат. Программа примет следующий вид:

```

PROGRAM СТЕПЕНЬ(INPUT,OUTPUT);
VAR M,N,M1,N1,P:INTEGER;
BEGIN
  READ(M,N); WRITE('M, ' В СТЕПЕНИ 'N, ' = ');
  P:=1; M1:=M; N1:=N;
  WHILE N1 < > 0 DO BEGIN

```

```

WHILE N1 MOD 2 = 0 DO BEGIN
  N1 := N1 DIV 2; M1 := M1 * M1
END;
N1 := N1 - 1; P := P * M1
(* P * (M В СТЕПЕНИ N1) = (M В СТЕПЕНИ N) *)
END;
(* P = (M В СТЕПЕНИ N) *)
WRITELN(P)
END.

```

По отношению к новой программе достаточно доказать конечность числа повторений вставленного цикла. Единственное целое число, которое можно до бесконечности делить пополам, получая четный результат, есть нуль. Но условие внешнего цикла $N1 < > 0$ гарантирует завершенность цикла внутреннего.

Проследив поведение нового варианта программы при $N=15$ и $N=16$, можно подсчитать, что в первом случае требуется семь умножений, а вот втором — пять. Таким образом, вставив цикл деления $N1$ пополам, мы получили существенный выигрыш в быстрейшем действии.

2.11. Определение констант

Программы описывают поведение реальных или модельных объектов. Объекты характеризуются рядом параметров. Некоторые из них меняются во время работы программы; они описываются переменными. Другие характеристики постоянны; их естественно описать константами. Если в программе явно фигурируют значения константных характеристик, это плохо по двум причинам.

1. Такая программа не является наглядной. За числовым значением не видно, какая характеристика имеется в виду: масса, скорость, размер и т. п.

2. Программу трудно переделывать. Если требуется произвести настройку на другие значения константных характеристик, придется модифицировать многие места программы.

В языке паскаль константы можно наделять именами. Эти имена разрешается употреблять везде, где может стоять соответствующее значение. Инструкция задания имен констант (определения констант) записывается так:

$\text{CONST } \langle \text{имя}_1 \rangle = \langle \text{значение}_1 \rangle, \dots, \langle \text{имя}_n \rangle = \langle \text{значение}_n \rangle;$

Тип константы определяется типом значения. Определение констант предшествует определению типов (см. разд. 2.12).

Приведем в качестве примера фрагмент программы, в котором вычисляются время падения тела с высоты H и его конечная скорость (сопротивлением воздуха пренебрегаем):

```
PROGRAM ПАДЕНИЕ(INPUT,OUTPUT);  
CONST G=9.81;  
VAR H,T,V: REAL;  
BEGIN  
  READ(H);  
  T:=SQRT(2*H/G);  
  V:=SQRT(2*G*H);  
  . . .
```

Программу, записанную в таком виде, легко переделать, если мы захотим точнее задать ускорение свободного падения на определенной широте Земли или вычислить время и конечную скорость падения на поверхность Луны.

2.12. Определение новых типов. Отрезочные типы

Тип переменной — это множество значений, которые может принимать переменная. В языке паскаль имеется несколько так называемых стандартных типов, которые не нуждаются в описании. Два из них мы изучили — INTEGER и REAL. Кроме того, программист может образовывать новые типы и наделять их именами, используя инструкцию определения типов:

TYPE <имя типа₁> = <описание типа₁>;

. . .
<имя типа_n> = <описание типа_n>;

Определения типов располагаются между определениями констант и описаниями переменных.

На протяжении курса мы будем знакомиться с разными способами образования новых типов. Сейчас изучим один из них — задание отрезков уже описанных типов:

TYPE <имя типа> = <нижняя граница>..<верхняя граница>

Пример.

```
CONST ПРЕДЕЛ = 200;  
TYPE ВОЗРАСТ = 0..ПРЕДЕЛ;  
VAR B1,B2:ВОЗРАСТ;
```

Присваивание переменным отрезочного типа значений, лежащих вне отрезка, является ошибкой. В остальном отрезочный тип (в нашем случае ВОЗРАСТ) равноправен с типом, значения которого задают границы отрезка (в нашем случае — с типом INTEGER).

Запрещается образовывать отрезки с границами типа REAL. Использование типов, определенных программистом, целесообразно по крайней мере по двум причинам.

1. Повышается наглядность программ. Название ВОЗРАСТ говорит читающему программу гораздо больше, чем просто INTEGER.

2. Повышается надежность программ. Паскаль-машина не допустит выхода за границы отрезка.

В заключение раздела упомянем еще один способ определения новых типов:

TYPE <новое имя типа> = <старое имя типа>;

Пр и м е р:

TYPE МАССА = REAL;

Мы настоятельно рекомендуем определять новые типы по соображениям повышения наглядности программ. Группа описаний

TYPE МАССА=REAL; СКОРОСТЬ=REAL;

VAR M1,M2:МАССА; V1,V2:СКОРОСТЬ;

предпочтительнее более короткой, но менее наглядной записи

VAR M1,M2,V1,V2:REAL;

2.13. Массивы

Массив — это составной объект, образованный из компонент. Все компоненты массива имеют один и тот же тип. Отдельные компоненты обозначаются упорядоченной совокупностью n значений, называемых индексами. Число n называется размерностью массива.

Определение типа, значения которого являются массивами, выглядит так:

TYPE <имя типа> = ARRAY[<тип индекса₁>, ..., <тип индекса_n>] OF <тип компонент>;

Переменные-массивы описываются обычным образом.

Пр и м е р.

TYPE МАТРИЦА = ARRAY[1...3,1...3] OF REAL;

VAR M1,M2,M3:МАТРИЦА;

Массивы одного типа можно присваивать. Отдельные компоненты массива обозначаются так:

<имя массива>[<значение индекса₁>, ..., <значение индекса_n>]

(после имени массива в квадратных скобках через запятую перечисляются значения всех n индексов).

Пример.

$$M1 := M2; M3[K, L] := (M3[K-1, L] + M3[K, L-1]) / 2$$

Рассмотрим задачу, в которой требуется прочитать последовательность вещественных чисел и напечатать ее в обратном порядке. Пусть исходные данные открываются целым числом, задающим длину последовательности. Известно, что это число не больше 1000:

```
PROGRAM НАОБОРОТ(INPUT,OUTPUT);
CONST M=1000;
TYPE ХРАНИЛИЩЕ=ARRAY[1..M] OF REAL;
VAR X:ХРАНИЛИЩЕ; N,I:INTEGER;
(* X — МАССИВ ДЛЯ ХРАНЕНИЯ
ПОСЛЕДОВАТЕЛЬНОСТИ
N — ЧИСЛО ЭЛЕМЕНТОВ ПОСЛЕДОВАТЕЛЬНОСТИ
I — НОМЕР ОБРАБАТЫВАЕМОГО ЭЛЕМЕНТА *)
BEGIN READ(N);
  WRITELN(' ПОСЛЕДОВАТЕЛЬНОСТЬ ИЗ ',N,' ЧИСЕЛ:');
  I:=0;
  WHILE I<N DO BEGIN
    I:=I+1; READ(X[I]); WRITE(X[I])
    (* ПЕРВЫЕ I ЭЛЕМЕНТОВ ПОСЛЕДОВАТЕЛЬНОСТИ
    ЗАПОМНИЛИ В ПЕРВЫХ I ЭЛЕМЕНТАХ МАССИВ
    X *)
  END;
  (* ЗАПОМНИЛИ ПОСЛЕДОВАТЕЛЬНОСТЬ В МАССИВ
  X *)
  WRITELN;
  WRITELN(' ПЕЧАТАЕТСЯ В ОБРАТНОМ ПОРЯДКЕ:');
  I:=N+1;
  WHILE I>1 DO BEGIN
    I:=I-1; WRITE(X[I])
    (* НАПЕЧАТАЛИ (N-I+1)
    ПОСЛЕДНИХ ЭЛЕМЕНТОВ ПОСЛЕДОВАТЕЛЬНОСТИ *)
  END;
  (* НАПЕЧАТАЛИ ПОСЛЕДОВ. В ОБРАТНОМ ПОРЯДКЕ *)
  WRITELN; WRITELN(' *****')
```

В примере использованы два важнейших свойства массивов.

1. Массив может хранить много значений (в данном случае цел). Без возможности запомнить всю исходную последовательность целиком напечатать ее в обратном порядке было бы нелегко.

2. К компонентам массива можно обращаться в произвольном порядке, вычисляя значения индекса (индексов).

В программе мы дважды обрабатывали все компоненты массива: сначала в порядке возрастания индекса, затем в порядке убывания. Ситуация последовательной обработки элементов массива столь распространена, что для нее ввели специальную синтаксическую форму — FOR-циклы. Пусть требуется выполнить группу инструкций S для значений переменной $I = \text{НИГР}, \text{НИГР}+1, \dots, \text{ВЕГР}$. Это можно сделать посредством цикла

```
FOR I:=НИГР TO ВЕГР DO BEGIN
  S
END
```

Если группу S нужно выполнять для убывающих значений $I = \text{ВЕГР}, \text{ВЕГР}-1, \dots, \text{НИГР}$, используется инструкция

```
FOR I:=ВЕГР DOWNTO НИГР DO BEGIN
  S
END
```

При $\text{НИГР} > \text{ВЕГР}$ FOR-цикл не выполняется ни разу.

С использованием FOR-циклов программу НАОБОРОТ можно записать следующим образом:

```
PROGRAM НАОБОРОТ(INPUT,OUTPUT);
CONST M=1000;
TYPE ХРАНИЛИЩЕ=ARRAY[1..M] OF REAL;
VAR X :ХРАНИЛИЩЕ; N:INTEGER, I:1..M;
(* X — МАССИВ ДЛЯ ХРАНЕНИЯ
ПОСЛЕДОВАТЕЛЬНОСТИ
N — ЧИСЛО ЭЛЕМЕНТОВ ПОСЛЕДОВАТЕЛЬНОСТИ
I — НОМЕР ОБРАБАТЫВАЕМОГО ЭЛЕМЕНТА *)
BEGIN READ(N);
  WRITELN(' ПОСЛЕДОВАТЕЛЬНОСТЬ ИЗ ',N,' ЧИСЕЛ:');
  FOR I:=1 TO N DO BEGIN
    READ(X[I]); WRITE(X[I])
    (* ПЕРВЫЕ I ЭЛЕМЕНТОВ ПОСЛЕДОВАТЕЛЬНОСТИ
    ЗАПОМНИЛИ В ПЕРВЫХ I ЭЛЕМЕНТАХ МАССИВА
    X *)
  END;
  (* ЗАПОМНИЛИ ПОСЛЕДОВАТЕЛЬНОСТЬ В МАССИВЕ
  X *)
  WRITELN;
  WRITELN(' ПЕЧАТАЕТСЯ В ОБРАТНОМ ПОРЯДКЕ:');
  FOR I:=N DOWNTO 1 DO WRITE(X[I]);
  (* НАПЕЧАТАЛИ ПОСЛЕДОВАТЕЛЬНОСТЬ В
  ОБРАТНОМ ПОРЯДКЕ *)
```

```

WRITELN;
WRITELN(' *****')
END.

```

Программа стала короче и нагляднее, поскольку два действия — начальную установку и продвижение переменной I (наряду с проверкой окончания) — вообрал в себя заголовок FOR-цикла.

Запишем фрагмент программы, в котором суммируются элементы двумерного массива A:

```

CONST M=100; N=100;
TYPE МАТРИЦА=ARRAY[1..M,1..N] OF REAL
VAR A:МАТРИЦА; СУММА:REAL; I:1..M; J:1..N,
...
BEGIN
...
СУММА:=0;
FOR I:=1 TO M DO
  FOR J:=1 TO N DO
    СУММА:=СУММА+A[I,J];

```

2.14. Тип BOOLEAN

Множество значений типа BOOLEAN (логического) содержит всего два элемента — «ложь» (в программах на паскале обозначается FALSE) и «истина» (обозначается TRUE). Эти значения получаются при вычислении условий в результате выполнения операций сравнения $<$, $<=$, $>$, $>=$, $=$, $<>$. Таким образом, условия не являются особой конструкцией языка паскаль, а представляют собой частный случай выражений, а именно выражения со значениями логического типа. Подобные выражения можно не только использовать в условных инструкциях, но также присваивать (переменным типа BOOLEAN) и печатать.

Кроме операций сравнения для построения логических выражений можно использовать функцию ODD и операции NOT, AND, OR. Функция ODD имеет аргумент типа INTEGER; ее результат равен TRUE, если аргумент — нечетное число, и FALSE, если четное. Операции NOT (логическое отрицание), AND (И, логическое умножение), OR (ИЛИ, логическое сложение) имеют аргументы — результаты логического типа. Операция NOT имеет один аргумент, AND и OR — два. Результаты этих операций полностью определены табл. 2.1.

Пример. Присвоим логической переменной В значение, показывающее, принадлежит ли значение переменной X отрезку 1 до 2. Чтобы это условие было истинным, необходимо одновременно

Таблица 2.1

Логические операции

X	Y	NOT Y	X AND Y	X OR Y
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	TRUE
TRUE	FALSE		FALSE	TRUE
TRUE	TRUE		TRUE	TRUE

ное выполнение двух неравенств; поэтому воспользуемся операцией AND:

```
VAR X :REAL; B :BOOLEAN;
  BEGIN...B:=(X>=1) AND (X<=2);
```

Скобки здесь необходимы, как и во всех случаях, когда операндами логических операций являются выражения. При отсутствии скобок операция AND будет выполняться до операций сравнения, что приведет к ошибке.

Нередко при составлении программ со сложными логическими выражениями нужно строить отрицания выражений. Для этого полезно воспользоваться тождествами, известными из алгебры логики:

```
NOT(NOT(A))=A
NOT(A AND B)=NOT(A) OR NOT(B)
NOT(A OR B)=NOT(A) AND NOT(B)
```

Составим эскиз программы поиска наименьшего натурального числа, обладающего каким-либо заданным свойством. Поиск ограничим первой тысячей чисел:

```
PROGRAM ПОИСК(OUTPUT);
CONST ПРЕДЕЛ=1000;
VAR I:0..ПРЕДЕЛ; (* I — ПРОВЕРЯЕМОЕ ЧИСЛО *)
    B:BOOLEAN; (* B ПОКАЗЫВАЕТ, ПОДХОДИТ ЛИ I *)
BEGIN I:=0;
  REPEAT
    I:=I+1;
    B:=ОБЛАДАЕТ ЛИ I ЗАДАННЫМ СВОЙСТВОМ;
  UNTIL (I=ПРЕДЕЛ) OR B;
  (* ПОСЛЕ ЦИКЛА ВЫПОЛНЕНО (I=ПРЕДЕЛ) OR B *)
  IF B THEN
    WRITE(I,'—ПЕРВОЕ ПОДХОДЯЩЕЕ ЧИСЛО')
```



```
ELSE WRITE(' ДО 'ПРЕДЕЛ,  
'ПОДХОДЯЩИХ ЧИСЕЛ НЕТ')  
END.
```

Чтобы превратить эскиз в законченную программу, следует заменить записанную словами инструкцию $V:=\text{ОБЛАДАЕТ}...$ на одну или несколько инструкций, проверяющих конкретное свойство и присваивающих переменной V соответствующее значение. Кроме того, нужно конкретизировать печать в конце программы.

2.15. Процедуры и функции

Чрезвычайно важную роль в процессе познания играет формирование новых понятий. Такие понятия фиксируют познание и являются ступенями, позволяющими двигаться дальше. Представим себе, что в доказательствах геометрических утверждений нельзя пользоваться ранее выведенными теоремами, а рассуждения нужно доводить до аксиом. Видимо, геометрия просто не могла бы существовать в таких условиях. А уж про размер школьного учебника (если бы геометрию все-таки изучали) и подумать страшно. И в программировании целесообразно иметь способ фиксации накопленного знания. Аппарат процедур и функций как раз и служит этой цели. Так, стандартные процедуры и функции паскаль-машины READ, WRITE, SQRT и т. п. позволяют выполнять довольно сложные действия, не думая об их реализации.

Вычислительные машины помогают людям решать сложные задачи. Иметь помощника, особенно исполнительного, полезно. Однако, если ему нужно долго и по многу раз объяснять одни и те же действия, сложную задачу решить будет крайне трудно. Таким исполнителем, но «бестолковым» помощником является машина Поста. Пять типов инструкций — вот весь арсенал программиста (Конечно, машина Поста и не предназначалась для программирования.) Паскаль-машина куда понятливее. Можно записать последовательность паскаль-инструкций, реализующую некоторое действие, наделить ее именем и в дальнейшем, упоминая имя, заставить паскаль-машину выполнять всю последовательность инструкций. Такая именованная последовательность инструкций называется *процедурой*. Частный случай процедуры, вырабатывающий одно значение, которое можно использовать в выражении, называется *функцией*. Определение имени и последовательности инструкций называется описанием процедуры (функции), а приказ паскаль-машине выполнить эту последовательность — вызовом процедуры (функции).

Вернемся к аналогии с математикой. Если нужно доказать трудную теорему, сначала доказывают ряд вспомогательных утвер-

дений (лемм), из которых затем выводят требуемое утверждение. И в программировании при составлении сложной программы определяют процедуры, выполняющие действия, полезные для решения задачи, а затем с использованием этих процедур записывают программу, часто называемую также главной программой, поскольку процедуры — ее подчиненные, помощники.

Описания процедур располагаются после описаний переменных. Они не являются выполняемыми инструкциями. Если описания переменных определяют структуру памяти паскаль-машины, то описания процедур задают набор умений нашего экземпляра паскаль-машины.

Описание процедуры выглядит почти так же, как и запись самой программы: ведь и главная программа, и процедура реализуют некий алгоритм, просто процедура не дает решения всей задачи. Имеются, однако, и различия. Одно из них состоит в том, что главная программа получает исходные данные из источника, внешнего по отношению к паскаль-машине. Также вовне помещаются результаты. Процедура же общается с главной программой или с другой процедурой, ее вызвавшей, т. е. взаимодействие идет в рамках самой паскаль-машины. Разные партнеры — разные способы общения. При вызове процедур работает механизм передачи параметров. Соответственно в заголовке процедуры указываются имена и типы *формальных параметров*, показывающих, с какими исходными данными готова работать процедура. Заголовок процедуры записывается следующим образом:

```
PROCEDURE (имя процедуры)
  ((имя формального параметра1):<тип1>;
  ...
  (имя формального параметран):<типн>));
```

Заголовок функции выглядит так:

```
FUNCTION (имя функции)
  ((имя формального параметра1):<тип1>;
  ...
  (имя формального параметран):<типн>):<тип результата>;
```

В описание функции должна входить инструкция, в которой значение-результат присваивается имени функции.

Описание процедуры нагляднее описания главной программы. По тексту программы иногда довольно сложно понять, какие исходные данные ей нужны — инструкции READ могут быть разбросаны по всей программе. В случае процедуры информация об исходных для процедуры данных сосредоточена в заголовке, поэтому записать вызов процедуры легко.

Имена формальных параметров служат для обозначения внутри процедуры тех значений, с которыми она должна работать после вызова. Сами эти значения задаются при вызове и называются *фактическими параметрами*. Вызов процедуры записывается следующим образом:

(имя процедуры)((фактический параметр₁), ..., (фактический параметр_n))

Вызов функции должен входить в выражение. Вызов других процедур образует отдельную инструкцию. Отвечающие друг другу формальные и фактические параметры должны быть одного и того же типа, т. е. имя типа у них должно быть общее. Мелкое отличие процедуры от всей программы состоит в том, что в конце описания процедуры ставится не точка, а точка с запятой.

Подчеркнем, что механизм передачи параметров применяется только при общении между процедурами или между главной программой и процедурой. Кроме того, процедура может взаимодействовать и с внешним миром посредством стандартных процедур READ, WRITE.

Перейдем к рассмотрению примера. Составим программу вычисления площади полной поверхности треугольной пирамиды, если заданы длины шести ее ребер. Условимся, что сначала вводятся длины сторон основания, затем длины боковых ребер. Ясно, что нужно сложить площади четырех треугольных граней. Вычисление площади треугольника по формуле Герона целесообразно опisać в виде функции:

```
PROGRAM ПОЛНАЯ-ПОВЕРХНОСТЬ(INPUT,OUTPUT);
VAR A,B,C,AB,BC,CA,S:REAL;
(* A,B,C,AB,BC,CA — ДЛИНЫ РЕБЕР ПИРАМИДЫ,
   S — ПЛОЩАДЬ ПОВЕРХНОСТИ *)
FUNCTION ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(A,B,C:REAL):REAL;
(* ВНУТРИ ФУНКЦИИ A,B,C — ДЛИНЫ СТОРОН
   ТРЕУГОЛЬНИКА *)
VAR P:REAL;
BEGIN
  P:=(A+B+C)/2;
  ПЛОЩАДЬ-ТРЕУГОЛЬНИКА:=SQRT(P*(P-A)*(P-B)*
    (P-C));
END;
(* КОНЕЦ ОПИСАНИЙ И НАЧАЛО ИСПОЛНЯЕМЫХ
   ИНСТРУКЦИЙ ГЛАВНОЙ ПРОГРАММЫ *)
BEGIN
  READ(A,B,C,AB,BC,CA);
  WRITELN(' ПЛОЩАДЬ ПОЛНОЙ ПОВЕРХНОСТИ');
```



```

WRITELN(' ТРЕУГОЛЬНОЙ ПИРАМИДЫ С РЕБРАМИ:');
WRITELN(' ',A,B,C,AB,BC,CA);
S:=ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(A,B,C)
(* ПЛОЩАДЬ ОСНОВАНИЯ *)
+ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(A,AB,CA)
+ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(B,AB,BC)
+ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(C,CA,BC);
(* ТРИ ПОСЛЕДНИХ СЛАГАЕМЫХ —
  ПЛОЩАДИ БОКОВЫХ ГРАНЕЙ *)
WRITELN(' РАВНА ',S)
END.

```

В данном примере три имени — A, B, C — употреблены и в главной программе, и внутри функции. Тем не менее к путанице это не приведет. Все имена формальных параметров, констант, типов, переменных и процедур, описанные внутри функции (процедуры), вне этой функции (процедуры) недоступны, их как бы не видно. Выбор внутренних имен не влияет на работу процедуры. Внутренние имена называются также локальными. Имена, определенные в главной программе, называются глобальными. Внутри процедуры наряду с локальными доступны все глобальные имена, если только они не совпадают с локальными. В последнем случае локальное имя как бы заслоняет глобальное, для процедуры оно ближе. Если в описание функции ПЛОЩАДЬ-ТРЕУГОЛЬНИКА по неосторожности вставить инструкции присваивания переменным AB, BC, CA, S, значения глобальных переменных изменятся; на глобальные же переменные A, B, C изнутри функции повлиять нельзя.

Проследим, как будут меняться значения переменных по мере выполнения программы. Пусть исходные данные имеют вид

```
3.0000 4.0000 5.0000 4.0000 5.6569 5.0000
```

Эти значения получают глобальные переменные A, B, C, AB, BC, CA после инструкции ввода. После печати исходных данных начнется вычисление площадей. Первое обращение к функции эквивалентно записи

```
ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(3.0000, 4.0000, 5.0000)
```

Эти значения в указанном порядке получают формальные параметры A, B, C. Результатом вычисления функции будет число 6.0000. Затем выполнится обращение к функции

```
ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(3.0000, 4.0000, 5.0000)
```

Результатом также будет 6.0000. Третье обращение, если явно записать значения фактических параметров, будет выглядеть как

```
ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(4.0000, 4.0000, 5.6569)
```

Формальные параметры А, В, С получают эти значения. Глобальные же переменные А, В, С своих значений не изменят. Результатом вызова функции будет число 8.0000. Наконец, последнее обращение можно записать в виде

ПЛОЩАДЬ-ТРЕУГОЛЬНИКА(5.0000, 5.0000, 5.6569)

Результатом будет 11.6620. После сложения четырех чисел значение S составит 31.6620, значения остальных глобальных переменных будут теми же, что и после ввода исходных данных.

Итак, на глобальные переменные, совпадающие по имени с локальными объектами, повлиять нельзя. Однако есть другой вопрос: можно ли, присваивая значения формальным параметрам, изменять значения переменных, являющихся параметрами фактическими? Оказывается, можно, если при описании процедуры в ее заголовке мы явно укажем наше желание. Поставив слово VAR перед именем формального параметра, мы добьемся того, что любое его изменение будет немедленно отражаться на соответствующем фактическом параметре. Разумеется, в этом случае фактический параметр обязан быть переменной. Например, после выполнения программы

```
PROGRAM ПРИМЕР(INPUT,OUTPUT);
VAR X,Y:INTEGER;
PROCEDURE ОБМЕН(VAR M,N:INTEGER);
  VAR T:INTEGER;
  BEGIN
    T:=M; M:=N; N:=T
  END; (* КОНЕЦ ОПИСАНИЯ ПРОЦЕДУРЫ ОБМЕН *)
BEGIN
  X:=2; Y:=3; WRITELN(X,Y);
  ОБМЕН(X,Y); WRITELN(X,Y)
END.
```

будет напечатано

```
2 3
3 2
```

При отсутствии слова VAR в заголовке процедуры значения X и Y после вызова ОБМЕН(X,Y) не изменились бы и программа напечатала бы две одинаковые строки

```
2 3
2 3
```

При отсутствии явных указаний в процедуру передаются копии значений фактических параметров. Мы можем внутри процедуры изменить копии, но вне процедуры заметить это нельзя.

Используем несколько измененную процедуру ОБМЕН для записи программы сортировки трех чисел (см. разд. 2.6):

```
PROGRAM СОРТИРОВКА3(INPUT,OUTPUT);
VAR A,B,C: REAL;
PROCEDURE ОБМЕН(VAR X,Y: REAL);
  VAR T:REAL;
  BEGIN
    T:=X; X:=Y; Y:=T
  END; (* КОНЕЦ ОПИСАНИЯ ПРОЦЕДУРЫ ОБМЕН *)
BEGIN
  WRITELN(
    'СОРТИРОВКА ТРЕХ ЧИСЕЛ ПО ВОЗРАСТАНИЮ');
  READ(A,B,C);
  WRITELN(' ИСХОДНЫЕ ЧИСЛА: ',A,' ',B,' ',C);
  IF A>B THEN ОБМЕН(A,B);
  (* ТЕПЕРЬ A <= B *)
  IF A>C THEN ОБМЕН(A,C);
  (* A <= B, A <= C *)
  IF B>C THEN ОБМЕН(B,C);
  (* A <= B <= C *)
  WRITELN(' РЕЗУЛЬТАТ СОРТИРОВКИ: ',A,' ',B,' ',C)
END.
```

Программа получилась чуть проще и нагляднее, чем первоначальный вариант, поскольку она написана для более умной паскаль-машины, обученной обменивать значения двух переменных.

При передаче массивов как параметров имеет смысл ставить перед именем соответствующего формального параметра слово VAR, даже если мы не собираемся воздействовать на фактический параметр. Делается это, чтобы избежать копирования массива при вызове процедуры. Приведем пример программы, в которой описана процедура, печатающая заданное число первых элементов массива, по пять чисел в строке.

```
PROGRAM ПРИМЕР2(INPUT,OUTPUT);
CONST N=100;
TYPE МАССИВ=ARRAY[1..N] OF REAL;
VAR M:МАССИВ;
PROCEDURE ПЕЧАТЬ-МАССИВА(VAR A:МАССИВ;
  M:INTEGER);
  CONST ДЛИНА-СТРОКИ=5;
  VAR I:INTEGER;
  BEGIN
    FOR I:=1 TO M DO BEGIN
      WRITE(A[I]);
```



```

      IF I MOD ДЛИНА-СТРОКИ=0 THEN WRITELN;
    END;
    IF M MOD ДЛИНА-СТРОКИ < > 0 THEN WRITELN;
  END; (* КОНЕЦ ОПИСАНИЯ ПРОЦЕДУРЫ
        ПЕЧАТЬ-МАССИВА *)
BEGIN
  (* БУДЕМ СЧИТАТЬ, ЧТО В ЭТОМ МЕСТЕ СТОЯТ
    ИНСТРУКЦИИ ЗАПОЛНЕНИЯ МАССИВА M *)
    ПЕЧАТЬ-МАССИВА(M,49)
  END.

```

З а м е ч а н и е 1. Если процедура не имеет параметров, в заголовке, ни при вызове круглые скобки не нужны.

З а м е ч а н и е 2. Если несколько подряд идущих формальных параметров имеют один и тот же тип, имя типа можно указать один раз на всю группу, перечислив перед именем типа параметры через запятую. Так же и слово VAR воздействует на все формальные параметры до ближайшей точки с запятой или закрывающей скобки.

З а м е ч а н и е 3. Выполнение главной программы завершается, когда паскаль-машина доходит до конца программы. Если выполнена последняя инструкция процедуры, происходит возврат в то место программы, откуда процедура была вызвана.

З а м е ч а н и е 4. После выхода из процедуры значения в локальных переменных забываются и в момент повторного вызова будут неопределенными.

З а м е ч а н и е 5. Определяя функцию, мы обязаны приписать ей имени значение-результат. В этом смысле имя функции называется обычной переменной. Однако, если имя функции употреблено в выражении (даже в определении самой функции), это трактуется как вызов функции. Обычно результат накапливают в какой-нибудь переменной, а в конце описания функции присваивают имени нужное значение. Определим для примера функцию, вычисляющую факториал своего целочисленного параметра:

```

FUNCTION ФАКТОРИАЛ(N:INTEGER):INTEGER;
VAR I,K:INTEGER;
BEGIN K:=1;
  FOR I:=2 TO N DO K:=K*I;
  ФАКТОРИАЛ:=K;
END;

```

З а д а ч и

Во всех задачах, где исходными данными являются числа, программа должна вводить и обрабатывать последовательность чисел, оканчивающуюся нулем, выполняя для каждого члена последовательности (кроме нуля) требуемые действия.

1. Разложить заданные числа на простые сомножители.

2. Найти все совершенные числа в диапазоне от 1 до 10 000. (Число называется совершенным, если оно равно сумме своих делителей, включая 1, но исключая само число.)

3. Исходные данные — три числа, задающие дату в следующем виде: год, месяц, число. Напечатать день недели, соответствующий этой дате. Программа должна вводить последовательность троек чисел, признак конца — три нуля.

4. Расположив числа от 1 до 10 000 на плоскости по спирали, отметить звездочками простые числа. Напечатать картинку, содержащую звездочки и число 1.

5. Напечатать заданные числа римскими цифрами.

6. Напечатать заданные натуральные числа в двоичной системе.

7. Вычислить 2^{1000} (решение см. в разд. 3.1).

8. Для заданной суммы денег (в копейках) напечатать способ ее представления минимальным числом купюр и монет.

9. Подсчитать $1 - 1/2 + 1/3 - 1/4 + \dots - 1/10\,000$ четырьмя способами:

а) складывая члены как написано, слева направо;

б) складывая их в обратном порядке;

в) подсчитав отдельно сумму положительных и отрицательных членов слева направо и сложив их;

г) то же, что и в), но складывать в обратном порядке.

10. Для заданного числа S найти первое N , такое что $1 + 1/2 + 1/3 + \dots + 1/N$ превосходит S .

11. Дана паскаль-функция от двух аргументов $F(X, Y)$. Напечатать картинку размером 101×101 , изображающую множество отрицательных значений этой функции. В позиции (i, j) должна стоять звездочка, если $F(X_i, Y_j) < 0$, где $X_i = (i - 51)/50$, $Y_j = (j - 51)/50$, i и j меняются от 1 до 101 (похожая задача разбирается в разд. 3.6).

12. Для заданных m и n подсчитать число сочетаний из n элементов по m . Признак конца последовательности исходных пар m и n — пара нулей.

13. Ввести матрицу $m \times n$, где m и n должны быть заданы в программе константами. Найти в ней седловую точку, т. е. элемент, превосходящий все другие элементы в своей строке и меньший всех других элементов в своем столбце.

14. Найти натуральное число от 1 до 10 000 с максимальной суммой делителей.

15. Ввести последовательность целых чисел и напечатать их в порядке возрастания.

16. Для заданного нечетного n напечатать волшебный квадрат $n \times n$. Волшебным квадратом называется квадратная таблица $n \times n$, заполненная числами от 1 до n^2 , такая что суммы чисел во всех строках, столбцах и двух больших диагоналях совпадают.

3. МЕТОДЫ И ПРИЕМЫ ПРОГРАММИРОВАНИЯ

3.1. Пошаговая разработка программ

Метод пошаговой разработки программ предназначен для упрощения работы с большими программами. Под работой с программой понимается ее составление, обоснование правильности, модификация. Специальные методы работы с большими программами появились вследствие осознания ограниченности мыслительных возможностей человека. Человек в силах охватить разом программу нескольких десятков строк, но попытаться сделать то же с программой из нескольких десятков тысяч строк — это примерно то же самое, что пешеходу попытаться увеличить свою обычную скорость в тысячу раз. Чтобы большие программы были обозримыми, они должны обладать определенной структурой, позволяющей разбиваться в отдельных частях программы независимо от других частей. На достижение такой структуры и направлен метод пошаговой разработки (иногда его называют также пошаговой детализацией программы или программированием сверху вниз).

Идея метода состоит в том, что мы не записываем программу сразу на языке паскаль, а сначала придумываем воображаемого исполнителя, с помощью которого требуемая задача решается просто. Таким образом, мы придумываем одновременно исполнителя и программу для него. Затем мы конструируем новых исполнителей программы для которых объясняют, конкретизируют действия первого исполнителя. Такая цепочка исполнителей, опирающихся друг на друга, строится до тех пор, пока программы для исполнителей самого нижнего уровня не будут записаны на языке паскаль.

Пусть, например, перед нами стоит задача вычислить и печатать большую степень двойки, которая не может быть записана с помощью величины типа INTEGER, скажем 2^{1000} . Вряд ли можно сомневаться в правильности следующей программы решения задачи, при условии что исполнитель понимает все используемые в ней инструкции и описания:

```
PROGRAM СТЕПЕНИ(INPUT,OUTPUT);  
(* ПОДСЧЕТ 2 В СТЕПЕНИ N *)
```



```

VAR N: INTEGER; X: БОЛЬШОЕ ЧИСЛО;
BEGIN READ(N); WRITE(' 2 В СТЕПЕНИ ',N,' = ');
  ВЫЧИСЛИТЬ В ПЕРЕМЕННОЙ X 2 В СТЕПЕНИ N;
  НАПЕЧАТАТЬ БОЛЬШОЕ ЧИСЛО(X);
END.

```

Эту программу частично понимает паскаль-машина. Однако описание типа БОЛЬШОЕ ЧИСЛО и инструкции вычисления 2^N и печати большого числа относятся к другому исполнителю.

Следующий шаг в разработке программы будет состоять в том, чтобы «объяснить» или детализировать инструкции нужного нам исполнителя, записав для них программы, понимаемые некоторым другим исполнителем, в каком-то смысле более близким к паскаль-машине. Затем точно так же объясняется и этот исполнитель и т. д., пока наконец исполнитель самого низкого уровня не будет объяснен в терминах языка паскаль. Таким образом, строится иерархия исполнителей, на самом верху которой находится исполнитель, для которого программа записывается проще всего, а внизу находится паскаль-машина. Каждый исполнитель объясняется посредством исполнителей более низкого уровня.

Прежде чем двигаться по этой цепочке, посмотрим более внимательно на уже написанную программу СТЕПЕНЬ1. Несмотря на тривиальный внешний вид, это настоящая программа. Для того же исполнителя можно составить и другие программы. Если, например, переставить инструкции в программе СТЕПЕНЬ1, то получится неправильная программа. Можно получить и осмысленную программу, если добавить в конце еще одну инструкцию печати большого числа X. Эта программа печатает требуемое число дважды, выполнив вычисления только один раз.

Переход с одного уровня иерархии на более низкий можно рассматривать с разных точек зрения. С одной стороны, мы объясняем сложное действие через более простые, т. е. разбиваем сложную задачу на несколько более простых, которые можно решать относительно независимо. (Например, в первой программе исходная задача разбита на две отдельные подзадачи: вычисление большого числа и его печать. Такой способ действий известен под названием «разделяй и властвуй» и полезен не только в программировании.) С другой стороны, каждое принимаемое нами решение по детализации программы уменьшает класс возможных программ для решения задачи, пока наконец этот класс не сведется к единственной программе на паскале. Так, программу СТЕПЕНЬ1 можно рассматривать как детализацию программы самого высокого уровня

```

PROGRAM СТЕПЕНЬ0(INPUT,OUTPUT);
BEGIN НАПЕЧАТАТЬ ЗАДАННУЮ СТЕПЕНЬ ДВОЙКИ
END.

```

Эта программа никак не ограничивает класс возможных решений. Мы еще не знаем, каким образом мы будем печатать степень двойки. Переходя к программе СТЕПЕНЬ4, мы уже решили, что нужно сначала вычислить целиком и запомнить требуемое число, а затем напечатать его. Тем самым отвергаются, например, все программы, которые вычисляют и печатают степень двойки по одной цифре, и храня все число целиком.

Перейдем к детализации программы СТЕПЕНЬ4. Необходимо объяснить три вещи: описание типа БОЛЬШОЕ ЧИСЛО, инструкции вычисления 2^N и печати большого числа. Возникает вопрос с чего начать? Этот вопрос на самом деле довольно сложный и творческий. Общие рекомендации таковы: сначала следует принимать такие решения, которые по возможности слабо зависят от еще принятых решений в других частях программы; кроме того, следует выбирать простые решения с максимальными шансами на то, что их не придется потом пересматривать. В данном случае начать следует, видимо, с инструкции вычисления 2^N . Действительно, способ вычисления 2^N определяет нужные нам операции с большими числами. Набор допустимых операций — вот то главное, что характеризует тип данных; представление значений выбирается из соображений простоты и эффективности реализации операций. После того как будет выбрано представление значений типа БОЛЬШОЕ ЧИСЛО, следует браться за детализацию инструкции печати большого числа.

Итак, вычислим 2^N . Будем делать это путем последовательных умножений на 2. Введем вспомогательную переменную K и выберем в качестве инварианта цикла условие $X = (2 \text{ в степени } K)$. Ниже приведен фрагмент программы, реализующий инструкцию вычисления 2^N :

ВЫЧИСЛИТЬ В ПЕРЕМЕННОЙ X 2 В СТЕПЕНИ N:

```
VAR K:INTEGER;  
X:=1; (* X=2 В СТЕПЕНИ 0 *)  
FOR K:=1 TO N DO BEGIN  
    УДВОИТЬ(X); (* X=2 В СТЕПЕНИ K *)  
END;  
(* X=2 В СТЕПЕНИ N *)
```

В этой программе две инструкции требуют дальнейшей детализации: УДВОИТЬ(X) и $X:=1$. (Последняя инструкция не может непосредственно выполняться паскаль-машиной, поскольку паскаль-машина не понимает тип переменной X — БОЛЬШОЕ ЧИСЛО.)

Дальнейшее продвижение, видимо, невозможно без детализации типа БОЛЬШОЕ ЧИСЛО. Заметим, что если реализовать БОЛЬШОЕ ЧИСЛО просто как тип INTEGER, то инструкции удво-

на печати сведутся к инструкциям паскаля и мы получим обычную программу возведения 2 в степень N, пригодную для небольших N. Нас это не устраивает. Чтобы хранить очень большие числа, их можно записывать в виде массива, каждый элемент которого будет хранить одну цифру числа. Так, для записи числа 8192 используем четыре элемента массива, в которые поместим цифры 8, 1, 9, 2.

Здесь, однако, встает еще одна проблема. В ходе работы программе придется иметь дело с различными большими числами, содержащими разное количество цифр. Массив же должен иметь постоянную длину. Поэтому длину массива следует выбирать из расчета на максимальное число цифр. Кроме того, следует договориться, какие элементы массива будут хранить цифры числа, если его длина меньше размера массива.

В ручном алгоритме умножения столбиком младшие цифры произведения записываются под младшими цифрами сомножителей, тогда как произведение растет влево. Это наводит на мысль записать число в массиве аналогично, дав ему возможность расти влево, в сторону старших цифр. Для этого запишем цифру единиц в последний элемент массива, цифру десятков — в предпоследний и т. д.

Программа также должна знать, сколько цифр имеется в числе. Эту информацию можно сохранить, например, помня номер элемента массива, в котором начинается число. Итак, представление большого числа состоит из двух частей: массива его цифр и номера элемента массива, с которого начинаются цифры. Поскольку мы не умеем (пока) объединять в одной переменной разнородные компоненты, нам придется детализировать описание переменной X посредством двух переменных:

X:БОЛЬШОЕ ЧИСЛО:

```
CONST МАКСДЛИНА=302; (* ЧИСЛО ЦИФР,  
    ДОСТАТОЧНОЕ ДЛЯ ХРАНЕНИЯ 2 В СТЕПЕНИ 1000 *)  
VAR ХЦИФРЫ:ARRAY[1..МАКСДЛИНА] OF 0..9;  
    ХНАЧАЛО:1..МАКСДЛИНА;  
    (* ЦИФРЫ ЧИСЛА—ХЦИФРЫ[ХНАЧАЛО..  
        МАКСДЛИНА] *)
```

Столь подробное описание детализации представления больших чисел дано потому, что оно — ключевой момент для данной программы. После выбора представления больших чисел дальнейшее программирование разбивается на совершенно независимые части по реализации отдельных действий с большими числами. Возможны другие представления больших чисел. Вероятно, было бы удобнее, хотя это и менее естественно, записывать цифру единиц первый элемент массива, цифру десятков — во второй и т. д.

Перейдем к детализации инструкций для работы с большими числами. Проще всего реализовать присваивание переменной начального значения 1. Для этого следует присвоить 1 цифре единицы и указать в переменной ХНАЧАЛО, что число Х содержит одну цифру:

X:=1;

ХЦИФРЫ[МАКСДЛИНА]:=1;

ХНАЧАЛО:=МАКСДЛИНА;

Следующая инструкция — удвоение Х. Будем использовать ручной алгоритм умножения столбиком, в котором производится строится по одной цифре, начиная с младшей. Для указания позиции текущей обрабатываемой цифры введем переменную I. Получится также переменная для запоминания величины переноса:

УДВОИТЬ(X):

VAR I:1..МАКСДЛИНА;

(* I — НОМЕР ОБРАБАТЫВАЕМОЙ ЦИФРЫ *)

ПЕРЕНОС:INTEGER;

FOR I:=МАКСДЛИНА DOWNTO ХНАЧАЛО DO BEGIN

УДВОИТЬ ЦИФРУ ХЦИФРЫ[I];

ПРИБАВИТЬ ПЕРЕНОС ИЗ ПРЕДЫДУЩЕГО РАЗРЯДА

ЗАПИСАТЬ ЦИФРУ ЕДИНИЦ СУММЫ В ХЦИФРЫ [I];

ЗАПОМНИТЬ ПЕРЕНОС;

END;

IF ПЕРЕНОС<>0 THEN

BEGIN ХНАЧАЛО:=ХНАЧАЛО-1;

ХЦИФРЫ[ХНАЧАЛО]:=ПЕРЕНОС END;

Реализация использованных в этом фрагменте инструкций довольно проста. Они связаны между собой использованием общепринятых переменных. Это, по-первых, переменная ПЕРЕНОС и, во-вторых, вспомогательная переменная (назовем ее V), в которой будет храниться значение удвоенной цифры. Единственный, пожалуй, подводный камень встретится при реализации инструкции ПРИБАВИТЬ ПЕРЕНОС ИЗ ПРЕДЫДУЩЕГО РАЗРЯДА. Сложнее всего в том, чтобы эта инструкция правильно работала с самым младшим разрядом, когда предыдущего разряда не существует и прибавить к нему ничего не надо. Можно, конечно, записать условную инструкцию, которая по значению I определяет, какой разряд обрабатывается: самый младший или нет. Более изящное решение состоит в том, чтобы до цикла присвоить переменной ПЕРЕНОС значение 0. Тогда прибавление на первом шаге такого «переноса» не изменит значения. Перепишем полностью предыдущий фрагмент, раскрыв в нем инструкции:

УДВОИТЬ(X):

```
VAR I:1..МАКСДЛИНА;  
  (* I—НОМЕР ОБРАБАТЫВАЕМОЙ ЦИФРЫ *)  
  ПЕРЕНОС,В:INTEGER;  
  ПЕРЕНОС:=0;  
FOR I:=МАКСДЛИНА DOWNTО ХНАЧАЛО DO BEGIN  
  (* УДВОИТЬ ЦИФРУ ХЦИФРЫ[I] *)  
  В:=2*ХЦИФРЫ[I];  
  (* ПРИБАВИТЬ ПЕРЕНОС ИЗ ПРЕДЫДУЩЕГО РАЗРЯДА *)  
  В:=В+ПЕРЕНОС;  
  (* ЗАПИСАТЬ ЦИФРУ ЕДИНИЦ СУММЫ В ХЦИФРЫ[I] *)  
  ХЦИФРЫ[I]:=В MOD 10;  
  (* ЗАПОМНИТЬ ПЕРЕНОС *)  
  ПЕРЕНОС:=В DIV 10;  
END;  
IF ПЕРЕНОС<>0 THEN  
BEGIN ХНАЧАЛО:=ХНАЧАЛО-1;  
  ХЦИФРЫ[ХНАЧАЛО]:=ПЕРЕНОС END;
```

Теперь осталось реализовать инструкцию печати числа X. Чтобы напечатать большое число, нужно последовательно напечатать его цифры, хранящиеся в массиве ХЦИФРЫ. Мы уже встречались с подобными задачами последовательной обработки элементов массива:

НАПЕЧАТАТЬ БОЛЬШОЕ ЧИСЛО(X):

```
VAR I:1..МАКСДЛИНА;  
FOR I:=ХНАЧАЛО TO МАКСДЛИНА DO  
  WRITE(ХЦИФРЫ[I]:1);  
WRITELN;
```

Обратим внимание на добавление :1 в записи параметра процедуры WRITE. Такая запись означает, что мы хотим напечатать число ХЦИФРЫ[I], отводя для него на бумаге одну позицию. Если не указывать число позиций (как мы делали до сих пор), под печатаемое число будет отводиться некоторое стандартное число позиций (при печати целых чисел — десять). Так, инструкция WRITE(1,2,3) напечатает

1 2 3

WRITE(1:1,2:1,3:1) напечатает

В общем случае параметры процедуры WRITE должны иметь одну из следующих форм:

e

$e:e_1$

$e:e_1:e_2$

где e — печатаемое выражение, e_1 — выражение типа INTEGER задающее число позиций для печати e , e_2 можно использовать только при печати вещественных чисел; это выражение задает число цифр печатаемых после десятичной точки.

Соберем все, что получилось, в одну программу. Это можно сделать двумя способами. С одной стороны, можно воспользоваться механизмом процедур, который как раз и позволяет обучить каль-машину тем инструкциям, которые она изначально не понимает. В этом случае текст программы для воображаемого исполнителя практически без изменений переносится в главную программу. Инструкции исполнителя становятся вызовами процедур, объяснения же этих инструкций составляют тела процедур. С другой стороны, можно просто заменить инструкции исполнителей верхнего уровня на последовательности инструкций, объясняющие их. В последнем случае исходные инструкции обязательно следует вставить в программу в виде комментариев, так как они хорошо поясняют смысл выполняемых программой действий. Мы используем оба приема, записав вычисление 2^N в виде процедуры, а все остальные действия подставив непосредственно в программу:

```
PROGRAM СТЕПЕНЬ2(INPUT,OUTPUT);
CONST МАКСДЛИНА=302; (* ЧИСЛО ЦИФР, ДОСТАТОЧНОЕ
    ДЛЯ ХРАНЕНИЯ 2 В СТЕПЕНИ 1000 *)
VAR (* X: БОЛЬШОЕ ЧИСЛО *)
    ХЦИФРЫ: ARRAY[1..МАКСДЛИНА] OF 0..9;
    ХНАЧАЛО: 1..МАКСДЛИНА;
    N: INTEGER; (* N — ПОКАЗАТЕЛЬ СТЕПЕНИ *)
    I: 1..МАКСДЛИНА;
PROCEDURE ВЫЧИСЛИТЬ-2-В-СТЕПЕНИ(N: INTEGER);
VAR I: 1..МАКСДЛИНА; K: INTEGER;
    ПЕРЕНОС,В: INTEGER;
BEGIN
    (* X:=1 *)
    ХЦИФРЫ[МАКСДЛИНА]:=1; ХНАЧАЛО:=МАКСДЛИНА;
    (* X=2 В СТЕПЕНИ 0 *)
    FOR K:=1 TO N DO BEGIN
        (* УДВОИТЬ X *)
        ПЕРЕНОС:=0;
        FOR I:=МАКСДЛИНА DOWNTO ХНАЧАЛО DO BEGIN
```



```

(* УДВОИТЬ ЦИФРУ ХЦИФРЫ[I] *)
V:=2*ХЦИФРЫ[I];
(* ПРИБАВИТЬ ПЕРЕНОС ИЗ ПРЕДЫДУЩЕГО
РАЗРЯДА *)
V:=V+ПЕРЕНОС;
(* ЗАПИСАТЬ ЦИФРУ ЕДИНИЦ СУММЫ В
ХЦИФРЫ [I] *)
ХЦИФРЫ[I]:=V MOD 10;
(* ЗАПОМНИТЬ ПЕРЕНОС *)
ПЕРЕНОС:=V DIV 10;
END;
IF ПЕРЕНОС<>0 THEN
BEGIN ХНАЧАЛО:=ХНАЧАЛО-1;
ХЦИФРЫ[ХНАЧАЛО]:=ПЕРЕНОС
END;
(* X=2 В СТЕПЕНИ K *)
END;
(* X=2 В СТЕПЕНИ N *)
END;
BEGIN (* ГЛАВНАЯ ПРОГРАММА *)
READ(N); WRITE(' 2 В СТЕПЕНИ ',N,' = ');
ВЫЧИСЛИТЬ-2-В-СТЕПЕНИ(N);
(* НАПЕЧАТАТЬ БОЛЬШОЕ ЧИСЛО(X) *)
FOR I:=ХНАЧАЛО TO МАКСДЛИНА
DO WRITE(ХЦИФРЫ[I];
WRITELN;
END.

```

После выполнения приведенной программы будет напечатано

```

2 В СТЕПЕНИ 1000=10715086071862673209484250490600018105614
048117055336074437503883703510511249361224931983788156958581275
946729175531468251871452856923140435984577574698574803934567774
824230985421074605062371141877954182153046474983581941267398767
559165543946077062914571196477686542167660429831652624386837205
668069376

```

В качестве дополнительной литературы по этой теме рекомендуем книгу [9].

3.2. Анализ алгоритмов

В этом разделе мы рассмотрим анализ алгоритмов с точки зрения времени их работы. Разработка эффективных, быстро работающих алгоритмов имеет большое, хотя и не первостепенное значение при решении практических задач. Для выбора самого быстрого из не-

скольких алгоритмов желательно научиться сравнивать времена работы. Однако точно вычислить время работы алгоритма оказывается невозможным. Это время помимо всего прочего зависит от характеристик реальной ЭВМ, на которой выполняется программа. Обычно время выполнения алгоритма вычисляют только приближенно. Более того, формулы для времени работы обычно содержат некоторые числовые константы, значения которых можно определить лишь опытным путем. Даже такие оценки позволяют довольно часто выбрать один наилучший алгоритм. В других же случаях с помощью оценок времени работы можно отбросить заведомо медленные алгоритмы, а лучший из оставшихся выбрать с помощью измерения фактического времени работы программ.

Поясним сказанное примером. Пусть нужно выполнять некоторую обработку последовательности из n чисел, где n может меняться (скажем, нужно найти минимальное из этих чисел). Пусть у нас есть две программы для решения задачи, времена работы которых приблизительно равны $C_1 n$ и $C_2 n^2$, где C_1 и C_2 — константы, не зависящие от n . Ясно, что при больших n первая программа будет работать быстрее. Скорее всего, при малых n она будет работать медленнее (именно так обычно бывает в практических случаях), а это означает, что $C_1 > C_2$. Пусть C_1 в пять раз больше C_2 : $C_1 = 5C_2$. Сравнив выражения для времен работы $C_1 n = 5C_2 n$ и $C_2 n^2$, можно увидеть, что при $n > 5$ время работы первой программы меньше, а при $n < 5$ быстрее вторая программа. Разумеется, ввиду приближенности оценок времени граничное значение $n = 5$ тоже устанавливается приближенно.

Из наших гипотетических рассуждений следуют два вывода. Первый вывод: даже приближенные оценки «с точностью до константы» могут оказаться полезными при сравнении алгоритмов. Второй вывод: обычно нельзя сказать, что какой-то один алгоритм решения задачи всегда самый быстрый. Некоторый алгоритм может превосходить другой для какого-то одного набора исходных данных и уступать ему для другого набора.

Остановимся теперь на вопросе о выборе такой характеристики исходных данных, которая определяла бы время работы программы. Эта характеристика называется размером задачи. Для разных задач на время решения влияют разные свойства исходных данных. Для задач типа поиска минимума или упорядочения заданных чисел, где каждое число рассматривается как единое целое, размером задачи является количество исходных чисел. Для задачи разложения на простые сомножители размером естественно считать величину числа. Выбор размера задачи основан на некотором, хотя бы грубом представлении об алгоритме решения. Обратимся к задаче вычисления 2^N , разобранный в предыдущем разделе. Учитывая, что возведение в степень будет выполняться путем последовательного умножения,

ния, выберем в качестве размера показатель степени N . Другим возможным кандидатом на роль размера является число цифр в результате. В нашей задаче число цифр в числе 2^N практически пропорционально N , поэтому оба размера эквивалентны.

Следующий важный момент при оценке времени работы — отыскание тех частей программы, выполнение которых занимает основную долю всего времени работы. Известен экспериментальный факт, справедливый для подавляющего большинства программ: основная часть времени работы тратится на выполнение очень небольшой части текста программы. Такая часть программы называется внутренним циклом (как правило, это действительно цикл в смысле языка паскаль). Мы с самого начала требуем лишь приближенной оценки времени работы, поэтому можно пренебречь временем выполнения всей остальной части программы и заняться только внутренним циклом, что сэкономит наши усилия.

Если программа представляет собой несколько вложенных друг в друга циклов, то внутренний цикл найти легко. Действительно, чаще всего будут выполняться те части программы, которые принадлежат всем циклам, а это как раз циклы, лежащие внутри всех остальных и не содержащие внутри себя других циклов. Это наблюдение оправдывает название «внутренний цикл» для наиболее часто выполняемых частей программы.

Если имеется несколько циклов, не содержащих внутри себя других, следует вычислить число повторений каждого из них. Если число повторений одного цикла окажется много больше, чем других, то временем работы этих других циклов можно пренебречь; в противном случае в окончательной формуле для времени работы придется учесть времена выполнения нескольких циклов. Число повторений цикла типа FOR определяется по следующей формуле:

для цикла $\text{FOR } I := A \text{ TO } B$
число повторений равно $B - A + 1$, если $B \geq A - 1$
и равно нулю, если $B < A$ (1)
для цикла $\text{FOR } I := A \text{ DOWNTO } B$
в формуле следует поменять местами A и B

Эта формула справедлива, если программа приступает к выполнению цикла всего один раз. Рассмотрим, как быть, если этот цикл вложен в другой и его выполнение повторяется много раз. Договоримся о терминологии. Когда нам понадобится выяснить число повторений инструкций внутри цикла, будем называть это число *числом выполнений тела цикла*. Если же речь пойдет о том, сколько раз начинает выполняться весь цикл, будем использовать термин *число выполнений заголовка цикла*. Таким образом, одно выполнение заголовка цикла предписывает выполнить тело несколько раз. Формула (1) даст не что иное, как число выполнений тела цикла при

однократном выполнении его заголовка. Если рассматриваемый цикл вложен в другой, его заголовок будет выполняться многократно, и, чтобы найти общее число выполнений тела, следует просуммировать числа, даваемые формулой (1), при всех выполнениях заголовка этого цикла.

В качестве простого примера применения этой формулы проанализируем следующий фрагмент программы:

```
FOR J:=1 TO N DO BEGIN (* ЦИКЛ 1 *)
  FOR K:=1 TO M DO ТЕЛО ЦИКЛА 2;
  FOR L:=N DOWNT0 J DO ТЕЛО ЦИКЛА 3;
END;
```

Тела циклов 2 и 3 не содержат циклов, M и N — некоторые достаточно большие константы. Оценим число повторений циклов 2 и 3. Оба они вложены в цикл 1, число повторений которого легко находится по формуле (1); оно равно $N-1+1=N$. При каждом из этих повторений тело цикла 2 выполняется $M-1+1=M$ раз. Поскольку M — константа, общее число выполнений тела цикла 2 равно $N_2 = N \cdot M$. Тело цикла 3 выполняется $N-J+1$ раз при каждом выполнении тела цикла 1. Это число в отличие от M меняется при выполнении цикла 1; поэтому просто перемножить два числа: необходимо суммировать все числа выполнений тела цикла 3. Заметим, что при выполнении цикла 1 переменная J принимает все значения от 1 до N (именно это и записано в заголовке цикла 1). Таким образом, общее число выполнений тела цикла 3 равно

$$N_3 = \sum_{J=1}^N (N - J + 1).$$

Это сумма арифметической прогрессии; она равна $(N+1) \cdot N/2$. Поскольку мы все равно вычисляем время приблизительно, то, пренебрегая 1 по сравнению с N, несколько упростим формулу: N_3 приблизительно равно $N^2/2$. Сравнивая N_2 и N_3 , мы не можем заключить, что одно из этих чисел значительно превосходит другое. Например, при некоторых M и N они будут одного порядка. Например, если $M=N$, то N_2 примерно в 2 раза больше N_3 . Поэтому в формулу для времени работы придется включить два слагаемых, отвечающих обоим циклам. Считая времена однократного выполнения тел циклов примерно постоянными и обозначив их T_2 и T_3 , получаем следующую приближенную формулу для общего времени работы:

$$T = M \cdot N \cdot T_2 + (N^2/2) \cdot T_3.$$

Если времена выполнения тел циклов нельзя считать постоянными (например, когда в них есть условные инструкции или вызовы процедур), нужно сначала проанализировать времена выполнения

тел, а затем просуммировать их (вместо умножения постоянного времени на число повторений).

Более детальную информацию о времени работы можно получить по результатам измерения фактического времени работы программы для различных исходных данных. С помощью этих экспериментальных данных можно оценить неизвестные постоянные, входящие в формулу (в нашем случае T_2 и T_3).

Перейдем к анализу конкретной программы — программы вычисления 2^N из предыдущего раздела. Работа программы состоит из двух крупных частей, включающих циклы. Это — вычисление 2^N и печать результата. При оценке времени работы программы обычно интересуются временем собственно вычислений, а не временем печати. Это объясняется тем, что время печати для многих возможных алгоритмов решения задачи будет одним и тем же, поскольку напечатать нужно одну и ту же информацию. Кроме того, в данном случае, как и во многих практических задачах, временем печати можно пренебречь, поскольку оно значительно меньше времени вычислений при достаточно больших N . Отбросив печать, а также все, что не лежит в циклах, получим такую программу:

```
FOR K:=1 TO N DO BEGIN (* ЦИКЛ 1 *)
  FOR I:=МАКСДЛИНА DOWNTO ХНАЧАЛО DO BEGIN
    (* ЦИКЛ 2 *)
    ДЕЙСТВИЯ, НЕ СОДЕРЖАЩИЕ ЦИКЛОВ
  END;
  ДЕЙСТВИЯ, НЕ СОДЕРЖАЩИЕ ЦИКЛОВ
END;
```

Один цикл, а именно цикл 2, не содержит других циклов. Он и будет внутренним циклом. Оценим число повторений его тела.

Число выполнений тела цикла 2, равное $\text{МАКСДЛИНА} - \text{ХНАЧАЛО} + 1$, меняется в ходе работы из-за изменения значения ХНАЧАЛО . Подобный случай встречался в рассмотренной выше программе; здесь, однако, все усложняется из-за того, что мы не знаем, как зависит значение ХНАЧАЛО от значения K — управляющей переменной цикла 1. Обозначив число выполнений тела цикла 2 через $N_2(K)$, приходим к необходимости подсчета такой суммы:

$$N_2 = \sum_{K=1}^N N_2(K).$$

Для нахождения функции $N_2(K)$ необходим содержательный анализ алгоритма. Применять непосредственно формулу (1) неудобно. Вместо этого заметим, что при каждом выполнении заголовка цикла 2 число выполнений его тела равно числу цифр в удваиваемом числе. Удваиваемое число, как легко увидеть из текста программы, равно 2^{K-1} . Число цифр в любом натуральном числе равно увели-

ченной на 1 целой части его десятичного логарифма, а десятичный логарифм 2^{K-1} равен $(K-1) \cdot \lg 2$, где $\lg 2$ примерно равен 0.3. Приходим, таким образом, к следующей формуле для $N_2(K)$:

$$N_2(K) = 1 + (K-1) \cdot \lg 2.$$

Точно вычислить сумму с таким общим членом достаточно сложно. Чтобы упростить подсчет, перейдем к приближенным числам. Заметим, что целая часть числа отличается от самого числа менее чем на 1. Пренебрегая этой разницей (по сравнению с достаточно большим на протяжении основной части программы числом цифр), заменим целую часть логарифма на сам логарифм. Теперь нужно суммировать арифметическую прогрессию с общим членом

$$1 + (K-1) \cdot \lg 2.$$

Сумма дается формулой

$$\sum_{K=1}^N (1 + (K-1) \lg 2) = \frac{(1 + 1 + (N-1) \lg 2) \cdot N}{2} \approx \frac{N^2 \cdot \lg 2}{2}.$$

При переходе к последней величине мы пренебрегли слагаемым $2 - \lg 2$, которое мало в сравнении с $N \cdot \lg 2$. Можно было бы и сразу получить окончательный результат, если заметить, что $1 + (K-1) \lg 2$ можно приближенно заменить на $K \cdot \lg 2$, а в формуле для суммы арифметической прогрессии можно пренебречь первым членом прогрессии. Погрешность при всех этих заменах будет одного порядка и составит около $1/N$ части окончательного результата.

Итак, суммарное время выполнения цикла 2, которое, как мы выяснили, составляет основную часть времени вычислений во всей программе, будет равно $N^2 \cdot \lg 2 / 2 \cdot T_2 \approx 0.15 \cdot N^2 \cdot T_2$, где T_2 — время однократного выполнения тела цикла.

Обратим внимание на важный момент. Мы провели оценку времени работы программы вычисления 2^N , не опираясь на ее окончательный текст. Прикидочные оценки времени работы можно и нужно делать еще на стадии разработки программ, чтобы как можно раньше отбросить заведомо медленные варианты.

Рассмотрим теперь пример анализа времени работы алгоритма с циклами типа WHILE-DO. Проанализируем время работы функции возведения целого числа в небольшую натуральную степень. Эта функция легко получается из программы разд. 2.10:

```
FUNCTION БЫСТРАЯ-СТЕПЕНЬ (M, N: INTEGER): INTEGER;
(* M — ОСНОВАНИЕ СТЕПЕНИ, N — ПОКАЗАТЕЛЬ *)
VAR M1, N1, P: INTEGER;
(* P — ПЕРЕМЕННАЯ ДЛЯ НАКОПЛЕНИЯ РЕЗУЛЬТАТА *)
BEGIN
```



```

P:=1; M1:=M; N1:=N;
WHILE N1<>0 DO BEGIN (* ВНЕШНИЙ ЦИКЛ *)
  WHILE N1 MOD 2=0 DO BEGIN (* ВНУТРЕННИЙ
                                ЦИКЛ *)
    N1:=N1 DIV 2; M1:=M1*M1
    (* ЗНАЧЕНИЕ (M1 В СТЕПЕНИ N1) НЕ ИЗМЕНИЛОСЬ
    *)
  END; (* КОНЕЦ ВНУТРЕННЕГО ЦИКЛА *)
  N1:=N1-1; P:=P*M1
  (* P * (M1 В СТЕПЕНИ N1)=(M В СТЕПЕНИ N) *)
END; (* КОНЕЦ ВНЕШНЕГО ЦИКЛА *)
(* P=(M В СТЕПЕНИ N) *)
БЫСТРАЯ-СТЕПЕНЬ:=P;
END;

```

Размером задачи будет служить значение показателя степени N . Ясно, что основная часть времени тратится на выполнение циклов, однако нельзя утверждать, что время выполнения тела внутреннего цикла существенно превосходит время выполнения цикла внешнего, поскольку при нечетных N_1 тело внутреннего цикла выполняться не будет. Обозначим через T_1 время однократного выполнения тела внешнего цикла (без инструкций внутреннего цикла), K_1 — число выполнений тела внешнего цикла. T_2 и K_2 — соответствующие характеристики внутреннего цикла. Общее время $T(N)$ работы функции приблизительно равно $T(N)=K_1 \cdot T_1 + K_2 \cdot T_2$.

Чтобы оценить K_1 и K_2 , проследим за процессом уменьшения N_1 . Функция БЫСТРАЯ-СТЕПЕНЬ включает операции проверки четности N_1 и деления на 2. Это наводит на мысль, что для анализа данного алгоритма целесообразно рассмотреть запись значения N_1 в двоичной системе счисления. (Напомним, что число A , записанное в позиционной системе счисления с основанием b цифрами $a_m a_{m-1} \dots a_1 a_0$, равно

$$A = \sum_{i=0}^m a_i b^i.$$

При $b=2$ получаем двоичную запись числа цифрами 0 и 1. Так, двоичное число 1001 равно 9.) Для нахождения K_1 и K_2 подсчитаем, сколько раз выполняются внешний и внутренний циклы при обработке одной, младшей двоичной цифры в значении N_1 . Возможны три случая.

1. $N_1=1$. Внутренний цикл выполняться не будет, а внешний выполнится один раз; N_1 уменьшится до 0, после чего произойдет выход из цикла.

2. $N_1>1$, N_1 оканчивается на 0 (т. е. четно). После однократного выполнения внутреннего цикла N_1 поделится пополам. Деление

на 2 в двоичной системе счисления равносильно отбрасыванию младшей цифры. Следовательно, для обработки младшего нуля достаточно одного выполнения внутреннего цикла.

3. $N1 > 1$, $N1$ оканчивается на 1 (нечетно). Внутренний цикл не будет выполняться, во внешнем из $N1$ вычтется 1, т. е. $N1$ станет четным, и мы приходим к случаю 2. Следовательно, для обработки младшей единицы требуется одно выполнение внешнего цикла и одно выполнение внутреннего цикла.

Так одна за другой будут обработаны и отброшены все двоичные цифры $N1$. Следовательно, учитывая начальную установку $N1 := N$, имеем:

$K_1 =$ число единиц в двоичной записи N ,

$K_2 =$ (число цифр в двоичной записи N) $- 1$.

Число цифр в двоичной записи N равно увеличенной на 1 целой части двоичного логарифма N , т. е.

$$K_2 = \lceil \log_2 N \rceil.$$

Написать столь же простую формулу для K_1 не удастся, поэтому оценим K_1 сверху. Ясно, что число единиц не больше числа цифр, т. е.

$$K_1 \leq 1 + \lceil \log_2 N \rceil.$$

Отсюда

$$T(N) \leq (1 + \lceil \log_2 N \rceil) \cdot T_1 + \lceil \log_2 N \rceil \cdot T_2 = (T_1 + T_2) \cdot \lceil \log_2 N \rceil + T_1.$$

Равенство достигается, когда N записывается в двоичной системе только единицами, т. е. на 1 меньше степени двойки. Чем меньше единиц в двоичной записи N , тем быстрее работает функция БЫСТРАЯ-СТЕПЕНЬ. Если N — степень двойки, K_1 достигает минимума, равного 1, а общее время составляет

$$T(N) = T_1 + \log_2 N \cdot T_2.$$

Сопоставим теперь функцию БЫСТРАЯ-СТЕПЕНЬ с прямолинейной:

```
FUNCTION ПРЯМОЛИНЕЙНАЯ (M,N: INTEGER): INTEGER;
(* M — ОСНОВАНИЕ СТЕПЕНИ, N — ПОКАЗАТЕЛЬ *)
VAR M1,N1,P: INTEGER;
(* P — ПЕРЕМЕННАЯ ДЛЯ НАКОПЛЕНИЯ РЕЗУЛЬТАТА *)
BEGIN
  P:=1; M1:=M; N1:=N;
  WHILE N1<>0 DO BEGIN
    N1:=N1-1; P:=P*M1;
    (* P *(M1 В СТЕПЕНИ N1)=(M В СТЕПЕНИ N) *)
  END;
```

(* P = (M В СТЕПЕНИ N) *)

ПРЯМОЛИНЕЙНАЯ: = P;

END;

Обозначим время работы этой функции через $ТП(N)$, время однократного выполнения цикла через $ТП_1$. Имеем

$$ТП(N) \approx N \cdot ТП_1.$$

Поскольку $\log_2 N$ растет медленнее, чем N , то при больших N $Т(N) < ТП(N)$, и, чем больше N , тем заметнее будет выигрыш в скорости. Выясним, начиная с каких N функция БЫСТРАЯ-СТЕПЕНЬ работает быстрее. Можно считать, что

$$ТП_1 \approx Т_1 \approx Т_2,$$

поскольку основное время в каждом из циклов тратится на умножение. Отсюда

$$Т(N) \leq ТП_1 \cdot (2 \cdot [\log_2 N] + 1).$$

Решая неравенство

$$2 \cdot [\log_2 N] + 1 < N,$$

находим, что $N > 5$. При поиске порогового значения N мы пользовались оценкой сверху для $Т(N)$. Значения N от 1 до 5 можно исследовать отдельно. При $N=1, 2, 3$ функция БЫСТРАЯ-СТЕПЕНЬ совершит столько же умножений, как и ПРЯМОЛИНЕЙНАЯ, а при $N=4, 5$ — меньше. Практически это означает, что функцию БЫСТРАЯ-СТЕПЕНЬ целесообразно использовать при всех N .

Подведем итоги. Мы видим, что если число повторений FOR-цикла переменнo и вычисляется в программе, то оценка времени работы не получается формальными методами, а требует содержательного анализа работы программы. Еще большие сложности вызывают циклы типа WHILE—DO и REPEAT—UNTIL, для которых вообще нельзя записать общую формулу числа повторений. Заметим также, что далеко не всегда удается преодолеть все сложности и получить хотя бы приближенную формулу для времени работы. Иногда приходится довольствоваться более грубой информацией. Такая ситуация имеет место, например, для алгоритма разложения числа на простые сомножители, основанного на переборе всех возможных сомножителей. Ясно, что, если повезет, мы можем сразу наткнуться на нужный сомножитель и быстро разложить даже очень большое число. Вместе с тем, если разлагаемое число простое, то перебор придется довести до конца, прежде чем можно будет удостовериться, что число действительно простое. Для этого алгоритма можно ставить задачу нахождения верхней оценки времени работы, т. е. такой величины $ТМАХ(N)$, что фактическое вре-

ми работы для задачи размера N не превосходит $TMAX(N)$ и иногда равно этой величине.

Помимо сравнения алгоритмов формулы для времени работы можно использовать для экстраполяции измеренного времени работы на другой размер задачи.

Пример. Пусть нужно вычислить $2^{10\,000}$ с помощью разобранной в разд. 3.1 программы. Можно заранее сказать, что это займет много времени. Чтобы узнать, сколько именно, вычислим 2^{1000} и измерим время вычислений. Допустим, оно оказалось равным 30 с. Тогда для $2^{10\,000}$ время вычислений будет в $10 \cdot 10 = 100$ раз больше и составит 3000 с, т. е. 50 мин. Это означает, что для вычисления $2^{10\,000}$ следует подыскать более быстрый алгоритм.

Тема для исследования. Оценить время работы программы для вычисления 2 в большой степени N , в которой числа, как и раньше, хранятся в виде массивов цифр, но вычисления выполняются, как в функции БЫСТРАЯ-СТЕПЕНЬ.

Указание. Логика работы этой программы наиболее проста, если показатель степени N сам является степенью двойки. Целесообразно начать исследование времени работы с этого частного случая.

3.3. Обработка рекуррентных последовательностей

Говорят, что элементы $U(1), U(2), \dots, U(N), \dots$ образуют рекуррентную последовательность K -го порядка, если заданы первые K членов последовательности $U(1), \dots, U(K)$, а для $N > K$

$$U(N) = F(U(N-1), \dots, U(N-K)),$$

где F — некоторая функция K аргументов. Известным примером рекуррентной последовательности второго порядка является последовательность чисел Фибоначчи, для которой $U(1) = U(2) = 1$, а для $N > 2$

$$U(N) = U(N-1) + U(N-2).$$

Пример рекуррентной последовательности первого порядка — последовательность для вычисления корня квадратного из числа a по методу Ньютона: $U(1) = a + 1$, а для $N > 1$

$$U(N) = 0.5(U(N-1) + a/U(N-1)).$$

Применительно к рекуррентным последовательностям часто ставятся следующие задачи.

1. Вычислить N -й элемент последовательности.
2. Вычислить элемент с минимальным номером, делающий истинной заданную логическую функцию $G(U(N), U(N-1), \dots, U(N-K))$.

Для решения обеих задач достаточно хранить в памяти $K+1$ элементов последовательности. Пусть $K1=K+1$. Тогда массив для хранения $K+1$ элементов можно описать так:

VAR T:ARRAY[1..K1] OF {тип элемента последовательности};

причем последний из вычисленных элементов последовательности будет храниться в элементе $T[K+1]$.

Решение обеих задач разбивается на два основных этапа: начальную установку и продвижение по последовательности. Начальную установку можно записать следующим образом:

$T[2]:=U(1); \dots; T[K+1]:=U(K);$

Продвижение по последовательности на один элемент вперед выполняется так:

FOR I:=1 TO K DO $T[I]:=T[I+1];$
 $T[K+1]:=F(T[K], \dots, T[1]);$

Каждый из элементов массива T , начиная с первого, как бы отбирает значение у следующего элемента. Последний элемент, после того как у него отобрали значение, получает новое путем применения рекуррентной формулы F . Безвозвратно теряемое старое значение $T[1]$ уже не нужно, поскольку на последующие вычисления оно повлиять не может.

Задачи 1 и 2 различаются только тем, до каких пор выполнять продвижение по последовательности. В задаче 1 продвижений должно быть $N-K$, поскольку после начальной установки $T[K+1]$ хранит K -й элемент последовательности, а нужно получить N -й. Для повторения продвижений удобно воспользоваться FOR-циклом:

```
FOR J:=K+1 TO N DO BEGIN
  FOR I:=1 TO K DO  $T[I]:=T[I+1];$ 
   $T[K+1]:=F(T[K], \dots, T[1]);$ 
  (*  $T[K+1]=U(J)$  *)
END;
(*  $T[K+1]=U(N)$  *)
```

(1)

В задаче 2 продвижение нужно закончить, как только станет истинным значение функции G . В этом случае для повторения продвижений целесообразно воспользоваться циклом REPEAT—UNTIL:

```
REPEAT
  FOR I:=1 TO K DO  $T[I]:=T[I+1];$ 
   $T[K+1]:=F(T[K], \dots, T[1]);$ 
UNTIL  $G(T[K+1], \dots, T[1]);$ 
```

(2)

В качестве примера решения задачи 1 рассмотрим функцию для вычисления N-го числа Фибоначчи:

```
FUNCTION ЧИСЛО-ФИБОНАЧЧИ(N: INTEGER): INTEGER;  
(* N — НОМЕР ИСКОМОГО ЭЛЕМЕНТА  
  ПРЕДПОЛАГАЕТСЯ, ЧТО N>0 *)  
VAR I, J: INTEGER;  
    T: ARRAY[1..3] OF INTEGER;  
BEGIN (* НАЧАЛЬНАЯ УСТАНОВКА *)  
    T[2]:=1; T[3]:=1;  
    (* ЦИКЛ ПРОДВИЖЕНИЙ *)  
    FOR J:=3 TO N DO BEGIN  
        FOR I:=1 TO 2 DO T[I]:=T[I+1];  
        T[3]:=T[2]+T[1];  
    END;  
    ЧИСЛО-ФИБОНАЧЧИ:=T[3];  
END;
```

Конечно, когда порядок рекуррентной последовательности невелик, можно обойтись без массива, наделив переменные, хранящие элементы последовательности, индивидуальными именами. В этом случае при продвижении по последовательности FOR-цикл заменится K присваиваниями. Рассмотрим в качестве примера решения задачи 2 функцию, вычисляющую корень квадратный из заданного числа A с заданной точностью. Пусть переменная ТЕК хранит текущее приближение к корню, а ПРЕД — предыдущее. Выражение

$ABS(U(N) - U(N-1)) < \text{ПОГРЕШНОСТЬ}$
будет выступать в качестве G.

```
FUNCTION КОРЕНЬ-КВАДРАТНЫЙ(A, ПОГРЕШНОСТЬ:  
REAL): REAL;  
(* A — ЗНАЧЕНИЕ, ИЗ КОТОРОГО ИЗВЛЕКАЕТСЯ КОРЕНЬ  
  ПОГРЕШНОСТЬ — ТРЕБУЕМАЯ ТОЧНОСТЬ  
  ПРИБЛИЖЕНИЙ *)  
VAR ТЕК, ПРЕД: REAL;  
(* ТЕК — ТЕКУЩЕЕ ПРИБЛИЖЕНИЕ К КОРНЮ  
  КВАДРАТНОМУ  
  ПРЕД — ПРЕДЫДУЩЕЕ ПРИБЛИЖЕНИЕ *)  
BEGIN (* НАЧАЛЬНАЯ УСТАНОВКА *)  
    ТЕК:=A+1;  
    (* ЦИКЛ ПРОДВИЖЕНИЙ *)  
    REPEAT  
        ПРЕД:=ТЕК;  
        ТЕК:=0.5*(ПРЕД+A/ПРЕД);
```



```
UNTIL ABS(ТЕК—ПРЕД)<ПОГРЕШНОСТЬ;
КОРЕНЬ-КВАДРАТНЫЙ:=ТЕК;
END;
```

В конце разд. 4.7 приведена уточненная версия функции для извлечения корня.

Определение рекуррентной последовательности можно несколько обобщить, если нумеровать элементы не обязательно от единицы и, что более важно, допустить зависимость функции F от N . При работе с такими последовательностями нужно включить в начальные установки присваивание начального номера и при каждом продвижении по последовательности увеличивать номер на единицу. Впрочем, от такого обобщения несколько усложнится только цикл (2). Цикл (1) не изменится, поскольку при вычислении элемента с данным номером все делает за нас FOR-цикл.

Поясним сказанное примерами. Попутно отметим, что к обработке рекуррентных последовательностей сводятся некоторые задачи, в формулировке которых последовательности явно не фигурируют. Пусть нужно вычислить значение многочлена степени M , заданного массивом коэффициентов C :

```
VAR C: ARRAY[0..M] OF REAL;
```

при данном значении переменной X . Известно, что для вычисления значения многочлена целесообразно воспользоваться схемой Горнера, записав многочлен в виде

$$((\dots (C[0] \cdot X + C[1]) \cdot X + \dots + C[M-1]) \cdot X + C[M]).$$

Теперь определим рекуррентную последовательность первого порядка $U(N)$ следующим образом:

$$U(0) = C[0], \\ U(N) = U(N-1) \cdot X + C[N], \quad 1 \leq N \leq M$$

(в данном случае элементы последовательности удобно нумеровать, начиная с нуля). Нетрудно показать, что $U(M)$ есть значение многочлена. Применим развитую выше методику для нахождения $U(M)$. Это — задача вычисления элемента последовательности по номеру. Если следовать общей схеме, нужно завести две переменные СПРЕД и СТЕК и выполнять продвижения следующим образом:

```
FOR J:=1 TO M DO BEGIN
  СПРЕД:=СТЕК;
  СТЕК:=СПРЕД*X+C[J];
END;
```

Ясно, однако, что переменная СПРЕД не нужна, поскольку при вычислении нового значения СТЕК можно просто воспользоваться

предыдущим значением этой переменной. Таким образом, в случае вычисления по номеру элемента рекуррентной последовательности первого порядка достаточно одной переменной — той, где будет накапливаться результат. Это же справедливо и для задачи вычисления элемента последовательности, удовлетворяющего заданному условию, если функция G зависит только от $U(N)$. Еще одна тонкость состоит в том, что, согласно разд. 2.8, целесообразно так организовывать начальные установки перед циклом, чтобы первый элемент последовательности вычислялся наравне с остальными. В итоге получаем следующую функцию:

```
CONST M=...;
TYPE НАБОР-КОЭФФИЦИЕНТОВ=ARRAY[0..M] OF REAL;
...
FUNCTION ГОРНЕР(VAR C: НАБОР-КОЭФФИЦИЕНТОВ;
X:REAL):REAL;
(* ФУНКЦИЯ ВЫЧИСЛЯЕТ ПО СХЕМЕ ГОРНЕРА
   ЗНАЧЕНИЕ МНОГОЧЛЕНА ОТ ОДНОЙ ПЕРЕМЕННОЙ,
   ЗАДАННОГО МАССИВОМ КОЭФФИЦИЕНТОВ C,
   ПРИ ДАННОМ ЗНАЧЕНИИ ПЕРЕМЕННОЙ X *)
VAR S: REAL; J: INTEGER;
(* В S БУДЕМ НАКАПЛИВАТЬ ЗНАЧЕНИЕ МНОГОЧЛЕНА
   *)
BEGIN S:=0;
  FOR J:=0 TO M DO S:=S*X+C[J];
  ГОРНЕР:=S;
END;
```

Техника работы с рекуррентными соотношениями полезна также при составлении программ обработки последовательностей, когда эффективнее не вычислять каждый член по общей формуле, а получать очередной элемент, зная значение предыдущего. Рассмотрим задачу суммирования ряда

$$1 + 1/1! + 1/2! + \dots + 1/N! + \dots$$

(это ряд для вычисления числа e). Пусть суммирование нужно производить до тех пор, пока очередной слагаемое не станет меньше наперед заданной величины ПОГРЕШНОСТЬ. Можно, конечно, для каждого N заново вычислять $N!$, но тогда время суммирования будет задаваться выражением $C_1 m^2$, где m — число просуммированных членов ряда. Вычисления можно ускорить, если воспользоваться соотношением

$$N! = N \cdot (N-1)!$$

Слагаемые ряда можно определить при помощи рекуррентной последовательности первого порядка

$$U(0)=1, U(N)=U(N-1)/N, N>0.$$

Найдем элемент последовательности, удовлетворяющий условию

$$U(N) < \text{ПОГРЕШНОСТЬ},$$

попутно суммируя уже вычисленные элементы, тогда время суммирования уменьшится до $C_2 m$:

```
FUNCTION ЧИСЛО-Е (ПОГРЕШНОСТЬ: REAL): REAL;
```

```
VAR N: INTEGER; U, S: REAL;
```

```
(* N — НОМЕР СЛАГАЕМОГО
```

```
U — ОЧЕРЕДНОЕ СЛАГАЕМОЕ
```

```
S — СУММА ПЕРВЫХ N ЭЛЕМЕНТОВ *)
```

```
BEGIN
```

```
  N:=0; U:=1; S:=U;
```

```
  REPEAT
```

```
    N:=N+1;
```

```
    U:=U/N; (* U=1/N! *)
```

```
    S:=S+U; (* S=1+1/1!+...+1/N! *)
```

```
  UNTIL U<ПОГРЕШНОСТЬ;
```

```
  ЧИСЛО-Е:=S;
```

```
END;
```

Сумма накапливается параллельно с вычислением элемента рекуррентной последовательности, удовлетворяющего заданному условию.

3.4. Упрощение циклов

Основную часть времени паскаль-машина тратит на выполнение циклов, в первую очередь внутреннего цикла (разд. 3.2). При заданном числе повторений время выполнения цикла зависит от числа действий, входящих в цикл. Чем проще циклы, т. е. чем меньше действий они содержат, тем быстрее выполнится вся программа. Поэтому, выявив внутренний цикл программы, целесообразно принять меры для его упрощения.

Проиллюстрируем упрощение циклов на примере задачи поиска информации в массиве. Требуется написать функцию с двумя параметрами — массивом целых чисел M и целой переменной K , — определяющую, содержит ли M элемент, равный K . (Параметр K обычно называют аргументом поиска или ключом.) Если такой элемент есть, результатом функции должен стать его номер в массиве (если искомым элементов несколько — номер любого из них). При отсутствии искомого элемента результатом функции должен быть нуль.

Для решения задачи будем последовательно просматривать

массив, пока не встретим искомый элемент или не выйдем за границу массива. Сначала применим приведенную в разд. 2.14 общую схему поиска наименьшего натурального числа, обладающего заданным свойством. Получаем функцию:

CONST N=...;

TYPE МАССИВ-ИНФОРМАЦИИ=ARRAY [1..N] OF INTEGER;

...

FUNCTION ПОСЛЕДОВАТЕЛЬНЫЙ-ПОИСК(

VAR M: МАССИВ-ИНФОРМАЦИИ; K: INTEGER): 0..N;

(* ФУНКЦИЯ ВЫДАЕТ НОМЕР ЭЛЕМЕНТА В МАССИВЕ M,
РАВНОГО АРГУМЕНТУ ПОИСКА K

ПРИ НЕУДАЧЕ ПОИСКА ВЫДАЕТСЯ 0 *)

VAR I:0..N; (* I — НОМЕР ПРОВЕРЯЕМОГО ЭЛЕМЕНТА *)

B: BOOLEAN; (* B ПОКАЗЫВАЕТ, РАВНЫ ЛИ M[I] И K *)

BEGIN

I:=0;

REPEAT

I:=I+1;

B:=(M[I]=K);

UNTIL (I=N) OR B;

IF B THEN

ПОСЛЕДОВАТЕЛЬНЫЙ-ПОИСК:=I

ELSE ПОСЛЕДОВАТЕЛЬНЫЙ-ПОИСК:=0;

END;

Нетрудно сообразить, что в данном случае стопт избавиться от переменной B, поскольку это упрощает цикл. После устранения B цикл поиска и последующая проверка запишутся так:

REPEAT

I:=I+1;

UNTIL (I=N) OR (M[I]=K);

IF M[I]=K THEN

ПОСЛЕДОВАТЕЛЬНЫЙ-ПОИСК:=I

ELSE ПОСЛЕДОВАТЕЛЬНЫЙ-ПОИСК:=0;

Кажется, что этот цикл уже не упростить, поскольку, с одной стороны, нужно продвигаться по массиву, а с другой — следить, не вышли ли мы за границу массива и не найден ли искомый элемент. Цикл как раз и содержит эти три действия. В то же время, если бы мы заранее знали, что искомый элемент в массиве есть, о выходе за границу можно было бы не беспокоиться. Руководствуясь этим соображением, подменим последний элемент массива ключом K, а после выхода из цикла разберемся, что же мы нашли — «подставной» или собственный элемент массива M:

```

FUNCTION УСКОРЕННЫЙ-ПОИСК(
  VAR M: МАССИВ-ИНФОРМАЦИИ; K: INTEGER): 0..N;
VAR I: 0..N;
  T: INTEGER; (* ВРЕМЕННОЕ ХРАНИЛИЩЕ ДЛЯ M[N] *)
BEGIN
  T:=M[N]; M[N]:=K;
  I:=0;
  REPEAT
    I:=I+1;
  UNTIL M[I]=K;
  M[N]:=T;
  IF M[I]=K THEN
    УСКОРЕННЫЙ-ПОИСК:=1
  ELSE УСКОРЕННЫЙ-ПОИСК:=0;
END;

```

Хотя вся функция несколько усложнилась, ее внутренний цикл стал заметно проще, что обещает существенное уменьшение времени ее работы.

Подчеркнем, что основное влияние на время работы программы оказывает выбор алгоритма. Сначала нужно проанализировать возможные алгоритмы решения задачи, выбрать из них самый быстрый, выявить в нем внутренний цикл и только затем приступить к упрощению выявленного внутреннего цикла.

3.5. Тип CHAR

Видимо, нет нужды доказывать, насколько важна такая область применения ЭВМ, как обработка текстов. Достаточно отметить, что без обработки текстов общение человека и ЭВМ было бы крайне затруднено. Базовым средством, позволяющим паскаль-машине работать с текстами, является тип данных CHAR. Значения этого типа образуют конечное, упорядоченное множество литер — букв, цифр, знаков и управляющих литер, таких как перевод строки.

Чтобы употребить в паскаль-программе константу типа CHAR, ее следует заключить в кавычки (такие же, как при печати текстов): 'A', '2', '+'. Переменные типа CHAR описываются обычным образом:

```

VAR ЛИТЕРА: CHAR;
  СТРОКА: ARRAY[1..128] OF CHAR;

```

Значения типа CHAR можно вводить, выводить, присваивать. В памяти паскаль-машины хранится, конечно, не начертание литеры, а ее порядковый номер в наборе литер, называемый кодом литеры. Обычно коды принадлежат диапазону от 0 до 255. Следующие

Для стандартные функции обеспечивают переход от литеры к ее коду и обратно:

ORD(J) — код литеры J,
CHR(K) — литеры с кодом K.

Пример.

VAR КОД-НУЛЯ: 0..255; ЛИТЕРА1: CHAR;
BEGIN КОД-НУЛЯ:=ORD('0');
ЛИТЕРА1:=CHR(КОД-НУЛЯ+1);

К значениям типа CHAR можно применять операции сравнения. Упорядочены литеры в соответствии с кодами: если J1 и J2 — литеры, то $J1 < J2$ при $ORD(J1) < ORD(J2)$.

Разные реализации паскаль-машины могут различаться наборами литер, однако все наборы удовлетворяют следующим условиям:

- 1) в набор входят латинские буквы A—Z и цифры 0—9;
- 2) латинские буквы упорядочены по алфавиту: 'A' < 'B' < ... < 'Y' < 'Z';
- 3) литера ЛИТ является цифрой тогда и только тогда, когда '0' < ЛИТ < '9', при этом '0' < '1' < ... < '8' < '9'.

Из условия 3) следует, что $ORD('9') = ORD('0') + 9$.

В качестве иллюстрации работы с литерами рассмотрим функцию, определяющую значение целого числа по его представлению в виде массива литер. Согласно трактовке записи числа в позиционной системе счисления (разд. 3.2), данная задача есть частный случай вычисления значения многочлена, заданного массивом коэффициентов. Применим ту же схему, что и в разд. 3.3:

```
CONST N=...,  
TYPE ТЕКСТ-ЧИСЛА=ARRAY[1..N] OF CHAR;  
...  
FUNCTION ПЕРЕВОД(VAR МЦ: ТЕКСТ-ЧИСЛА): INTEGER;  
(* ФУНКЦИЯ ОПРЕДЕЛЯЕТ ЗНАЧЕНИЕ ЦЕЛОГО ЧИСЛА  
ПО ЕГО ТЕКСТОВОМУ ПРЕДСТАВЛЕНИЮ  
В ВИДЕ МАССИВА ЦИФР, МЦ *)  
VAR I, S: INTEGER;  
(* I — НОМЕР ОБРАБАТЫВАЕМОЙ ЦИФРЫ *)  
(* В S БУДЕМ НАКАПЛИВАТЬ РЕЗУЛЬТАТ *)  
BEGIN  
S:=0;  
FOR I:=1 TO N DO  
S:=S*10+(ORD(МЦ[I])-ORD('0'));  
ПЕРЕВОД:=S;  
END;
```


3.6. Понижение размерности массивов

До сих пор, анализируя программы, мы уделяли основное внимание времени их работы. Есть, однако, и другая немаловажная характеристика — объем памяти, необходимой для работы программы. Память иногда оказывается не менее дефицитным ресурсом, чем время; соответственно при составлении программ следует обращать внимание на ее экономию. Самый радикальный способ экономии памяти — понижение размерности (в идеале — полное устранение) используемых в программе массивов. Чтобы добиться понижения размерности, следует, с одной стороны, стараться использовать информацию сразу после ее получения, не запасать ее впрок, а с другой — не хранить информацию, ставшую ненужной.

Рассмотрим следующую задачу. Дана скаляр-функция от двух целочисленных параметров $F(I, J)$, I и J изменяются от 1 до 101 (конкретный вид функции F для нас сейчас не важен). Напечатать картинку 101×101 , изображающую множество пар (I, J) таких, что $F(I, J) < 0$. В позиции (I, J) должна стоять звездочка, если $F(I, J) < 0$. Первый вариант решения может быть следующим. Заведем массив литер размером 101×101 , где будем формировать картинку. После окончания формирования останется только напечатать массив:

```
PROGRAM C-МАССИВОМ(OUTPUT);
CONST M=101; N=101;
VAR КАРТИНКА: ARRAY[1..M,1..N] OF CHAR;
    I:1..M; J:1..N;
(* (I,J) — КООРДИНАТЫ ОБРАБАТЫВАЕМОЙ ТОЧКИ *)
FUNCTION F(I, J:INTEGER):REAL;
...
END;
BEGIN (* ГЛАВНАЯ ПРОГРАММА *)
  (* ФОРМИРОВАНИЕ КАРТИНКИ *)
  FOR I:=1 TO M DO
    FOR J:=1 TO N DO
      IF F(I,J)<0 THEN КАРТИНКА[I,J]:='*'
      ELSE КАРТИНКА[I,J]:=' ';
    (* ВЫВОД КАРТИНКИ *)
    FOR I:=1 TO M DO BEGIN
      FOR J:=1 TO N DO WRITE(КАРТИНКА[I,J]);
      WRITELN;
    END;
  END.
```

Если, однако, вспомнить, что при использовании процедур WRITE и WRITELN имитируется работа пишущей машинки (разд. 2.4),

то станет ясно, что картинку можно выводить на печать сразу вслед за анализом значения $F(I, J)$, не накапливая ее целиком. Получаем следующую программу без массива:

```
PROGRAM ОБЛАСТЬ-ОТРИЦАТЕЛЬНОСТИ(OUTPUT);
CONST M=101; N=101;
VAR I:1..M; J:1..N;
(* (I,J) — КООРДИНАТЫ ОБРАБАТЫВАЕМОЙ ТОЧКИ *)
FUNCTION F(I,J:INTEGER):REAL;
...
END;
BEGIN (* ГЛАВНАЯ ПРОГРАММА *)
  FOR I:=1 TO M DO BEGIN
    FOR J:=1 TO N DO
      IF F(I,J)<0 THEN WRITE('#')
      ELSE WRITE(' ');
    WRITELN;
  END;
END.
```

В данном случае удалось избавиться от массива из $101 \cdot 101 = 10201$ элемента.

Рассмотрим еще один пример. Требуется напечатать первые строки треугольника Паскаля с нулевой по N-ю. Если через $T(I, J)$ обозначить число, стоящее в I-й строке на J-м месте, то по определению треугольника Паскаля справедливы соотношения:

$$T(I, 0) = T(I, I) = 1;$$

$$T(I, J) = T(I-1, J-1) + T(I-1, J), \quad I \geq 0, \quad 0 < J < I,$$

(строки и элементы в строках нумеруются, начиная с 0). Несколько первых строк треугольника Паскаля выглядят так:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Если буквально воспользоваться определением треугольника Паскаля, получается следующая процедура:

```
PROCEDURE ЦЕЛИКОМ-ТРЕУГОЛЬНИК(N: INTEGER);
CONST ПРЕДЕЛ=...;
(* ПРОЦЕДУРА ПЕЧАТАЕТ СТРОКИ ТРЕУГОЛЬНИКА
  ПАСКАЛЯ, С 0 ПО N.
  ПРЕДПОЛАГАЕТСЯ, ЧТО 0 <= N <= ПРЕДЕЛ *)
VAR T: ARRAY[0..ПРЕДЕЛ, 0..ПРЕДЕЛ] OF INTEGER;
    I, J: 0..ПРЕДЕЛ;
```

```

(* T — МАССИВ ДЛЯ ФОРМИРОВАНИЯ ТРЕУГОЛЬНИКА,
   I — НОМЕР ВЫЧИСЛЯЕМОЙ СТРОКИ,
   J — НОМЕР ЭЛЕМЕНТА В СТРОКЕ *)
BEGIN
  (* ВЫЧИСЛЕНИЕ ЭЛЕМЕНТОВ ТРЕУГОЛЬНИКА
     ПАСКАЛЯ *)
  FOR I:=0 TO N DO BEGIN
    T[I,0]:=1; T[I,I]:=1;
    FOR J:=1 TO I-1 DO T[I,J]:=T[I-1,J-1]+T[I-1,J];
  END;
  (* ПЕЧАТЬ ВЫЧИСЛЕННЫХ СТРОК *)
  FOR I:=0 TO N DO BEGIN
    FOR J:=0 TO I DO WRITE (T[I,J]);
    WRITELN;
  END;
END;

```

Подумаем, как уменьшить объем необходимой памяти. При вычислении очередной строки используются не все ранее найденные элементы треугольника, а только предыдущая строка. Печатать строки треугольника можно не в конце программы, а по мере их вычисления. Значит, достаточно двух одномерных массивов, в которых по аналогии с рекуррентной последовательностью первого порядка будут вычисляться строки треугольника. Чтобы избежать лишних копирований массивов, наведем логическую переменную, которая покажет, в каком из двух массивов нужно вычислять очередную строку. После вычисления строки к переменной следует применить операцию логического отрицания.

```

PROCEDURE ДВЕ-СТРОКИ(N: INTEGER);
CONST ПРЕДЕЛ=...;
VAR T1, T2: ARRAY[0..ПРЕДЕЛ] OF INTEGER;
    BT1: BOOLEAN;
    I, J: 0..ПРЕДЕЛ;
(* T1 и T2 ХРАНЯТ ДВЕ ПОСЛЕДНИЕ
   ИЗ ВЫЧИСЛЕННЫХ СТРОК ТРЕУГОЛЬНИКА
   ПАСКАЛЯ
   BT1 ИСТИННО, ЕСЛИ НОВУЮ СТРОКУ
   СЛЕДУЕТ ВЫЧИСЛЯТЬ В T1
   I — НОМЕР ВЫЧИСЛЯЕМОЙ СТРОКИ,
   J — НОМЕР ЭЛЕМЕНТА В СТРОКЕ *)
BEGIN
  T2[0]:=1; (* ЭТОТ ЭЛЕМЕНТ МЕНЯТЬСЯ НЕ БУДЕТ *)
  BT1:=TRUE; (* НУЛЕВУЮ СТРОКУ СФОРМИРУЕМ В
               T1 *)
  FOR I:=0 TO N DO BEGIN

```



```

IF BT1 THEN BEGIN
T1[I]:=1;
FOR J:=1 TO I-1 DO T1[J]:=T2[J-1]+T2[J];
FOR J:=0 TO I DO WRITE(T1[J]);
END
ELSE BEGIN
T2[I]:=1;
FOR J:=1 TO I-1 DO T2[J]:=T1[J-1]+T1[J];
FOR J:=0 TO I DO WRITE (T2[J]);
END;
WRITELN; BT1:=NOT (BT1);
END;
END;

```

Процедура усложнилась, но и экономия памяти по сравнению с предыдущим вариантом существенная: в массивах хранится 2·ПРЕДЕЛ чисел вместо ПРЕДЕЛ·ПРЕДЕЛ. Можно, однако, уменьшить расход памяти еще в два раза, если обойтись одним одномерным массивом и вычислять очередную строку треугольника Паскаля на месте предыдущей. Элементы строки целесообразно вычислять справа налево, чтобы изменять значения только тех компонент массива, которые в дальнейшем не понадобятся:

```

PROCEDURE СТРОКА-ТРЕУГОЛЬНИКА-ПАСКАЛЯ (N:
INTEGER);
CONST ПРЕДЕЛ=...;
VAR TT: ARRAY [0..ПРЕДЕЛ] OF INTEGER;
    I, J: 0..ПРЕДЕЛ;
(* TT — МАССИВ ДЛЯ ХРАНЕНИЯ ПОСЛЕДНЕЙ
ИЗ ВЫЧИСЛЕННЫХ СТРОК ТРЕУГОЛЬНИКА
ПАСКАЛЯ
I — НОМЕР ВЫЧИСЛЯЕМОЙ СТРОКИ;
J — НОМЕР ЭЛЕМЕНТА В СТРОКЕ *)
BEGIN
FOR I:=0 TO N DO BEGIN
    (* ЗАНЕСЕНИЕ ПРАВОЙ ЕДИНИЦЫ В СТРОКУ *)
    TT[I]:=1;
    (* ПРИ I=0 ЭТА ИНСТРУКЦИЯ ПОМЕЩАЕТ 1
    В НУЛЕВУЮ КОМПОНЕНТУ МАССИВА TT, КОТОРАЯ
    ПОТОМ НЕ МЕНЯЕТСЯ *)
    FOR J:=I-1 DOWNT0 1 DO TT [J]:=TT[J-1]+TT[J];
    FOR J:=0 TO I DO WRITE (TT[J]);
    WRITELN;
END;
END;

```

По сравнению с первоначальным вариантом мы не только сократили расход памяти в ПРЕДЕЛ раз, но и получили более изящную процедуру.

Наконец, можно обойтись совсем без массивов, но для этого нужно догадаться, что (I, J) -й элемент треугольника Паскаля есть число сочетаний из I по J и что справедливо соотношение

$$T(I, J) = T(I, J-1) \cdot (I-J+1)/J, \quad 0 < J \leq I.$$

Печатать элементы треугольника будем сразу после их вычисления.

PROCEDURE ОДИН-ЭЛЕМЕНТ (N: INTEGER);

(* ПРОЦЕДУРА ПЕЧАТАЕТ СТРОКИ
ТРЕУГОЛЬНИКА ПАСКАЛЯ, С 0 ПО N.

ПРЕДПОЛАГАЕТСЯ, ЧТО $N \geq 0$ *)

VAR I, J, C: INTEGER;

(* (I, J) — КООРДИНАТЫ ВЫЧИСЛЯЕМОГО ЭЛЕМЕНТА,
C — ЕГО ЗНАЧЕНИЕ *)

BEGIN

FOR I:=0 TO N DO BEGIN

C:=1; WRITE (C);

FOR J:=1 TO I DO BEGIN

C:=C*(I-J+1) DIV J; WRITE (C);

END;

WRITELN;

END;

END;

Конечно, наибольший эффект получается тогда, когда удастся по-новому посмотреть на поставленную задачу. Но и следование несложному правилу — потреблять информацию сразу после ее получения — способно существенно снизить расход памяти.

3.7. Несколько примеров программ

1. Первая из разбираемых задач известна как задача о лондонском метеорологе. Ее формулировка такова. Вводится последовательность целых чисел, обозначающих количество осадков в Лондоне за определенные дни. Числа эти должны быть неотрицательными. Признак конца последовательности — число 99 999. Требуется подсчитать число дождливых дней (когда количество осадков положительно) и среднее количество осадков.

Это — типичная задача обработки последовательности. Следует циклически выполнять три действия: чтение очередного числа, сравнение его с 99 999 и обработку этого числа. Разумеется, и речи быть не может о том, чтобы предварительно запомнить все введенные числа в массиве. Применим методику, развитую в разд. 2.8.

Предполагается, что программа общается с человеком, сидящим за терминалом:

```
PROGRAM МЕТЕО (INPUT, OUTPUT);
CONST ПРИЗНАК-КОНЦА=99999;
VAR ОЧЕРЕДНОЕ-ЧИСЛО,
      ЧИСЛО-ДНЕЙ,
      ЧИСЛО-ДОЖДЛИВЫХ-ДНЕЙ,
      ВСЕГО-ОСАДКОВ: INTEGER;
BEGIN
  (* НАЧАЛЬНЫЕ УСТАНОВКИ *)
  ЧИСЛО-ДНЕЙ:=0;
  ЧИСЛО-ДОЖДЛИВЫХ-ДНЕЙ:=0;
  ВСЕГО-ОСАДКОВ:=0;
  WRITELN(' ПРОГРАММА МЕТЕО ГОТОВА К РАБОТЕ');
  WRITELN(' ВВОДИТЕ ЦЕЛЫЕ НЕОТРИЦАТЕЛЬНЫЕ');
  WRITELN(' ЧИСЛА, ОБОЗНАЧАЮЩИЕ КОЛИЧЕСТВО ');
  WRITELN(' ОСАДКОВ ЗА ДЕНЬ. ДЛЯ ПРЕКРАЩЕНИЯ ');
  WRITELN(' ОБРАБОТКИ ВВЕДИТЕ ЧИСЛО ');
  WRITELN(ПРИЗНАК-КОНЦА);
  READ(ОЧЕРЕДНОЕ-ЧИСЛО);
  WHILE ОЧЕРЕДНОЕ-ЧИСЛО<> ПРИЗНАК-КОНЦА DO
  BEGIN
    IF ОЧЕРЕДНОЕ-ЧИСЛО>=0 THEN BEGIN
      ЧИСЛО-ДНЕЙ:=ЧИСЛО-ДНЕЙ+1;
      IF ОЧЕРЕДНОЕ-ЧИСЛО>0 THEN BEGIN
        ВСЕГО-ОСАДКОВ:=
          ВСЕГО-ОСАДКОВ+ОЧЕРЕДНОЕ-ЧИСЛО;
        ЧИСЛО-ДОЖДЛИВЫХ-ДНЕЙ:=
          ЧИСЛО-ДОЖДЛИВЫХ-ДНЕЙ+1;
      END
    END
    ELSE BEGIN (* ОЧЕРЕДНОЕ-ЧИСЛО<0 *)
      WRITELN(' *****');
      WRITELN(' ВВЕДЕНО ОТРИЦАТЕЛЬНОЕ ЧИСЛО ');
      WRITELN(' ***** ', ОЧЕРЕДНОЕ-ЧИСЛО,
        '*****');
      WRITELN(' ПОВТОРИТЕ ВВОД, ');
      WRITELN(' НАЧИНАЯ С МЕСТА ОШИБКИ ');
      READLN;
      (* ПРОПУСТИМ ЧИСЛА, ОСТАВШИЕСЯ В
        СТРОКЕ *)
    END;
    READ(ОЧЕРЕДНОЕ-ЧИСЛО);
  END; (* КОНЕЦ ЦИКЛА *)
  WRITELN(' ИЗ ', ЧИСЛО-ДНЕЙ, ' ДНЕЙ');
```



```

WRITELN(' ДОЖДЛИВЫХ БЫЛО ',
ЧИСЛО-ДОЖДЛИВЫХ-ДНЕЙ);
IF ЧИСЛО-ДНЕЙ>0 THEN
  WRITELN(' СРЕДНЕЕ КОЛИЧЕСТВО ОСАДКОВ РАВНО
    ВСЕГО-ОСАДКОВ DIV ЧИСЛО-ДНЕЙ);
END.

```

В подобных задачах самое существенное — разумно организовать диалог с человеком. Собственно обработка вводимых данных тривиальна.

2. Напишем процедуру перевода числа из I-ричной системы счисления в J-ричную, $2 \leq I, J \leq 10$. Параметрами процедуры должны быть I, J, массив, в элементах которого хранятся I-ричные цифры числа, и переменная, указывающая номер элемента массива, в котором помещается старшая цифра числа. Младшая цифра хранится в элементе массива с наибольшим номером. Требуется поместить в тот же массив J-ричную запись числа, запомнив расположение старшей цифры. Предполагается, что само число не слишком велико и может быть представлено значением типа INTEGER:

```

CONST МАКС-РАЗМЕР=40;
TYPE МАССИВ-ЦИФР=ARRAY [1..МАКС-РАЗМЕР] OF CHAR;

```

...

```

PROCEDURE ПЕРЕВОД
(I,J: INTEGER; VAR M: МАССИВ-ЦИФР;
  VAR НАЧАЛО: 1..МАКС-РАЗМЕР);
  VAR K: 0..МАКС-РАЗМЕР; ЧИСЛО: INTEGER;
(* K — НОМЕР ОБРАБАТЫВАЕМОГО ЭЛЕМЕНТА
  МАССИВА
  ЧИСЛО — ВЕЛИЧИНА ПРЕОБРАЗУЕМОГО ЧИСЛА *)
BEGIN
  (* ВЫЧИСЛИМ ЧИСЛО ПО I-РИЧНОЙ ЗАПИСИ *)
  ЧИСЛО:=0;
  FOR K:=НАЧАЛО TO МАКС-РАЗМЕР DO
    ЧИСЛО:=ЧИСЛО*I+(ORD(M[K])—ORD('0'));
  (* СФОРМИРУЕМ J-РИЧНУЮ ЗАПИСЬ ЧИСЛА *)
  K:=МАКС-РАЗМЕР;
  REPEAT
    M[K]:=CHR(ORD('0')+(ЧИСЛО MOD J));
    K:=K—1; ЧИСЛО:=ЧИСЛО DIV J;
  UNTIL ЧИСЛО=0;
  НАЧАЛО:=K+1;
END;

```

3. Дан массив P, содержащий N натуральных чисел. Требуется найти наименьшее натуральное число, не представимое в виде

суммы элементов массива P. Каждый элемент может входить в сумму не более одного раза.

Кажется, что в этой задаче не обойтись без перебора возможных сумм. Оказывается, однако, что ситуацию можно существенно упростить, если предварительно упорядочить массив. Будем считать, что $P[1] \leq P[2] \leq \dots \leq P[N]$. Обозначим через $S(K)$ сумму K наименьших элементов массива. Пусть далее элементы $P[1], \dots, P[K]$ таковы, что в виде их сумм можно представить любое число от 1 до $S(K)$. Утверждается, что любое число от 1 до $S(K+1)$ можно представить в виде сумм элементов $P[1], \dots, P[K+1]$ тогда и только тогда, когда $P[K+1] \leq S(K) + 1$. Действительно, пусть $P[K+1] \leq S(K) + 1$. Тогда все числа, не превосходящие $S(K)$, можно составить с помощью первых K элементов массива, а большие числа M ($S(K) + 1 \leq M \leq S(K+1) = S(K) + P[K+1]$) разбиваем на два слагаемых: $P[K+1]$ и $(M - P[K+1])$. Второе слагаемое неотрицательно и не превосходит $S(K)$, поэтому его можно составить из сумм первых K элементов. С другой стороны, если

$$P[K+1] > S(K) + 1,$$

то число $S(K) + 1$ составить, очевидно, невозможно. Тем самым мы обосновали правильность следующей несложной программы:

```
CONST N=...;
TYPE НАБОР-СЛАГАЕМЫХ=ARRAY[1..N] OF INTEGER;
...
FUNCTION НЕПРЕДСТАВИМОЕ
(VAR P:НАБОР-СЛАГАЕМЫХ): INTEGER;
VAR I,J,НОМ-МИН,МИН,S: INTEGER;
BEGIN
  (* УПОРЯДОЧИМ МАССИВ ПО ВОЗРАСТАНИЮ *)
  FOR I:=1 TO N-1 DO BEGIN
    (* ВЫБЕРЕМ МИНИМАЛЬНЫЙ СРЕДИ ЭЛЕМЕНТОВ
      P[I],...,P[N]
      И ПОМЕНЯЕМ ЕГО МЕСТАМИ С P[I] *)
    МИН:=P[I]; НОМ-МИН:=I;
    FOR J:=I+1 TO N DO
      IF МИН>P[J] THEN BEGIN
        МИН:=P[J]; НОМ-МИН:=J;
      END;
    P[НОМ-МИН]:=P[I]; P[I]:=МИН;
  END;
  (* МАССИВ УПОРЯДОЧЕН. ТЕПЕРЬ ИЩЕМ
    МИНИМАЛЬНОЕ ЧИСЛО,
    НЕ ПРЕДСТАВИМОЕ В ВИДЕ
    СУММ ЭЛЕМЕНТОВ МАССИВА P *)
  I:=1; S:=1;
```

```

WHILE (I <= N) AND (S >= P[I]) DO BEGIN
  S := S + P[I];
  (* S = 1 + P[1] + ... + P[I] И ВСЕ ЧИСЛА,
    МЕНЬШИЕ S, МОЖНО СОСТАВИТЬ
    ИЗ ЭЛЕМЕНТОВ P[1], ..., P[I] *)
  I := I + 1;
END;
НЕПРЕДСТАВИМОЕ := S;
END;

```

4. Рассмотрим задачу, в которой перебора избежать не удастся. Нужно напечатать все существенно различные способы разбиения данного натурального числа N на слагаемые. Способы, различающиеся лишь порядком слагаемых, существенно различными не считаются.

Главное в подобных задачах — выбрать способ систематического порождения разбиений, чтобы все разбиения встречались ровно один раз. Введем сначала понятие лексикографической упорядоченности разбиений. Пусть в различных разбиениях P_1 и P_2 первые k слагаемых одинаковы, а $(k+1)$ -е слагаемое в разбиении P_1 меньше, чем в P_2 , или вообще отсутствует. В таком случае будем говорить, что разбиение P_1 лексикографически меньше, чем P_2 . Например, разбиение числа 13 на слагаемые

$$4 + 4 + 3 + 2$$

лексикографически меньше разбиения

$$4 + 4 + 4 + 1.$$

Будем записывать разбиения в порядке убывания слагаемых и генерировать их в лексикографическом порядке. Первое разбиение — это разбиение в сумму единиц. Последнее разбиение представляется самим числом N . Чтобы от данного разбиения перейти к следующему, сохраняя упорядоченность слагаемых, нужно найти самое правое слагаемое, меньшее своего соседа слева, и (если это возможно) увеличить его на единицу, дополнив сумму единичными слагаемыми (если сумма предыдущих слагаемых меньше N).

Наборы слагаемых будем хранить в массиве. Ясно, что в наборах будет много одинаковых слагаемых, поэтому полезно уплотнить информацию: вместо хранения каждого слагаемого в отдельном элементе массива будем хранить пары (величина слагаемого, количество таких слагаемых). С одной стороны, такое уплотнение ведет к экономии памяти, а с другой — оно упрощает программу, поскольку для нахождения слагаемого, которое будет увеличиваться на единицу, не понадобится цикл:


```

PROGRAM РАЗБИЕНИЕ-НА-СЛАГАЕМЫЕ (INPUT, OUTPUT);
CONST ПРЕДЕЛ=100;
VAR ВЕЛ: ARRAY[0..ПРЕДЕЛ] OF INTEGER;
    КРАТ: ARRAY[1..ПРЕДЕЛ] OF INTEGER;
(* МАССИВОВ ТАКОГО РАЗМЕРА ДОСТАТОЧНО,
   ЧТОБЫ РАЗЛОЖИТЬ НА СЛАГАЕМЫЕ ЛЮБОЕ ЧИСЛО
   ВПЛОТЬ ДО 5150 *)
PC, ОСТАТОК, N: INTEGER;
ВСЕ СЛАГАЕМЫЕ РАЗБИВАЮТСЯ НА ГРУППЫ
РАВНЫХ ЧИСЕЛ
ВЕЛ[I], I>0 — ВЕЛИЧИНА СЛАГАЕМОГО ИЗ
I-Й ГРУППЫ, *)
ВЕЛ[0] — ВСПОМОГАТЕЛЬНЫЙ ЭЛЕМЕНТ, РАВНЫЙ 0
КРАТ[I] — КОЛИЧЕСТВО СЛАГАЕМЫХ В I-Й ГРУППЕ
PC — ЧИСЛО РАЗНЫХ СЛАГАЕМЫХ, Т. Е. ЧИСЛО
ГРУПП
ОСТАТОК — КОЛИЧЕСТВО ЕДИНИЦ, КОТОРЫМИ
НУЖНО ДОПОЛНИТЬ СУММУ ПРИ ПЕРЕХОДЕ
К СЛЕДУЮЩЕМУ РАЗБИЕНИЮ, N — РАЗЛАГАЕМОЕ
ЧИСЛО *)
PROCEDURE ПЕЧАТЬ-РАЗБИЕНИЯ;
VAR I: 1..ПРЕДЕЛ; J: INTEGER;
BEGIN
    FOR I:=1 TO PC DO
        FOR J:=1 TO КРАТ[I] DO WRITE(' ', ВЕЛ[I]:1);
        WRITELN;
    END; (* КОНЕЦ ОПИСАНИЯ ПРОЦЕДУРЫ
        ПЕЧАТЬ-РАЗБИЕНИЯ *)
BEGIN READ(N);
    WRITELN(' РАЗБИЕНИЕ ЧИСЛА ', N,
            ' НА СЛАГАЕМЫЕ');
    (* СФОРМИРУЕМ И НАПЕЧАТАЕМ РАЗБИЕНИЕ
       НА ЕДИНИЦЫ *)
    ВЕЛ[0]:=0; ВЕЛ[1]:=1; КРАТ[1]:=N; PC:=1;
    ПЕЧАТЬ-РАЗБИЕНИЯ;
    (* БУДЕМ ПОРОЖДАТЬ РАЗБИЕНИЯ
       В ЛЕКСИКОГРАФИЧЕСКОМ ПОРЯДКЕ. ПОСЛЕДНЕЕ
       РАЗБИЕНИЕ СОСТОИТ ИЗ ОДНОГО СЛАГАЕМОГО,
       РАВНОГО N *)
    WHILE ВЕЛ[1]<N DO BEGIN
        ОСТАТОК:=0;
        (* ЕСЛИ КРАТНОСТЬ МИНИМАЛЬНОГО СЛАГАЕМОГО
           РАВНА 1, ЕГО УВЕЛИЧИВАТЬ НЕЛЬЗЯ, ИНАЧЕ
           СУММА ПРЕВЫСИТ N; УВЕЛИЧИМ ТОГДА
           СЛАГАЕМОЕ ИЗ ПРЕДПОСЛЕДНЕЙ ГРУППЫ *)

```

```

IF КРАТ[РС]=1 THEN BEGIN
    ОСТАТОК:=ВЕЛ[РС];
    РС:=РС-1;
END;
ОСТАТОК:=ОСТАТОК÷ВЕЛ[РС]×КРАТ[РС]-
(ВЕЛ[РС]+1);
(* ПРОВЕРИМ, НЕ РАВНО ЛИ УВЕЛИЧЕННОЕ
СЛАГАЕМОЕ
ЭЛЕМЕНТАМ СОСЕДНЕЙ ГРУППЫ. В ЭТОМ
СЛУЧАЕ НУЖНО УВЕЛИЧИТЬ НА 1 КРАТНОСТЬ
СОСЕДНЕЙ ГРУППЫ И ОТБРОСИТЬ ПОСЛЕДНЮЮ
ГРУППУ.
ПРИ РС=1 ВЫПОЛНИТСЯ ELSE-ЧАСТЬ,
ТАК КАК ВЕЛ[0]=0 *)
IF ВЕЛ[РС-1]=ВЕЛ[РС]÷1 THEN BEGIN
    РС:=РС-1;
    КРАТ[РС]:=КРАТ[РС]÷1;
END
ELSE BEGIN
    ВЕЛ[РС]:=ВЕЛ[РС]+1;
    КРАТ[РС]:=1;
END;
(* ЕСЛИ НУЖНО, ДОПОЛНИМ СУММУ ЕДИНИЦАМИ *)
IF ОСТАТОК>0 THEN BEGIN
    РС:=РС+1;
    ВЕЛ[РС]:=1;
    КРАТ[РС]:=ОСТАТОК;
END;
(* ПЕРЕХОД К СЛЕДУЮЩЕМУ РАЗБИЕНИЮ
ЗАКОНЧЕН *)
ПЕЧАТЬ-РАЗБИЕНИЯ;
END;
END.

```

Следующие две задачи иллюстрируют методы работы с упорядоченными массивами.

5. Даны два массива целых чисел А и В, упорядоченные по возрастанию. Требуется поместить все элементы этих массивов в массив С, который также должен быть упорядочен по возрастанию.

Основная идея решения состоит в том, чтобы, сравнивая очередные элементы А[І] и В[Ј], выяснить, какой из них меньше, перенести его в массив С и продвигнуться по тому массиву, из которого мы взяли элемент. Когда один из массивов будет пройден до конца, останется только перенести в С «хвост» другого массива;

```

CONST M=...; N=...; P=...;
TYPE ТИП-А=ARRAY[1..M] OF INTEGER;
      ТИП-В=ARRAY[1..N] OF INTEGER;
      ТИП-С=ARRAY[1..P] OF INTEGER;
(* P=M+N *)

...
PROCEDURE СЛИЯНИЕ(VAR А:ТИП-А;
  VAR В:ТИП-В;VAR С:ТИП-С);
VAR I,J,K: INTEGER;
(* I — НОМЕР ОБРАБАТЫВАЕМОГО ЭЛЕМЕНТА
  МАССИВА А
  J — НОМЕР ОБРАБАТЫВАЕМОГО ЭЛЕМЕНТА
  МАССИВА В
  K — НОМЕР ЗАПОЛНЯЕМОГО ЭЛЕМЕНТА
  МАССИВА С *)
BEGIN
  I:=1; J:=1; K:=1;
  WHILE (I<=M) AND (J<=N) DO
    IF A[I]<B[J] THEN BEGIN
      C[K]:=A[I]; I:=I+1; K:=K+1;
    END
    ELSE BEGIN
      C[K]:=B[J]; J:=J+1; K:=K+1;
    END;
  (* ЦИКЛ ОКОНЧЕН
    ОДИН ИЗ МАССИВОВ ОБРАБОТАН ПОЛНОСТЬЮ
    ПЕРЕПИШЕМ ОСТАТОК ДРУГОГО МАССИВА *)
  WHILE I<=M DO BEGIN
    C[K]:=A[I]; I:=I+1; K:=K+1;
  END;
  WHILE J<=N DO BEGIN
    C[K]:=B[J]; J:=J+1; K:=K+1;
  END;
END;
END;

```

6. Рассмотрим еще один пример работы с упорядоченными массивами. Дан двумерный массив целых чисел, содержащий M строк и N столбцов. Строки массива упорядочены по возрастанию. Требуется найти и напечатать элемент массива, входящий во все строки. Если такого элемента нет, печатается слово НЕТ.

Пусть у нас есть кандидат, которого мы пытаемся найти во всех строках. Будем двигаться по каждой строке, пока либо не найдем элемент, равный искомому (после чего перейдем к рассмотрению следующей строки), либо не найдем сразу больший элемент (это значит, что рассматриваемый кандидат входит не во все строки

и надо искать другой элемент), либо не выйдем за пределы строки (после чего поиск продолжать бессмысленно). Возобновлять поиск в данной строке нужно с того места, где мы остановились в предыдущий раз, поскольку все ранее рассмотренные элементы заведомо меньше искомого. В предыдущей задаче для запоминания позиции в трех массивах нужны были три переменные. В данной задаче для той же цели нужен массив из M элементов:

```
CONST M=...; N=...;
TYPE МАТРИЦА=ARRAY [1..M,1..N] OF INTEGER;
...
PROCEDURE ВО-ВСЕХ-СТРОКАХ (VAR P:МАТРИЦА);
VAR I: 0..M; J: 0..N;
    НАШЛИ,НЕГДЕ-ИСКАТЬ: BOOLEAN;
    ПРОДОЛЖ: ARRAY [1..M] OF 1..N;
    КАНДИДАТ: INTEGER;
(* (I,J) — координаты проверяемого элемента
НАШЛИ—ИСТИННО, ПОКА ГИПОТЕЗА О ТОМ, ЧТО
ЭЛЕМЕНТ ВХОДИТ ВО ВСЕ СТРОКИ,
НЕ ОТВЕРГНУТА
НЕГДЕ-ИСКАТЬ—ИСТИННО, ЕСЛИ НАЙДЕНА
СТРОКА,
ВСЕ ЭЛЕМЕНТЫ КОТОРОЙ МЕНЬШЕ ИСКОМОГО
ПРОДОЛЖ—МАССИВ ДЛЯ ХРАНЕНИЯ МЕСТ
ПРОДОЛЖЕНИЯ ПОИСКА ДЛЯ КАЖДОЙ СТРОКИ
КАНДИДАТ—ИСКОМЫЙ ЭЛЕМЕНТ *)
BEGIN
    FOR I:=1 TO M DO ПРОДОЛЖ[I]:=1;
    КАНДИДАТ:=P[1,1]; НЕГДЕ-ИСКАТЬ:=FALSE;
    REPEAT
        (* ЦИКЛ ПЕРЕБОРА ВОЗМОЖНЫХ КАНДИДАТОВ *)
        I:=0; НАШЛИ:=TRUE;
        REPEAT
            (* ЦИКЛ ПРОВЕРКИ ОДНОГО КАНДИДАТА *)
            I:=I+1, J:=ПРОДОЛЖ[I]-1;
            REPEAT
                (* ЦИКЛ ПРОСМОТРА ОДНОЙ СТРОКИ *)
                J:=J+1;
            UNTIL (J=N) OR (P[I,J]>=КАНДИДАТ);
            IF P[I,J]<>КАНДИДАТ THEN BEGIN
                НАШЛИ:=FALSE;
                IF P[I,J]>КАНДИДАТ THEN BEGIN
                    КАНДИДАТ:=P[I,J]; ПРОДОЛЖ[I]:=J;
                END
            ELSE (* ВСЕ ЭЛЕМЕНТЫ СТРОКИ МЕНЬШЕ
```

```

      ИСКОМОГО *)
      НЕГДЕ-ИСКАТЬ:=TRUE;
END
ELSE (*P[I,J]=КАНДИДАТ *)
  IF J=N THEN НЕГДЕ-ИСКАТЬ:=TRUE
  ELSE ПРОДОЛЖ[I]:=J+1;
  (* ЕСЛИ КАНДИДАТ В ДАЛЬНЕЙШЕМ БУДЕТ
    ОТВЕРГНУТ,
    ПОИСК В СТРОКЕ НУЖНО ВОЗОБНОВЛЯТЬ
    СО СЛЕДУЮЩЕГО ЭЛЕМЕНТА (ЕСЛИ ОН ЕСТЬ)
    ИЛИ ПРЕКРАЩАТЬ ВВИДУ НЕУДАЧИ *)
  UNTIL (I=M) OR NOT(НАШЛИ);
UNTIL НАШЛИ OR НЕГДЕ-ИСКАТЬ;
IF НАШЛИ THEN WRITELN(КАНДИДАТ)
ELSE WRITELN(' НЕТ');
END;

```

Можно показать, что время работы этой процедуры не превосходит $S \cdot M \cdot N$. Если бы мы не пользовались упорядоченностью элементов и искали очередного кандидата с начала каждой строки, время работы возросло бы до величины порядка $M \cdot N^2$.

Задачи 3, 4 и 6 предлагались в 1985 г. участникам олимпиады по программированию в учебно-производственном центре вычислительной техники Октябрьского района г. Москвы. Много интересных задач, предлагавшихся на олимпиадах в прежние годы, приведено в книге [3].

3.8. Тестирование и отладка программ

Тестирование — это процесс выполнения программы с целью установить наличие в ней ошибок. Чем раньше будут выявлены ошибки, тем меньше вреда они принесут. Известно, например, что запуск первой американской станции к Венере окончился неудачей вследствие одной ошибки в программе. Пока не существует теории тестирования, применение которой гарантировало бы выявление всех ошибок. Мы приведем несколько практически полезных правил тестирования, следование которым позволяет уменьшить число необнаруженных ошибок по сравнению с бессистемной проверкой программы.

1. Следует готовить не только исходные данные для тестов, но и заранее находить результаты, которые должны получиться. Лучше всего сравнивать эталонные и полученные результаты в самом тесте. Такие тесты называются самопроверяющимися. Если не знать заранее результатов теста, очень легко принять неверные результаты за правильные.

2. Очень важно, чтобы программа не только давала правильные результаты для корректных исходных данных, но и осмысленно реагировала на некорректные данные. Поэтому среди наборов тестовых данных обязательно должны быть представлены некорректные исходные данные.

3. Составление тестов следует начинать до составления программы. Когда выявлены случаи, подлежащие проверке, программу писать проще, а учет возможных ошибок в исходных данных сделает программу устойчивее.

4. Составление тестов продолжается параллельно с разработкой программы. Как только в программе пишется условная инструкция, тестовые данные пополняются, чтобы обеспечить проверку работы программы и для истинного, и для ложного условия. Как только пишется инструкция цикла, следует обеспечить проверку работы программы в случаях, когда цикл не выполняется ни разу, один раз или несколько раз.

5. В тестах должны быть проверены все крайние случаи. Среди тестовых данных обязательно должны быть значения, граничные между допустимыми и недопустимыми, а также значения, которые по условию задачи должны обрабатываться особым образом.

6. Следует тщательно анализировать результаты выполнения тестов, иначе тестирование просто теряет смысл.

7. Набор тестов следует хранить, чтобы при необходимости повторить тестирование (например, после исправления программы).

Пропилюстрируем применение сформулированных правил двумя примерами. Сначала составим тесты для программы решения квадратных уравнений. Нам нужно проверить следующие случаи: 1) старший коэффициент равен нулю (некорректные исходные данные), 2) корни комплексные, 3) корни совпадают (крайний случай), 4) один из корней равен нулю (еще один крайний случай), 5) уравнение имеет два различных корня, не равных нулю. Сведем тестовые данные и ожидаемые результаты в таблицу (см. табл. 3.1).

Таблица 3.1

Тесты для программы решения квадратных уравнений

А	В	С	Корни уравнения
0	0	0	Корень — любое число
0	0	5	Корней нет
0	2	-4	2
1	-2	2	Корни комплексные: $X_1 = 1 - i$ $X_2 = 1 + i$
1	-2	1	1 1
1	-2	0	0 2
6	-13	6	2/3 3/2

Программа решения квадратных уравнений, приведенная в разд. 2.4, справится со всеми тестами, кроме трех первых. При их выполнении произойдет деление на нуль. В конце данного раздела мы модифицируем программу, чтобы она успешно справлялась со всеми тестами.

Рассмотрим теперь второй пример. Требуется составить программу, читающую три целых числа, задающих длины сторон треугольника, и определяющую вид треугольника (правильный, равнобедренный или разносторонний).

Начнем с составления тестов к этой программе. Сначала проверим длины сторон, из которых треугольник составить нельзя, включая случаи, когда одна из сторон равна сумме двух других. Затем проверим длины сторон правильного треугольника, после этого — равнобедренность по каждой паре сторон и, наконец, изучим случай разностороннего треугольника. Такой набор тестов

Таблица 3.2

Тесты для программы определения вида треугольника

Длины сторон	Треугольник	Длины сторон	Треугольник
0 0 0	Не существует	5 3 3	Равнобедренный
2 2 4	Не существует	4 6 4	Равнобедренный
1 3 —1	Не существует	5 5 7	Равнобедренный
1 1 5	Не существует	9 8 7	Разносторонний
4 4 4	Правильный		

(см. табл. 3.2) заставит нас и при составлении программы учесть все возможности.

Составим программу. Когда мы точно знаем, какие случаи нуждаются в рассмотрении, сделать это нетрудно:

```

PROGRAM ВИД-ТРЕУГОЛЬНИКА (INPUT,OUTPUT);
VAR A,B,C: INTEGER;
(* A,B,C — ДЛИНЫ СТОРОН ТРЕУГОЛЬНИКА,
   ВИД КОТОРОГО ТРЕБУЕТСЯ ОПРЕДЕЛИТЬ *)
BEGIN READ (A,B,C);
      WRITE (' ТРЕУГОЛЬНИК СО СТОРОНАМИ: 'A,B,C);
      IF (A>=B+C) OR (B>=A+C) OR (C>=A+B) THEN
        WRITELN (' НЕ СУЩЕСТВУЕТ')
      ELSE
        IF (A=B) AND (B=C) THEN WRITELN (' ПРАВИЛЬНЫЙ')
        ELSE
          IF (A=B) OR (B=C) OR (A=C) THEN

```

```

WRITELN(' РАВНОБЕДРЕННЫЙ')
ELSE WRITELN(' РАЗНОСТОРОННИЙ');
END.

```

Выясним теперь, что делать, когда тестирование показало, что в программе есть ошибки. Процесс поиска ошибок и их исправления называется *отладкой*. Ошибки могут проявляться по-разному. Самый простой случай, когда происходит аварийное окончание программы (деление на нуль, выход за границу массива и т. п.). При этом паскаль-машина указывает инструкцию, выполнение которой привело к аварии, и значения переменных. Такую аварийную выдачу, называемую *дампом*, необходимо тщательно изучить. После того как ошибка найдена, необходимо проверить, согласуются ли с ней значения всех переменных. Если нет, следует поискать другие ошибки, объясняющие полученный результат. Только после того, как выдача объяснена полностью, следует приступить к исправлению ошибок.

Труднее всего найти ошибку, когда программа работает «почти правильно» (например, работает правильно почти для всех исходных данных или даже всегда выдает правильный результат, но время ее работы не отвечает нашим ожиданиям). В этом случае лучшее, что можно посоветовать, — это положить перед собой текст программы и стараться вникнуть в нее. Замечено, что многочисленные запуски программы с выдачей промежуточных результатов не способны сэкономить время при поиске ошибок.

При исправлении программы следует постараться мысленно охватить всю программу, чтобы, исправив одно, не испортить другое. Не следует бояться существенных переделок. Дело в том, что мелкие исправления, как правило, ухудшают структуру программы, и в какой-то момент программа может оказаться неуправляемой.

Рассмотрим процесс поиска ошибки на примере программы разложения натурального числа на простые множители. Первоначально была написана программа, которая могла обрабатывать одно число и выглядела следующим образом:

```

PROGRAM РАЗЛОЖЕНИЕ (INPUT, OUTPUT);
(* РАЗЛОЖЕНИЕ НА ПРОСТЫЕ МНОЖИТЕЛИ *)
VAR X, M: INTEGER; (* X — ЧИСЛО, M — МНОЖИТЕЛЬ *)
BEGIN READ (X); M:=2;
  WRITELN(' РАЗЛОЖЕНИЕ ЧИСЛА', X,
    ' НА ПРОСТЫЕ МНОЖИТЕЛИ');
  WHILE M <= X DO
    IF (X MOD M)=0 THEN BEGIN
      WRITE(' ', M:1); X:=X DIV M
    END
    ELSE M:=M+1;

```

```
WRITELN.  
END.
```

Эта программа правильная. Затем потребовалось, чтобы она могла обрабатывать последовательности чисел (признак конца — 0). Программу переделали так:

```
PROGRAM РАЗЛОЖЕНИЕ(INPUT,OUTPUT);  
(* РАЗЛОЖЕНИЕ НА ПРОСТЫЕ МНОЖИТЕЛИ *)  
VAR X,M: INTEGER; (* X — ЧИСЛО, M — МНОЖИТЕЛЬ *)  
BEGIN READ(X); M:=2;  
  WHILE X<>0 DO BEGIN  
    WRITELN(' РАЗЛОЖЕНИЕ ЧИСЛА',X,  
      ' НА ПРОСТЫЕ МНОЖИТЕЛИ');  
    WHILE M<=X DO  
      IF (X MOD M)=0 THEN BEGIN  
        WRITE(' ',M:1); X:=X DIV M  
      END  
      ELSE M:=M+1;  
    WRITELN;  
    READ(X)  
  END  
END.
```

При обработке последовательности чисел

25 12355 507 304 0

программа напечатала

```
РАЗЛОЖЕНИЕ ЧИСЛА    25  НА ПРОСТЫЕ МНОЖИТЕЛИ  
5 5  
РАЗЛОЖЕНИЕ ЧИСЛА 12355 НА ПРОСТЫЕ МНОЖИТЕЛИ  
5 7 353  
РАЗЛОЖЕНИЕ ЧИСЛА    507 НА ПРОСТЫЕ МНОЖИТЕЛИ  
507  
РАЗЛОЖЕНИЕ ЧИСЛА    304 НА ПРОСТЫЕ МНОЖИТЕЛИ
```

Для первых двух чисел программа проработала правильно, зато число 507 оказалось простым (хотя делится на 3), а от разложения числа 304 вообще нет следов! Будем искать ошибку, отираясь от последнего обстоятельства. Нет следов — значит, мы не пытались делить даже на 2. Но как же избежать деления на 2, если M принимает это значение в самом начале! Дело в том, что M получает значение 2 даже слишком близко к началу — перед циклом обработки элементов последовательности. После выхода

из цикла разложения одного числа множитель M равен наибольшему простому множителю числа и поиск простых множителей следующего числа начинается с этого значения. Для числа 12 355 результат оказался правильным случайно — перед ним раскладывали 25, а минимальный делитель 12 355 равен 5. Число 507 программа еще пыталась раскладывать, правда, начав перебор делителей с 353. Зато при разложении 304 условие продолжения цикла оказалось ложным с самого начала.

Выдача объяснена полностью, можно исправлять ошибку. Для этого достаточно поместить инструкцию $M:=2$ непосредственно перед внутренним циклом. Причина же ошибки в том, что при модификации программы оказалась рассечена связанная группа действий — начальная установка делителя и цикл разложения. Возможно, этого бы не произошло, если бы две инструкции — $READ(X)$ и $M:=2$ — не были записаны в одной строке.

Теперь переделаем программу решения квадратных уравнений так, чтобы избавиться от деления на нуль:

```
PROGRAM КОРНИ(INPUT, OUTPUT);
VAR A,B,C,D,X1, X2, REX,IMX: REAL;
BEGIN READ(A,B,C);
  WRITELN (' КОЭФФИЦИЕНТЫ КВАДРАТНОГО',
    ' УРАВНЕНИЯ');
  WRITELN(' A=',A,' B=',B,' C=', C);
  IF A<>0 THEN BEGIN
    D:=B*B-4*A*C;
    IF D<0 THEN BEGIN
      WRITE(' КОРНИ КОМПЛЕКСНЫЕ');
      D:=SQRT(-D); REX:=-B/(2*A);
      IMX:=D/(2*ABS(A));
      WRITELN(' X1=',REX,'-', IMX,'*I',
        ' X2=',REX,'+', IMX,'*I');
    END
  ELSE BEGIN
    WRITE(' КОРНИ ДЕЙСТВИТЕЛЬНЫЕ');
    D:=SQRT(D); X1:=(-B-D)/(2*A);
    X2:=(-B+D)/(2*A);
    WRITELN(' X1=',X1,' X2=',X2);
  END
ELSE (* A=0 *)
  IF B<>0 THEN WRITELN(' ОДИН КОРЕНЬ X=',-B/C)
  ELSE (* A=B=0 *)
    IF C=0 THEN WRITELN(' КОРЕНЬ — ЛЮБОЕ ЧИСЛО')
    ELSE WRITELN(' КОРНЕЙ НЕТ');
END.
```

Может показаться обидным усложнить программу ради маловероятных случаев. Еще обиднее, однако, будучи пользователем программы решения квадратных уравнений, получить сообщение ДЕЛЕНИЕ НА НУЛЬ.

3.9. Переборные задачи

3.9.1. Полный перебор. Инструкции перехода. Мы уже неоднократно встречались с задачами, где требовалось обработать (перебрать) все элементы некоторого конечного множества. Для решения таких задач использовались разновидности следующей схемы программы:

(* СХЕМА 1 *)

```
VAR X:ЭЛЕМЕНТ МНОЖЕСТВА;  
X:=ПЕРВЫЙ ЭЛЕМЕНТ; ОБРАБОТАТЬ X;  
WHILE X<> ПОСЛЕДНИЙ ЭЛЕМЕНТ DO BEGIN  
  X:=СЛЕДУЮЩИЙ ЗА X;  
  ОБРАБОТАТЬ X;  
END;
```

Иногда удастся добавить к множеству еще один, фиктивный элемент, не подлежащий обработке, который располагается либо перед первым обрабатываемым элементом, либо вслед за последним. В этом случае обработку первого элемента можно выполнить сразу со всеми в цикле:

(* СХЕМА 2 *)

```
X:=ФИКТИВНЫЙ ЭЛЕМЕНТ;  
REPEAT  
  X:=СЛЕДУЮЩИЙ ЗА X;  
  ОБРАБОТАТЬ X;  
UNTIL X=ПОСЛЕДНИЙ ЭЛЕМЕНТ;
```

(* СХЕМА 3 *)

```
X:=ПЕРВЫЙ ЭЛЕМЕНТ;  
REPEAT  
  ОБРАБОТАТЬ X;  
  X:=СЛЕДУЮЩИЙ ЗА X;  
UNTIL X=ФИКТИВНЫЙ ЭЛЕМЕНТ;
```

Для конкретизации этих схем необходимо ответить на два вопроса.

1. Как будут записываться в программе элементы множества (тип ЭЛЕМЕНТ МНОЖЕСТВА)?

2. В каком порядке мы будем перебирать их (инструкция X:=СЛЕДУЮЩИЙ ЗА X)?

В простейших случаях элемент множества можно представить одним числом. Пример такой задачи — поиск заданного числа в одномерном массиве. Здесь для указания элемента массива достаточно задать его индекс. Рассмотрим более сложный случай, когда элементы интересующего нас множества сами имеют некоторую структуру. (Именно такие задачи обычно относят к переборным.) Во многих случаях элементы множества естественно записывать в виде последовательности чисел фиксированного или переменного размера. Рассмотрим несколько типичных примеров.

Первая задача состоит в том, чтобы подсчитать число счастливых билетов, т. е. таких наборов из шести цифр A, B, C, D, E, F, что $A+B+C=D+E+F$. Будем решать задачу путем перебора всех комбинаций из шести цифр (это решение далеко не самое эффективное). Таким образом, элемент множества задается последовательностью из шести цифр, каждая из которых принимает значения от 0 до 9. Для хранения этой последовательности используем шесть отдельных переменных. В данном случае можно, не следуя буквально приведенным выше схемам, организовать перебор с помощью шести вложенных циклов FOR:

```
PROGRAM СЧАСТЛИВЫЕ_БИЛЕТЫ(OUTPUT);
(* ПОДСЧЕТ ЧИСЛА СЧАСТЛИВЫХ БИЛЕТОВ
ПОЛНЫМ ПЕРЕБОРОМ *)
VAR A,B,C,D,E,F:0..9; N:INTEGER;
(* A—F—ЦИФРЫ БИЛЕТА
N— ЧИСЛО НАЙДЕННЫХ СЧАСТЛИВЫХ БИЛЕТОВ *)
BEGIN N:=0;
  FOR A:=0 TO 9 DO
    FOR B:=0 TO 9 DO
      FOR C:=0 TO 9 DO
        FOR D:=0 TO 9 DO
          FOR E:=0 TO 9 DO
            FOR F:=0 TO 9 DO
              (* ОБРАБОТКА ЭЛЕМЕНТА *)
              IF A+B+C=D+E+F THEN N:=N+1;
            WRITELN(' ЧИСЛО СЧАСТЛИВЫХ БИЛЕТОВ =',N);
          END.
```

Кстати сказать, счастливых билетов 55 252 (если считать и номер 000000), что составляет примерно $1/18$ общего числа билетов.

Теперь усложним задачу. Допустим, что длина обрабатываемых последовательностей по-прежнему постоянна, но не известна заранее, а задается в качестве исходного данного. Требуется перебрать и обработать (например, выделить счастливые) все последовательности из K цифр ($K \leq 10$). Для хранения последовательности придется использовать массив, в элементах 1..K которого будут записаны

отдельные цифры. Чтобы перейти от одной последовательности к следующей, нужно прибавить 1 к последней (т. е. К-й) цифре. Это, однако, можно сделать, только если она не равна 9. Если последняя цифра 9, ее надо заменить на 0 и попытаться прибавить 1 к предпоследней цифре и т. д. Эти действия записываются в виде следующего фрагмента программы:

X:=СЛЕДУЮЩИЙ ЗА X:

VAR X:ARRAY [1..10] OF 0..9; (* X [1..K] — ЦИФРЫ *)

K, I:INTEGER;

I:=K;

WHILE X[I]=9 DO BEGIN

X[I]:=0; I:=I-1;

END; (* X[I]<>9 *)

X[I]:=X[I]+1;

Для организации перебора воспользуемся схемой 3, поскольку она упрощает проверку окончания (проверка условия X=ПОСЛЕДНИЙ ЭЛЕМЕНТ требует сравнения всех цифр с 9). Вслед за последним элементом — последовательностью из всех девяток — по логике программы перехода к следующему должна стоять последовательность 100...0, в которой цифра 1 попадает на «нулевой» место, стоящее перед всеми компонентами последовательности. Введя для хранения этой компоненты дополнительную компоненту массива с индексом 0, приходим к программе:

PROGRAM НЕИЗВЕСТНАЯ-ДЛИНА (INPUT,OUTPUT);

VAR X:ARRAY [0..10] OF 0..9; K, I:INTEGER;

(* X [1..K] — КОМПОНЕНТЫ ПОСЛЕДОВАТЕЛЬНОСТИ *)

BEGIN READ(K);

(* X:=ПЕРВЫЙ ЭЛЕМЕНТ *)

FOR I:=0 TO K DO X [I]:=0;

REPEAT

(***** *)

(* В ЭТОМ МЕСТЕ ДОЛЖНА ВЫПОЛНЯТЬСЯ *)

(* ОБРАБОТКА ПОСЛЕДОВАТЕЛЬНОСТИ X[1..K] *)

(***** *)

(* X:=СЛЕДУЮЩИЙ ЗА X *)

I:=K;

WHILE X[I]=9 DO BEGIN

X [I]:=0; I:=I-1;

END; (* X[I]<>9 *)

X [I]:=X[I]+1;

UNTIL X[0]=1;

END.

Порядок, в котором выполняется перебор последовательностей в приведенных выше программах, называется лексикографическим.

Напомним: последовательность $a_1, \dots, a_m$ лексикографически меньше последовательности $b_1, \dots, b_n$, если найдется k такое, что (для всех $i \leq k$ $a_i = b_i$) и ($a_{k+1} < b_{k+1}$ или $k = m < n$). Иными словами, из двух несовпадающих последовательностей лексикографически меньше та, у которой первая несовпадающая компонента меньше или вовсе отсутствует. Именно в лексикографическом порядке расположены слова в словарях. Из определения видно, что для получения последовательности, лексикографически следующей за данной, следует увеличить компоненту по возможности с наибольшим номером. Так и поступают приведенные выше программы, работающие с последовательностями фиксированной длины. Если же допустить изменение длины последовательности, то лексикографически следующая за данной последовательностью будет получаться из нее приписыванием справа минимальной из возможных компонент (так, в слове за словом «пар» стоит «пара», а не «пас»).

Рассмотрим организацию перебора для самого общего случая, когда множество состоит из последовательностей различной длины и, кроме того, в него включены не все вообще последовательности из заданных компонент (их бесконечно много), а лишь конечная часть. Будем называть последовательности, подлежащие перебору, хорошими, а остальные плохими. Не конкретизируя пока способ разбиения последовательностей на хорошие и плохие, потребуем от него, однако, выполнения свойства локальности: плохую последовательность нельзя превратить в хорошую путем добавления справа новых компонент. Из свойства локальности вытекает, что, получив в процессе перебора плохую последовательность, ее уже не следует расширять, а нужно пытаться преобразовать в хорошую, изменяя последние компоненты. Инструкции для перехода от хорошей последовательности к следующей хорошей могут выглядеть так (считаем, что 0 — минимальная компонента последовательностей):

```
X:=СЛЕДУЮЩИЙ ХОРОШИЙ ЗА X;
VAR X:ПОСЛЕДОВАТЕЛЬНОСТЬ;
ПРИПИСАТЬ К X СПРАВА 0;
WHILE ПЛОХОЙ(X) DO
    УВЕЛИЧИТЬ ПОСЛЕДНИЕ КОМПОНЕНТЫ X;
```

Действие увеличения последних компонент похоже на рассмотренный выше фрагмент программы нахождения следующего за X в случае фиксированной длины, но отличается способом обработки последних девяток. Действительно, вслед за последовательностью, оканчивающейся на девятки, например 199, следует рассматривать не 200, а просто 2 — эта последовательность лексикографически предшествует 200. Таким образом, заключительные девятки следует заменять на нули, а просто отбрасывать. Реализовав последовательность переменной длины с помощью массива X , содержащего

компоненты, и переменной XL, в которой записана длина, получим следующую конкретизацию предыдущего фрагмента программы:

```
X:=СЛЕДУЮЩИЙ ХОРОШИЙ ЗА X:
VAR XC:ARRAY [1..LMAX] OF 0..9: XL:0..LMAX;
  (* XC [1..XL] — КОМПОНЕНТЫ
    ПОСЛЕДОВАТЕЛЬНОСТИ X *)
(* ПРИПИСАТЬ К X СПРАВА 0 *)
XL:=XL+1; XC [XL]:=0;
WHILE ПЛОХОЙ(X) DO BEGIN
  (* УВЕЛИЧИТЬ ПОСЛЕДНИЕ КОМПОНЕНТЫ *)
  WHILE XC [XL]=9 DO
    (* ОТБРОСИТЬ ПОСЛЕДНЮЮ ЦИФРУ *)
    XL:=XL-1;
  XC [XL]:=XC [XL]+1;
END;
```

В качестве конкретного примера рассмотрим задачу порождения всех последовательностей из не более чем K различных цифр. Условие хорошей последовательности запишется следующим образом:

хороший(X)=(все компоненты X различны) и $(XL \leq K)$

Можно просто запрограммировать проверку этого условия и подставить его отрицание вместо ПЛОХОЙ(X) в предыдущий фрагмент программы. Однако такое решение будет неэффективным, и его можно улучшить по крайней мере по двум направлениям.

Во-первых, для проверки, хорошая ли последовательность, достаточно сравнить только ее последнюю компоненту со всеми остальными. Действительно, группа инструкций для получения следующего хорошего за X применяется к уже хорошим последовательностям, а все последовательности, получающиеся в процессе ее работы, совпадают с исходной хорошей во всех своих компонентах, кроме последней.

Второй источник неэффективности — то, что инструкции для получения следующего хорошего будут пытаться расширить хорошие последовательности длины K, получая последовательности длины $K+1$, которые затем будут отвергаться проверкой ПЛОХОЙ(X). Вместо этого их лучше вовсе не получать. Последовательности длины K нужно не расширять, а сразу увеличивать у них последнюю цифру (тогда можно и не проверять превышение длины в функции ПЛОХОЙ).

В этой задаче нет элемента, который получался бы естественным образом вслед за последним обрабатываемым элементом 987... . Зато есть элемент, предшествующий первому, — это пустая последовательность. Воспользуемся поэтому схемой 2 перебора. Провер-

ка условия $X = \text{ПОСЛЕДНИЙ ЭЛЕМЕНТ}$ требует анализа всех компонент последовательности. Обидно записывать этот анализ, когда последний элемент можно очень легко выявить, попытавшись построить следующий за ним. В этом случае цикл увеличения последних компонент зайдет в тупик, поскольку будут отброшены все компоненты последовательности. Чтобы воспользоваться этим, мы должны суметь завершить работу всей программы при выполнении некоторого условия, обнаруживаемого во внутреннем цикле.

Для выполнения столь радикальных изменений хода программы служит *инструкция перехода*. Она записывается следующим образом:

GOTO <метка>;

где <метка> — некоторое натуральное число. Кроме того, в программе должна присутствовать некоторая инструкция, *помеченная* данной меткой. Она имеет вид

<метка> : <инструкция>;

Выполнение инструкции перехода GOTO <метка> предписывает продолжить выполнение программы не со следующей инструкцией, а с инструкцией, помеченной данной меткой. Все используемые в программе метки должны быть описаны. Описание меток выглядит так:

LABEL <метка 1>, <метка 2>, ..., <метка N>;

и размещается сразу за заголовком программы перед всеми другими описаниями и определениями.

Следует избегать неумеренного использования инструкций перехода — это приводит к запутанной, трудной для понимания программе.

Полная программа выглядит так:

PROGRAM ПЕРЕМЕННАЯ_ДЛИНА (INPUT, OUTPUT);

LABEL 99;

CONST LMAX=10; (* ПОСЛЕДОВАТЕЛЬНОСТЬ ИЗ РАЗНЫХ ЦИФР НЕ ДЛИННЕЕ 10 *)

VAR XC:ARRAY[1..LMAX] OF 0..9; XL:0..LMAX; K:INTEGER;

(* XC [1..XL] — КОМПОНЕНТЫ ПОСЛЕДОВАТЕЛЬНОСТИ X *)

FUNCTION ПЛОХОЙ-X:BOOLEAN;

(* ПРОВЕРКА, СОДЕРЖИТ ЛИ X СОВПАДАЮЩИЕ КОМПОНЕНТЫ ПРИ УСЛОВИИ, ЧТО XC[1..XL-1] — ВСЕ РАЗНЫЕ *)

VAR I:INTEGER;

BEGIN I:=1;

WHILE XC[I] <> XC [XL] DO I:=I+1;

ПЛОХОЙ-X:=I <> XL;

```

END: (* ПЛОХОЙ-Х *)
BEGIN (* ГЛАВНАЯ ПРОГРАММА *)
  READ(K);
  WRITELN(' ПОРОЖДЕНИЕ ВСЕХ',
    ' ПОСЛЕДОВАТЕЛЬНОСТЕЙ ИЗ <= ', K,
    ' РАЗЛИЧНЫХ ЦИФР');
  IF K>LMAX THEN K:=LMAX;
  (* X:=ПУСТАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ *) XL:=0;
  REPEAT
    (* X:=СЛЕДУЮЩИЙ ХОРОШИЙ ЗА Х *)
    IF XL<K THEN (* ПРИПИСАТЬ СПРАВА 0 *) BEGIN
      XL:=XL+1; XC[XL]:=0 END
    ELSE (* УВЕЛИЧИТЬ ПОСЛЕДНИЕ КОМПОНЕНТЫ *)
      BEGIN
        WHILE XC[XL]=9 DO (* ОТБРОСИТЬ ПОСЛЕДНЮЮ
          ЦИФРУ *)
          BEGIN XL:=XL-1; IF XL=0 THEN GOTO 99 END;
          XC[XL]:=XC[XL]+1;
        END;
        WHILE ПЛОХОЙ-Х DO (* УВЕЛИЧИТЬ ПОСЛЕДНИЕ
          КОМПОНЕНТЫ *)
          BEGIN
            WHILE XC[XL]=9 DO (* ОТБРОСИТЬ ПОСЛЕДНЮЮ
              ЦИФРУ *)
              BEGIN XL:=XL-1; IF XL=0 THEN GOTO 99 END;
              XC[XL]:=XC[XL]+1;
            END;
            (* В ЭТОМ МЕСТЕ ДОЛЖНЫ СТОЯТЬ ИНСТРУКЦИИ,
              ОПИСЫВАЮЩИЕ ОБРАБОТКУ
              ПОСЛЕДОВАТЕЛЬНОСТИ Х,
              НАПРИМЕР ИНСТРУКЦИИ ПЕЧАТИ Х *)
            UNTIL FALSE; (* ВЫХОД ИЗ ЭТОГО ЦИКЛА ТОЛЬКО
              ПО GOTO *)
          99: WRITELN(' ОБРАБОТКА ЗАВЕРШЕНА');
        END.

```

Задача. Последовательность из разных цифр не может кончаться на 99. Заменим поэтому циклы WHILE XC[XL]=9 DO ... на условные инструкции IF XC[XL]=9 THEN ... Объясните, почему полученная программа будет неправильной.

Очень многие переборные задачи можно свести к перебору последовательностей. Примером такой задачи (в действительности она эквивалентна задаче перебора последовательностей) является задача прохождения деревьев. Дерево — это граф без циклов, в котором выделена одна вершина, называемая корнем. Для каждой вер-

линии дерева существует единственный путь в нее из корня. Задача состоит в том, чтобы «обойти» все вершины дерева. Если перенумеровать все ребра, выходящие из каждой вершины, то путь из корня, однозначно сопоставляемый каждой вершине, можно будет в свою очередь задать с помощью последовательности номеров проходимых ребер. Лексикографический порядок перебора последовательностей отвечает способу прохождения дерева, известному как *перебор в глубину*. Находясь в какой-либо вершине, мы сначала пытаемся пойти «в глубину», т. е. удалиться от корня; если это невозможно, то мы возвращаемся назад и пытаемся пойти в глубину по следующему ребру.

3.9.2. Сокращение перебора. Динамическое программирование. Многие практические задачи сводятся к отысканию минимума некоторой функции на множестве. Искомым элементом множества часто является функция от времени (например, ищется закон управления самолетом, обеспечивающий наибо́льшее достижение заданной высоты и скорости). Эту функцию аппроксимируют последовательностью ее значений в дискретные моменты времени, получая задачу поиска минимума в множестве последовательностей. Можно осуществлять поиск минимума путем полного перебора всех допустимых последовательностей. Метод динамического программирования позволяет в ряде случаев сократить перебор.

Будем предполагать, что определена функция F , которая сопоставляет некоторое число каждой последовательности из некоторого конечного множества допустимых последовательностей. Задача состоит в отыскании такой последовательности $a_1, a_2, \dots, a_N$, которая минимизирует $F(a_1, \dots, a_N)$ (часто заранее задается длина последовательности и (или) ее первая и последняя компоненты). Метод динамического программирования применим при условии, что функция F удовлетворяет *принципу оптимальности Беллмана*. Этот принцип требует, чтобы всякий отрезок $a_K, a_{K+1}, \dots, a_M$ оптимальной последовательности $a_1, \dots, a_N$ ($1 \leq K \leq M \leq N$) был бы сам оптимальным (т. е. давал бы минимальное значение функции $F(a_K, \dots, a_M)$) среди всех последовательностей, совпадающих с ним в крайних компонентах и по числу компонент.

Это свойство позволяет находить оптимальную последовательность, постепенно удлиняя уже найденные оптимальные отрезки. Действительно, пусть для всех возможных последних компонент (которые мы обозначим числами от 1 до M) мы уже нашли оптимальные последовательности длины L (обозначим их $P_1, P_2, \dots, P_M$), оканчивающиеся соответственно на 1, 2, ..., M . Тогда всякая оптимальная последовательность длины $L+1$ должна в качестве своих первых L компонент содержать одну из этих последовательностей, и для нахождения оптимальной последовательности длины $L+1$, оканчивающейся компонентой K , следует сравнить и выбрать

по минимуму F одну из M последовательностей $(P_1, K), (P_2, K), \dots, (P_M, K)$ ((P_j, K) обозначает последовательность, получаемую приписыванием K к P_j). Для нахождения всего набора оптимальных последовательностей длины $L+1$ (оканчивающихся на все возможные компоненты) нам следует проделать эти действия для всех K от 1 до M , т. е. сравнить M^2 последовательностей, а построение последовательности длины L с самого начала потребует L таких шагов, что дает время работы алгоритма, пропорциональное $L \cdot M^2$. Вместе с тем, полный перебор всех последовательностей длины L , каждая компонента которых принимает одно из M значений, требует сравнения M^L последовательностей, что почти всегда много больше предыдущего числа.

Важным классом функций, заведомо удовлетворяющих принципу Беллмана, являются аддитивные функции, равные сумме некоторых функций, зависящих только от соседних компонент последовательности:

$$F(a_1, \dots, a_N) = \sum_{i=1}^{N-1} G(a_i, a_{i+1}).$$

Рассмотрим программную реализацию метода динамического программирования на конкретном примере. Пусть компоненты последовательности принимают значения от 1 до M , а функция G задана таблицей своих значений, которую поместим в двумерный массив. Требуется найти последовательность $a_1, \dots, a_L$ заданной длины L , минимизирующую сумму

$$G[a_1, a_2] + G[a_2, a_3] + \dots + G[a_{L-1}, a_L].$$

В данном случае удобнее наращивать последовательность не с конца, а с начала, приписывая компоненты слева. Будем строить оптимальные последовательности $a_1, a_{1+1}, \dots, a_L$ с заданной первой компонентой a_1 , постепенно уменьшая I . Для запоминания всех этих последовательностей используем двумерный массив S , причем в $S[I, K]$ запишем $(I+1)$ -ю компоненту a_{I+1} оптимальной последовательности, начинающейся с I -й компоненты, равной K . По этому массиву легко восстановить всю последовательность. Действительно, найдя $(I+1)$ -ю компоненту, мы определим $(I+2)$ -ю, обратившись к $S[I+1, a_{I+1}]$, и т. д. Кроме того, будем хранить значение функции F на оптимальной последовательности, начинающейся с $a_1 = K$. Обозначим это число $F(I, K)$. В соответствии с методом динамического программирования $F(I, K)$ определяется по следующей формуле:

$$F(I, K) = \min_Q G[K, Q] + F(I+1, Q), \quad Q = 1, \dots, M.$$

Здесь Q пробегает все возможные значения компоненты a_1 , $F(I+1, Q)$ — «цена» оптимальной последовательности, начинающейся с

$a_{I+1}=Q$, а $G[K, Q]+F(I+1, Q)$ — значение функции F на последовательности, начинающейся с $a_I=K$, $a_{I+1}=Q$. По отношению к переменной I функция $F(I, K)$ определена рекуррентной формулой первого порядка, поэтому в соответствии с разд. 3.3 используем для хранения F два одномерных массива: $FC[K]$ будет хранить текущее значение, $F(I, K)$, а $FP[K]$ — предыдущее, $F(I+1, K)$:

```

PROGRAM ДИНАМИЧЕСКАЯ (INPUT, OUTPUT);
(* МЕТОД ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ
   ДЛЯ АДДИТИВНОЙ ФУНКЦИИ *)
CONST M=...; L=...; L1=...; (* L1=L-1 *)
(* L — ДЛИНА ИСКОМОЙ ПОСЛЕДОВАТЕЛЬНОСТИ
   ИЗ КОМПОНЕНТ 1..M *)
VAR G: ARRAY [1..M, 1..M] OF REAL;
    FC, FP: ARRAY [1..M] OF REAL;
    S: ARRAY [1..L1, 1..M] OF 1..M;
(* S[I, K] — (I+1)-Я КОМПОНЕНТА ОПТИМАЛЬНОЙ
   ПОСЛЕДОВАТЕЛЬНОСТИ, ПРИ УСЛОВИИ ЧТО I-Я РАВНА
   K *)
I, J, K, Q: INTEGER; FMIN: REAL;
BEGIN
  WRITELN(' НАХОЖДЕНИЕ ПОСЛЕДОВАТЕЛЬНОСТИ',
    ' A[1], ..., A[L:1,
    ' ], МИНИМИЗИРУЮЩЕЙ СУММУ ПО I G[A [I], A[I+1]]');
  WRITELN(' ТАБЛИЦА ФУНКЦИИ G:');
  FOR I:=1 TO M DO BEGIN
    FOR J:=1 TO M DO BEGIN
      READ(G[I, J]); WRITE(G[I, J]);
    END; WRITELN;
  END;
  (* НАЧАЛЬНАЯ УСТАНОВКА F(L, Q):=0: ЦЕНА
     ПОСЛЕДОВАТЕЛЬНОСТИ ИЗ ОДНОЙ КОМПОНЕНТЫ
     РАВНА НУЛЮ *)
  FOR Q:=1 TO M DO FC[Q]:=0.0;
  FOR I:=L1 DOWNT0 1 DO BEGIN
    FP:=FC;
    FOR K:=1 TO M DO BEGIN
      (* ВЫЧИСЛЯЕМ В FC[K] ЗНАЧЕНИЕ F(I, K);
         ЗНАЧЕНИЕ Q, ДАЮЩЕЕ МИНИМУМ G[K, Q]+
         F(I+1, Q) ЗАПОМИНАЕМ В S[I, K] *)
      FC[K]:=G[K, 1]+FP[1]; S[I, K]:=1;
      FOR Q:=2 TO M DO
        IF G[K, Q]+FP[Q]<FC[K] THEN
          BEGIN FC[K]:=G[K, Q]+FP[Q]; S[I, K]:=Q END;
      (* END FOR Q *)
    END;
  END;

```

```

END;
END;
(* НАХОДИМ ОПТИМАЛЬНОЕ ЗНАЧЕНИЕ A[1] *)
FMIN:=FC[1]; K:=1;
FOR Q:=2 TO M DO
  IF FC[Q]<FMIN THEN BEGIN FMIN:=FC[Q]; K:=Q END;
(* ОПТИМАЛЬНАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ
НАЧИНАЕТСЯ С K *)
WRITELN(' ОПТИМАЛЬНАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ:');
WRITE(K);
(* ВЫЧИСЛЯЕМ И ПЕЧАТАЕМ ОСТАЛЬНЫЕ
КОМПОНЕНТЫ *)
FOR I:=1 TO L1 DO BEGIN
  (* ВЫЧИСЛЯЕМ (I+1)-ю КОМПОНЕНТУ *)
  K:=S[I, K]; WRITE(K);
END;
WRITELN;
END.

```

Как и в п. 3.9.1, рассмотренная задача имеет естественную формулировку на языке теории графов. Будем считать, что компоненты последовательности являются вершинами графа; тогда всей последовательности соответствует путь по графу, проходящий по перечисленным в ней вершинам. Функция G сопоставляет некоторое число каждому ребру графа; это число можно мыслить как длину ребра: $G[I, J]$ = длина ребра от вершины I к вершине J . Тогда $\sum_i G[a_i, a_{i+1}]$ есть суммарная длина пути, задаваемого последовательностью. Задача поиска оптимальной последовательности есть, таким образом, задача поиска кратчайшего пути на графе (при дополнительных ограничениях, например при заданном числе ребер или при заданных начальной и конечной вершинах). Метод динамического программирования сводит перебор всех путей к перебору вершин графа, которых для графов с циклами существенно меньше, чем путей.

4. БОЛЕЕ СЛОЖНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА ПАСКАЛЬ

4.1. Типы данных, заданные перечислением

С помощью ЭВМ можно моделировать и изучать поведение самых разнообразных объектов. Удобно при этом называть изучаемые объекты привычными именами. Типы данных, заданные перечислением, как раз и отвечают указанной потребности. Общая форма задания перечисляемого типа такова:

TYPE <имя типа>=(<имя₁>,..., <имя_n>);

где <имя₁>,..., <имя_n> — значения определяемого типа. Таким образом, определяя перечисляемый тип данных, мы задаем имя типа и все значения (константы) этого типа, причем значения обозначаются именами.

Пр и м е р:

TYPE ФИГУРА=(КОРОЛЬ, ФЕРЗЬ, ЛАДЬЯ, СЛОН, КОНЬ);
ДЕНЬ-НЕДЕЛИ=(ПОНЕДЕЛЬНИК, ВТОРНИК, СРЕДА,
ЧЕТВЕРГ, ПЯТНИЦА, СУББОТА, ВОСКРЕСЕНЬЕ);
ДОМАШНЕЕ-ЖИВОТНОЕ=(КОЗЛЕНОК, ТЕЛЕНОК,
КОРОВА, БЫК, СВИНЬЯ);

Возможность задания типа данных путем перечисления принадлежащих ему значений позволяет, как говорится, называть вещи своими именами. Без такой возможности пришлось бы при составлении и чтении программ, подобно козленку из мультфильма, постоянно твердить, что 1 — это теленок, 2 — это корова, а 3 — это бык.

Переменные перечисляемых типов описываются обычным образом. Значения таких типов можно присваивать и сравнивать. Значения упорядочены в соответствии с порядком их перечисления в определении типа. Так, КОРОЛЬ <ФЕРЗЬ, а СУББОТА > ВТОРНИК. Можно образовывать отрезки перечисляемых типов, например

TYPE РАБОЧИЙ-ДЕНЬ=ПОНЕДЕЛЬНИК..ПЯТНИЦА;

Для значений перечисляемых типов определены стандартные функции

SUCC — получение следующего значения данного типа,

PRED — получение предыдущего значения данного типа,

ORD — порядковый номер значения данного типа (значения нумеруются, начиная с 0, в порядке их перечисления в определении типа).

Так,

SUCC(СЛОН)=КОНЬ PRED(СРЕДА)=ВТОРНИК

ORD(КОРОВА)=2

Функцию SUCC нельзя применять к последнему значению типа, функцию PRED — к первому.

Поскольку к значениям перечисляемых типов применимы функции получения следующего и предыдущего значения, а также операции сравнения, такие значения могут задавать границы изменения переменной FOR-цикла. Пусть, например, массив РВ хранит количество часов, отработанных в каждый из дней недели. Подсчитаем общее рабочее время за неделю:

```
VAR Д: ДЕНЬ-НЕДЕЛИ; ОРВ: INTEGER;
```

```
РВ: ARRAY[ДЕНЬ-НЕДЕЛИ] OF INTEGER;
```

```
...
```

```
РВ[СРЕДА]:=8;
```

```
...
```

```
ОРВ:=0;
```

```
FOR Д:=ПОНЕДЕЛЬНИК TO ВОСКРЕСЕНЬЕ DO
```

```
ОРВ:=ОРВ+РВ[Д];
```

Стандартные типы данных INTEGER, BOOLEAN, CHAR также можно отнести к типам, заданным перечислением. Различие типов INTEGER и CHAR состоит лишь в том, что их значения обозначаются не именами, а особым образом, и эти значения можно вводить и выводить с помощью процедур READ и WRITE, а значения других перечисляемых типов — нельзя. Еще меньше отличается тип BOOLEAN — лишь тем, что его значения можно выводить. Следовательно, к значениям трех стандартных типов применимы функции SUCC, PRED и ORD. Например,

SUCC(6)=7 PRED(TRUE)=FALSE SUCC('0')='1'

ORD(FALSE)=0

(Результат применения функции ORD к целочисленному параметру равен этому параметру, т. е. может быть отрицательным.)

З а м е ч а н и е 1. В программе на языке паскаль одно и то же имя не может входить в определения разных перечисляемых ти-

пов. Так, имя КОНЬ нельзя включить одновременно и в определение типа ФИГУРА, и в определение типа ДОМАШНЕЕ-ЖИВОТНОЕ.

З а м е ч а н и е 2. Переменной какого-либо типа можно присваивать значение только этого типа. Оба операнда операции сравнения также должны быть одного типа. Бессмысленно интересоваться, что больше — ВТОРНИК или ЛАДЬЯ.

З а м е ч а н и е 3. Типы индексов массивов должны быть либо перечисляемыми типами, либо отрезками перечисляемых типов.

4.2. Инструкции выбора

Условные инструкции позволяют выбрать в зависимости от истинности логического выражения один из двух вариантов продолжения работы программы. Иногда бывает нужно разветвиться не на два, а на несколько направлений в зависимости от значения выражения перечисляемого типа. В этом случае удобно воспользоваться более общей инструкцией выбора, которая записывается так:

```
CASE {выражение} OF
    {константа1}, . . . , {константаk}: {инструкция1};
    . . .
    {константаl}, . . . , {константаn}: {инструкцияn};
END
```

Все перечисленные константы должны быть различными. Инструкция выбора трактуется следующим образом. Если значение выражения равно одной из перечисленных констант, выполняется инструкция, записанная после этой константы, и на этом выполнение инструкции выбора заканчивается. Если значение выражения отличается от всех перечисленных констант, паскаль-машина фиксирует ошибку. Наконец, если для некоторых значений нужно выполнить несколько инструкций, эти инструкции следует заключить между словами BEGIN и END.

П р и м е р. Пусть нужно напечатать название дня недели по его номеру, являющемуся значением переменной НОМЕР-ДНЯ (0 — понедельник, 1 — вторник и т. д.). Воспользуемся инструкцией выбора:

```
CASE НОМЕР_ДНЯ OF
    0: WRITE('ПОНЕДЕЛЬНИК');
    1: WRITE('ВТОРНИК');
    2: WRITE('СРЕДА');
    3: WRITE('ЧЕТВЕРГ');
    4: WRITE('ПЯТНИЦА');
    5: WRITE('СУББОТА');
    6: WRITE('ВОСКРЕСЕНЬЕ');
END
```


З а м е ч а н и е. Перед словом END, завершающим инструкцию выбора, точку с запятой ставить нельзя.

4.3. Множества

В языке паскаль множество — это произвольная совокупность значений перечисляемого типа. Тип множеств описывается следующим образом:

TYPE <имя типа>=SET OF <тип элементов>;

Тип элементов называется базовым типом множества.

Над множествами определены бинарные операции

+ объединение;

* пересечение;

— разность;

=, <> сравнение на равенство и неравенство;

<=, >= проверка на включение. Выражения $S1 \leq S2$ и $S2 \geq S1$ имеют значение TRUE, если множество $S1$ является подмножеством $S2$;

IN проверка на принадлежность. Выражение $C \text{ IN } S$ имеет смысл, если S — множество, а C — значение его базового типа; оно истинно, если элемент C принадлежит S .

Для задания значений-множеств следует в квадратных скобках через запятую перечислить значения или отрезки значений базового типа.

П р и м е р.

```
VAR БУКВЫ, ЦИФРЫ, ПОДМНОЖЕСТВО_ЦИФР,
```

```
    БУКВЫ-И_ЦИФРЫ: SET OF CHAR;
```

```
    C: CHAR; K: 0..9;
```

```
    ...
```

```
БУКВЫ:=['A','B','C','D','E','F','G','H','I','J','K','L','M','N',  
'O','P','Q','R','S','T','U','V','W','X','Y','Z'];
```

```
ЦИФРЫ:=['0'..'9'];
```

```
ПОДМНОЖЕСТВО_ЦИФР:=['0'..CHR(ORD('0')+K)];
```

```
БУКВЫ-И_ЦИФРЫ: = БУКВЫ + ЦИФРЫ;
```

Здесь множество цифр задано в виде отрезка значений типа CHAR. Множество букв не является, вообще говоря, единым отрезком, поэтому для его задания пришлось перечислить все его элементы. ПОДМНОЖЕСТВО_ЦИФР задано как отрезок от литеры '0' до литеры, получаемой в результате вычисления выражения.

Чтобы проверить, является ли значение переменной C буквой, достаточно вычислить логическое выражение

$C \text{ IN БУКВЫ}$

К сожалению, практическое использование множеств наталкивается на ограничения, накладываемые реализациями паскаль-машины на базовый тип. Как правило, он может содержать весьма небольшое количество значений, скажем 64 или 48. Множество литер — SET OF CHAR, причем в урезанном виде, с одними только цифрами, некоторыми знаками и прописными латинскими буквами — вот на что реально можно рассчитывать. Именно множества литер наиболее употребительны. Другие типы множеств используются редко.

4.4. Упакованные структуры данных

Известно, что технически проще изготовить устройство, способное находиться в одном из двух устойчивых состояний, чем, скажем, в десяти, поэтому современные ЭВМ работают в двоичной системе счисления. Память ЭВМ разбивается на группы по n двоичных разрядов. Такие группы называются *машинными словами*, а двоичные разряды — *битами*. Типичные значения n равны 16 (для миниЭВМ), 32 (для ЕС ЭВМ), 48 (для БЭСМ-6), 64 (для суперЭВМ). С помощью n битов можно представить 2^n значений. Значение любого типа занимает не менее одного машинного слова. Для типов INTEGER и REAL это вполне естественно, а вот для типа BOOLEAN, которому принадлежат всего два значения, или для любого другого типа, содержащего несколько десятков значений, отведение машинного слова под каждое значение может оказаться слишком расточительным.

В разд. 3.6 описан один из методов экономии памяти — понижение размерности массивов. Еще одна возможность — упаковка в одном машинном слове нескольких компонент массива. Чтобы воспользоваться этой возможностью, нужно в описании массива перед словом ARRAY поставить слово PACKED.

Пр и м е р.

```
TYPE ШКАЛА=PACKED ARRAY[1..1000000] OF BOOLEAN;  
СТРОКА=PACKED ARRAY[1..80] OF CHAR
```

К упакованным массивам применимы все операции, предназначенные для обработки массивов неупакованных. Более того, упакованные массивы литер можно сравнивать посредством операций $=$, $<>$, $<$, $<=$, $>$, $>=$. Массивы сравниваются лексикографически — из двух различных массивов литер меньше тот, у которого меньше первая несовпадающая литера.

Последовательность из N ($N > 1$) литер, заключенная в апострофы, трактуется как константа типа

```
PACKED ARRAY [1..N] OF CHAR
```

Такие константы могут присваиваться и сравниваться.

Пример.

```
VAR ИМЯ: PACKED ARRAY [1..10] OF CHAR;
```

```
ИМЯ:='ВЛАДИМИР'
```

Таким образом, упакованные массивы литер очень удобны при обработке текстов.

Следует учитывать, что упаковка может замедлить доступ к компонентам массива, поэтому пользоваться ею нужно осмотрительно. Приведем пример ситуации, где упаковка ведет к существенной экономии памяти. В реализации паскаль-машины на БЭСМ-6 для хранения данных можно использовать около 25 000 машинных слов. Если же описать массив логических значений как упакованный, каждое машинное слово будет содержать 48 компонент, а общее число компонент способно превысить миллион. Воспользуемся указанной возможностью, чтобы написать программу, печатающую четверки простых чисел, принадлежащие одному десятку (например, 11, 13, 17, 19 или 13 001, 13 003, 13 007, 13 009):

```
PROGRAM ЧЕТВЕРКИ-ПРОСТЫХ(OUTPUT);
CONST N=1000000; NS=1000; (* NS=SQRT(N) *)
(* ПРОГРАММА НАХОДИТ ЧЕТВЕРКИ ПРОСТЫХ ЧИСЕЛ
   ДО N,
   ПРИНАДЛЕЖАЩИЕ ОДНОМУ ДЕСЯТКУ,
   ПОСРЕДСТВОМ РЕШЕТА ЭРАТОСФЕНА *)
VAR J, K: INTEGER;
    S: PACKED ARRAY[2..N] OF BOOLEAN;
BEGIN
  WRITELN(' ЧЕТВЕРКИ ПРОСТЫХ ЧИСЕЛ ДО ',N,',');
  WRITELN(' ПРИНАДЛЕЖАЩИЕ ОДНОМУ ДЕСЯТКУ');
  FOR K:=2 TO N DO S[K]:=TRUE;
  (* ЧИСЛО K БУДЕТ ПРОСТЫМ, ЕСЛИ S[K]=TRUE *)
  FOR K:=2 TO NS DO
    IF S[K] THEN BEGIN
      (* ОТМЕТИМ ЧИСЛА, КРАТНЫЕ ПРОСТОМУ,
         КАК СОСТАВНЫЕ *)
      J:=K*K;
      WHILE J<=N DO BEGIN
        S[J]:=FALSE; J:=J+K
      END
    END;
  (* ПРОСЕИВАНИЕ ЧИСЕЛ ОКОНЧЕНО,
     НАПЕЧАТАЕМ ЧЕТВЕРКИ ПРОСТЫХ *)
  K:=19;
  WHILE K<=N DO BEGIN
```


IF S[K-8] AND S[K-6] AND S[K-2] AND S[K] THEN
WRITELN(K-8, K-6, K-2, K);

K:=K+10

END

END.

Упаковывать можно не только массивы, но и описываемые ниже структуры данных — файлы и записи.

4.5. Файловая структура данных

Как указывалось в гл. 2, паскаль-машина может общаться с внешним миром: вводить исходные данные и выводить результаты. Устройства, обменивающиеся информацией с паскаль-машиной, называются внешними устройствами. Они могут быть самые разнообразные: дисплей, устройство ввода перфокарт, печатающее устройство, устройство хранения информации на магнитных лентах (аналогичное бытовому магнитофону) или на магнитных дисках. Разнообразны и свойства этих устройств. Также разнообразны характеристики информации, которая поступает в паскаль-машину. Одни наборы данных настолько велики, что разместить их целиком в памяти паскаль-машины невозможно. Единственный выход — обрабатывать их по частям. Другие наборы данных не могут быть целиком помещены в память паскаль-машины потому, что они порождаются в процессе работы программы. Например, человек, сидящий за дисплеем и играющий с шахматной программой, сообщает свой очередной ход только после того, как программа ответила на предыдущий. Чтобы не потонуть в этом разнообразии, нужны языковые средства, позволяющие одинаковым образом работать с разными устройствами и с разными наборами данных. В языке паскаль таким языковым средством является файловая структура данных.

Файлом с компонентами некоторого типа T называется последовательность значений типа T, потенциально сколь угодно длинная. Файловый тип данных описывается следующим образом:

TYPE {имя типа}=FILE OF {тип компонент};

или

TYPE {имя типа}=PACKED FILE OF {тип компонент};

Пример.

TYPE ФАЙЛ-МНОЖЕСТВ=FILE OF SET OF CHAR;

TEXT=PACKED FILE OF CHAR;

На тип компонент файла накладывается только одно ограничение: он не должен быть файловым. Можно работать с файлами чисел, литер, массивов и т. д.

Чтобы одинаковым образом работать с разными внешними устройствами и наборами данных, нужно предъявлять к ним минимальные требования. В языке паскаль компоненты файла можно обрабатывать только последовательно, начиная с первой, и в каждый момент времени доступна только одна компонента.

Переменные файлового типа описываются обычным образом, например

```
VAR INPUT, OUTPUT: TEXT;
```

однако каждое описание такой переменной неявно вводит еще одну (так называемую буферную) переменную файла. Тип буферной переменной совпадает с типом компонент файла, а ее имя получается из имени файловой переменной приписыванием вертикальной стрелки, например INPUT↑, OUTPUT↑. В буферной переменной хранится та единственная компонента файла, которая доступна в данный момент.

Пусть F — переменная файлового типа. К файловым переменным применимы следующие стандартные процедуры:

RESET(F) — подготовка к просмотру файла F. Если файл не пуст, значением буферной переменной F становится первая компонента файла;

GET(F) — чтение с продвижением по файлу. Значением буферной переменной становится первая из еще не просмотренных компонент файла (если такие есть). При следующем применении процедуры GET буферная переменная получит значение следующей компоненты и т. д.;

REWRITE(F) — опустошение файла с именем F;

PUT(F) — добавление к концу файла одной компоненты, являющейся значением буферной переменной.

Стандартная функция с логическим значением

```
EOF(F)
```

(от словосочетания End of File — конец файла) выдает значение TRUE, если обработаны все компоненты файла. Только в этом случае к файлу применима процедура PUT. С другой стороны, применять процедуру GET можно только до тех пор, пока прочитаны не все компоненты файла, т. е. EOF(F) имеет значение FALSE. После обращения к процедурам RESET и GET значение буферной переменной определено, только если EOF(F) имеет значение FALSE.

Пусть нужно применить инструкцию S ко всем компонентам файла F. К цели ведет следующая последовательность действий:

```
RESET(F);  
WHILE NOT EOF(F) DO BEGIN  
  S;
```

```
GET(F)  
END
```

Пример. Подсчитаем сумму компонент файла вещественных чисел:

```
VAR F: FILE OF REAL; S: REAL;  
BEGIN  
  S:=0; RESET(F);  
  WHILE NOT EOF(F) DO BEGIN  
    (* S — СУММА ОБРАБОТАННЫХ КОМПОНЕНТ ФАЙЛА *)  
    S:=S+F↑;  
    GET(F)  
  END  
END.
```

Рассмотрим еще один пример. Пусть файл FF хранит последовательность чисел Фибоначчи. Требуется добавить к этой последовательности очередной элемент:

```
VAR FF: FILE OF INTEGER; A,B: INTEGER;  
(* A — ПРЕДПОСЛЕДНЯЯ ИЗ ПРОЧИТАННЫХ  
КОМПОНЕНТ ФАЙЛА,  
B — ПОСЛЕДНЯЯ КОМПОНЕНТА *)  
BEGIN  
  A:=1; B:=0; RESET(FF);  
  (* ТАКИЕ НАЧАЛЬНЫЕ УСТАНОВКИ ПОЗВОЛЯЮТ  
  ЕДИНООБРАЗНО  
  ТРАКТОВАТЬ СЛУЧАИ, КОГДА ФАЙЛ ПУСТ  
  ИЛИ СОДЕРЖИТ ТОЛЬКО ОДНУ КОМПОНЕНТУ *)  
  WHILE NOT EOF(FF) DO BEGIN  
    A:=B; B:=FF↑;  
    GET(FF)  
  END;  
  FF↑:=A+B;  
  PUT(FF)  
END.
```

Прежде чем добавить компоненту, мы вышли на конец файла и только после этого применили процедуру PUT.

Рассмотрим теперь задачу о слиянии упорядоченных файлов. Пусть А и В — файлы целых чисел, компоненты которых упорядочены по возрастанию. Компоненты этих файлов нужно поместить в файл С, причем файл С тоже должен быть упорядочен по возрастанию. Решение легко получается путем модификации процедуры слияния массивов:

```

TYPE ФАЙЛ-ЦЕЛЫХ=FILE OF INTEGER;
...
PROCEDURE СЛИЯНИЕ-ФАЙЛОВ(VAR A,B,C: ФАЙЛ_ЦЕЛЫХ);
BEGIN
  RESET(A); RESET(B); REWRITE(C);
  WHILE NOT EOF(A) AND NOT EOF(B) DO
    IF A↑ < B↑ THEN BEGIN
      C↑:=A↑; PUT(C); GET(A)
    END
    ELSE BEGIN
      C↑:=B↑; PUT(C); GET(B)
    END;
  (* ЦИКЛ ОКОНЧЕН.
    ОДИН ИЗ ФАЙЛОВ ОБРАБОТАН ПОЛНОСТЬЮ.
    ПЕРЕПИШЕМ ОСТАТОК ДРУГОГО ФАЙЛА *)
  WHILE NOT EOF(A) DO BEGIN
    C↑:=A↑; PUT(C); GET(A)
  END;
  WHILE NOT EOF(B) DO BEGIN
    C↑:=B↑; PUT(C); GET(B)
  END
END;

```

З а м е ч а н и е 1. Тип TEXT, определенный в начале раздела, есть стандартный тип-языка паскаль, и включать его определение в программу не следует. Точно так же INPUT и OUTPUT — стандартные (т. е. определенные для любой паскаль-программы) файловые переменные. Обычно файл INPUT содержит литеры, вводимые с клавиатуры дисплея, а файл OUTPUT — литеры, выводимые на экран. На работу с этими файлами естественно наложить следующие ограничения: к файлу INPUT применимы только процедура GET и функция EOF, к файлу OUTPUT применима только процедура PUT.

З а м е ч а н и е 2. Пусть $C, C_1, \dots, C_n$ — переменные типа CHAR; $E, E_1, \dots, E_n$ — выражения типа CHAR; F — файловая переменная. Для упрощения работы с файлами, в частности с файлами INPUT и OUTPUT, можно использовать следующие сокращения:

Полная запись	Сокращенная запись
C:=INPUT↑; GET(INPUT)	READ(C)
C:=F↑; GET(F)	READ(F, C)
OUTPUT↑:=E; PUT(OUTPUT)	WRITE(E)
F↑:=E; PUT(F)	WRITE(F, E)
READ(C ₁);...READ(C _n)	READ(C ₁ ,...,C _n)
READ(F, C ₁);...READ(F, C _n)	READ(F, C ₁ ,...,C _n)
WRITE(E ₁);...WRITE(E _n)	WRITE(E ₁ ,...,E _n)
WRITE(F, E ₁);...WRITE(F, E _n)	WRITE(F,E ₁ ,...,E _n)

При вводе и выводе нетекстовой информации процедуры READ и WRITE работают более сложным образом. При вводе чисел процедура READ преобразует текстовое представление числа во внутреннее представление, удобное для хранения в памяти паскаль-машины и выполнения операций. Процедура WRITE, напротив, преобразует внутреннее представление значений в текстовое и затем выводит его литерой за литерой.

З а м е ч а н и е 3. Файлы типа TEXT называются текстовыми. Кроме обычной файловой они имеют дополнительную структуру: хранящиеся в них тексты делятся на строки, в конце каждой строки стоит специальный маркер. Если во время чтения текстового файла F мы доходим до маркера, буферная переменная F↑ получает значение ' ' (пробел), а стандартная функция

EOLN(F)

(от словосочетания End of Line — конец строки) выдает TRUE. При обращении за следующей литерой либо значение EOF(F) станет истинным, либо F будет присвоена первая литера очередной строки, а значение EOLN(F) станет ложным (если только строка не пуста). Маркер не есть значение типа CHAR, его можно образовать лишь посредством обращения к процедуре WRITELN.

Обращение к стандартной процедуре

PAGE(F)

вызовет при печати текстового файла F переход к началу новой страницы.

З а м е ч а н и е 4. Мы не рассматриваем вопрос об установлении соответствия между файловыми переменными паскаль-программы и наборами данных, существующими независимо от паскаль-машины. В разных реализациях паскаль-машины это соответствие устанавливается по-разному. Отметим лишь, что файловые пере-

менные, соответствующие внешним наборам данных, должны быть перечислены в скобках после имени программы. Приведем пример — копирование одного текстового файла в другой:

```
PROGRAM КОПИЯ-ТЕКСТОВОГО-ФАЙЛА (F, G);
VAR F,G: TEXT; C: CHAR;
(* F — ИСХОДНЫЙ ФАЙЛ
   В G СОЗДАЕТСЯ ЕГО КОПИЯ *)
BEGIN
  RESET(F); REWRITE(G);
  WHILE NOT EOF(F) DO BEGIN
    WHILE NOT EOLN(F) DO BEGIN
      READ(F,C); WRITE(G,C)
    END;
    (* ПРОПУСТИМ МАРКЕР КОНЦА СТРОКИ В ФАЙЛЕ F
       И ВЫВЕДЕМ МАРКЕР В ФАЙЛ G *)
    READLN(F); WRITELN(G)
  END
END.
```

З а м е ч а н и е 5. Формальным параметрам файлового типа должно предшествовать слово VAR. Такое ограничение вызвано трудностью копирования значений-файлов.

4.6. Запись

Запись — единственный из составных типов языка паскаль, который позволяет объединять в одном значении компоненты разных типов. Такая возможность полезна, например, для хранения последовательностей цифр переменной длины. Ранее мы использовали для этого две переменные — в одной хранилась длина, в другой — сами цифры, составляющие последовательность. Обе эти переменные можно объединить в одну запись с компонентами ДЛИНА и ЦИФРЫ.

```
TYPE ПОСЛЕДОВАТЕЛЬНОСТЬ=
RECORD
  ДЛИНА: INTEGER;
  ЦИФРЫ: ARRAY [1..МАКС-ДЛИНА] OF 0..9
END;
```

Каждая компонента записи имеет свое имя. Между словами RECORD и END перечисляются имена всех компонент записи с указанием их типа. Описание компонент записи составляется по тем же правилам, что и описание переменных после слова VAR. Имена компонент могут совпадать с именами переменных или компонент записей других типов.

Определив тип ПОСЛЕДОВАТЕЛЬНОСТЬ, можно обычным образом описать переменные этого типа:

VAR X,Y: ПОСЛЕДОВАТЕЛЬНОСТЬ;

Каждая из этих переменных имеет компоненты ДЛИНА и ЦИФРЫ. Для указания в программе компонент записи используется конструкция

⟨переменная типа запись⟩.⟨имя компоненты⟩

С компонентами записей можно работать точно так же, как с индивидуальными переменными того же типа.

П р и м е р. Инструкции

X.ДЛИНА:=X.ДЛИНА+1; X.ЦИФРЫ[X.ДЛИНА]:=Y.ЦИФРЫ[1]

приписывают к последовательности X первую цифру последовательности Y. Записи целиком можно присваивать и передавать в качестве параметров процедурам.

Записи — самый гибкий способ создания новых типов данных. Многие объекты реального мира естественным образом описываются записями. Пусть, например, мы разрабатываем автоматический классный журнал, предназначенный для учета успеваемости школьников. В данной задаче ученик характеризуется фамилией и оценками, полученными на всех занятиях по всем предметам. Перенумеруем занятия числами от 1 до ЧИСЛО-ЗАНЯТИЙ. Для хранения фамилии используем массив из 20 литер, более короткие фамилии дополним справа пробелами. В итоге получим такое описание ученика и всего журнала:

```
TYPE ФАМИЛИЯ=PACKED ARRAY [1..20] OF CHAR;  
    ПРЕДМЕТЫ=(ПРОГРАММИРОВАНИЕ, МАТЕМАТИКА,  
    ФИЗИКА, . . .);  
    ОЦЕНКИ=ARRAY[ПРЕДМЕТЫ, 1..ЧИСЛО-ЗАНЯТИЙ]  
        OF 0..5;  
    (* ОЦЕНКА 0 ОЗНАЧАЕТ ОТСУТСТВИЕ ОЦЕНКИ *)  
    УЧЕНИК=RECORD  
        ФАМ:ФАМИЛИЯ; ОЦ:ОЦЕНКИ  
    END;  
    ЖУРНАЛ=ARRAY[1..ЧИСЛО_УЧЕНИКОВ] OF УЧЕНИК;
```

Учитель должен иметь возможность поставить оценку ученику. Для этого напишем процедуру ПОСТАВИТЬ-ОЦЕНКУ:

```
PROCEDURE ПОСТАВИТЬ-ОЦЕНКУ(VAR ЖУР:ЖУРНАЛ;  
    ПРЕД:ПРЕДМЕТЫ; ЗАНЯТИЕ:1..ЧИСЛО-ЗАНЯТИЙ; ФАМ:  
    ФАМИЛИЯ, ОЦ:1..5);
```

```

(* ПОСТАВИТЬ ОЦЕНКУ ОЦ УЧЕНИКУ С ФАМИЛИЕЙ ФАМ
   ЗА ДАННОЕ ЗАНЯТИЕ *)
VAR I:0..ЧИСЛО-УЧЕНИКОВ; J:INTEGER;
BEGIN
  (* ПОИСК УЧЕНИКА В ЖУРНАЛЕ *)
  I:=0;
  REPEAT
    I:=I+1
  UNTIL (ЖУР[I].ФАМ=ФАМ) OR (I=ЧИСЛО-УЧЕНИКОВ);
  IF ЖУР[I].ФАМ=ФАМ THEN
    ЖУР[I].ОЦ[ПРЕД,ЗАНЯТИЕ]:=ОЦ
  ELSE (* НЕ НАШЛИ УЧЕНИКА *) BEGIN
    WRITE(' УЧЕНИКА ПО ФАМИЛИИ ');
    FOR J:=1 TO 20 DO WRITE(ФАМ[J]);
    WRITELN(' НЕТ В ЖУРНАЛЕ')
  END
END;

```

В качестве еще одного примера рассмотрим бегло организацию информации для городской справочной службы. Информация о человеке в этом случае включает фамилию, имя, отчество, дату рождения, адрес, телефон:

```

TYPE ИМЕНА=PACKED ARRAY[1..20] OF CHAR;
   ДАТА=RECORD ГОД, МЕСЯЦ, ЧИСЛО:INTEGER END;
   ТЕКСТ=PACKED ARRAY[1..100] OF CHAR;
   НОМЕР=ARRAY[1..7] OF 0..9;
   ЧЕЛОВЕК=RECORD
     ФАМИЛИЯ, ИМЯ, ОТЧЕСТВО:ИМЕНА;
     ДАТА-РОЖДЕНИЯ:ДАТА;
     АДРЕС:ТЕКСТ; ТЕЛЕФОН: НОМЕР
   END;

```

В процедуре поиска адреса человека в справочнике, представляющем собой массив из записей типа ЧЕЛОВЕК, мы могли бы выполнять такое сравнение:

```

IF (СПРАВОЧНИК[I].ФАМИЛИЯ=ДАННАЯ-ФАМИЛИЯ) AND
   (СПРАВОЧНИК[I].ИМЯ=ДАННОЕ-ИМЯ) AND
   (СПРАВОЧНИК[I].ОТЧЕСТВО=ДАННОЕ-ОТЧЕСТВО) AND
   (СПРАВОЧНИК[I].ДАТА-РОЖДЕНИЯ.ГОД=
    ДАННЫЙ-ГОД) AND
   (СПРАВОЧНИК[I].ДАТА-РОЖДЕНИЯ.МЕСЯЦ=
    ДАННЫЙ-МЕСЯЦ) AND
   (СПРАВОЧНИК[I].ДАТА-РОЖДЕНИЯ.ЧИСЛО=
    ДАННОЕ-ЧИСЛО)
THEN ПЕЧАТЬ(СПРАВОЧНИК[I].АДРЕС);

```

Можно сократить программу, воспользовавшись инструкцией WITH. Она позволяет один раз указать, с какой записью будет вестись работа, и затем указывать в программе только имена компонент записи. Общий вид инструкции WITH:

```
WITH <запись1>, ... , <записьk> DO BEGIN
  <инструкция1>;
  ...
  <инструкцияn>;
END
```

В инструкциях между BEGIN и END для указания компонент записей, перечисленных после WITH, можно вместо полной формы <запись>. <имя компоненты> использовать сокращенную — <имя компоненты>. При этом изнутри инструкции WITH нельзя обратиться к переменным, имена которых совпадают с именами компонент перечисленных записей. Приведенный выше фрагмент с использованием WITH переписывается так (по общему правилу здесь можно опустить BEGIN и END):

```
WITH СПРАВОЧНИК[I] DO WITH ДАТА-РОЖДЕНИЯ DO
IF (ФАМИЛИЯ=ДААННАЯ-ФАМИЛИЯ) AND
  (ИМЯ=ДААННОЕ-ИМЯ) AND
  (ОТЧЕСТВО=ДААННОЕ-ОТЧЕСТВО) AND
  (ГОД=ДААННЫЙ-ГОД) AND
  (МЕСЯЦ=ДААННЫЙ-МЕСЯЦ) AND
  (ЧИСЛО=ДААННОЕ-ЧИСЛО)
THEN ПЕЧАТАТЬ(АДРЕС);
```

Рассмотрим еще один пример. Пусть мы разрабатываем некоторую геометрическую систему и хотим работать в ней с плоскими фигурами. Ограничимся следующими тремя видами фигур: прямоугольником, кругом, правильным многоугольником. Каждая фигура может быть охарактеризована несколькими числовыми параметрами, поэтому для представления фигур в программе естественно использовать записи.

Какие же компоненты должны быть у записи типа ФИГУРА? Во-первых, конечно, вид фигуры:

```
TYPE ВИД-ФИГУРЫ=(ПРЯМОУГОЛЬНИК, КРУГ,
  ПРАВ-МНОГУГОЛЬНИК);
ФИГУРА=RECORD ВИД: ВИД-ФИГУРЫ;
```

...

Далее, для круга нужно хранить его радиус, для прямоугольника — размеры сторон, для правильного многоугольника можно хранить, например, радиус описанной окружности и число сторон. Получается, что необходимый нам набор компонент различен для разных экземпляров записей одного типа. Работать с такими

объектами позволяет имеющийся в паскале механизм *записей с вариантами*. Он дает возможность объединять в одном описании типа записи несколько вариантов описания. Какой из вариантов используется в каждом конкретном экземпляре записи, определяется значением одной выделенной компоненты, называемой *компонентой-дискриминантом*. В нашем случае — компонентой ВИД.

В описании записи сначала описываются компоненты, общие для всех вариантов. Наличие вариантов указывается конструкцией

CASE <описание компоненты-дискриминанта> OF

вслед за которой описываются все варианты, разделяемые точками с запятой. Описание одного варианта начинается с одной или нескольких констант, указывающих значения дискриминанта для данного варианта. После констант записывается двоеточие и затем в круглых скобках — описание компонент, относящихся к данному варианту. Таким образом, общий вид описания записи следующий:

RECORD

<имя₁>, . . . , <имя_р>: <тип₁>;

...

<имя_q>, . . . , <имя_n>: <тип_n>;

CASE <имя дискриминанта>: <тип дискриминанта> OF

<константа₁>, . . . , <константа_r>:

((список компонент варианта 1));

...

<константа_s>, . . . , <константа_m>:

((список компонент варианта L))

END

Здесь все константы должны принадлежать типу дискриминанта.

В описании записи может отсутствовать либо общая часть (между словами RECORD и CASE), либо вариантная часть (от слова CASE до, но не включая, END). Можно также опустить <имя дискриминанта> с последующим двоеточием; в этом случае в записи не будет дискриминанта. Такая возможность используется, если варианты записи различаются каким-либо другим способом, например на основе информации, хранящейся вне самой записи.

Описания компонент вариантов составляются по тем же правилам, что и описание всей записи между словами RECORD и END; в частности, они сами могут содержать вариантную часть.

Вернемся к примеру с геометрическими фигурами. Если мы хотим запоминать не только форму, но и положение фигуры на плоскости, то можно хранить координаты центра фигуры и наклон — величину угла, на который следует повернуть фигуру из ее начального стандартного положения. Эта информация — общая для всех фигур, поэтому используем для ее хранения компоненты в общей

части записи. Кроме того, полезно вывести в общую часть компонента РАЗМЕР, содержащую некоторую характерную длину фигуры. Это позволит, например, выполнять гомететию любой фигуры, не выясняя ее вид, а просто изменив значение компоненты РАЗМЕР. В компоненту РАЗМЕР поместим длину одной из сторон прямоугольника, или радиус круга, или радиус описанной окружности. Вариантная часть записи будет описывать собственно форму фигуры. Для прямоугольника форма задается отношением второй стороны прямоугольника к стороне, принятой за РАЗМЕР. Итак, приходим к следующему описанию:

TYPE

ВИД-ФИГУРЫ=(ПРЯМОУГОЛЬНИК,КРУГ,
ПРАВ-МНОГОУГОЛЬНИК);

ФИГУРА=RECORD

X,Y:REAL (* КООРДИНАТЫ ЦЕНТРА *);

НАКЛОН.РАЗМЕР:REAL;

CASE ВИД:ВИД-ФИГУРЫ OF

ПРЯМОУГОЛЬНИК: (ОТНОШЕНИЕ-СТОРОН: REAL);

КРУГ: ();

(* ВАРИАНТНАЯ ЧАСТЬ ДЛЯ КРУГА ПУСТА *)

ПРАВ-МНОГОУГОЛЬНИК: (ЧИСЛО-СТОРОН: INTEGER)

END;

Опишем переменную типа фигура и присвоим ей в качестве значения правильный пятиугольник, вписанный в окружность единичного радиуса с центром в начале координат. Следует иметь в виду, что при обработке записей с вариантами можно обращаться к компонентам только того варианта записи, который отвечает значению дискриминанта. Поэтому при присваивании начального значения записи следует сначала присвоить значение компоненте-дискриминанту, и лишь затем — компонентам вариантной части:

VAR ПЯТИУГОЛЬНИК:ФИГУРА;

...

WITH ПЯТИУГОЛЬНИК DO BEGIN

X:=0; Y:=0; НАКЛОН:=0; РАЗМЕР:=1;

ВИД:=ПРАВ-МНОГОУГОЛЬНИК;

ЧИСЛО-СТОРОН:=5;

END;

Напишем процедуру для выполнения параллельного переноса фигур и функцию для вычисления площади фигур. Процедура параллельного переноса работает единым образом для всех фигур:

PROCEDURE ПАРАЛЛЕЛЬНЫЙ-ПЕРЕНОС

(VAR Ф:ФИГУРА; DX,DY:REAL);

(* ПЕРЕНЕСТИ ФИГУРУ Ф НА (DX,DY) *)

```
BEGIN WITH Ф DO BEGIN
```

```
  X:=X+DX;
```

```
  Y:=Y+DY;
```

```
END END;
```

Иначе обстоит дело в функции ПЛОЩАДЬ. Формулы площади для фигур разных видов различны, поэтому придется осуществить разветвление по виду фигуры. Наиболее естественным механизмом в данном случае, как и при всякой работе с вариантной частью записи, является инструкция выбора CASE, в которой для разветвления используется значение дискриминанта:

```
FUNCTION ПЛОЩАДЬ(Ф:ФИГУРА):REAL;
```

```
(* ПЛОЩАДЬ ФИГУРЫ Ф *)
```

```
CONST PI=3.14159265;
```

```
VAR S:REAL;
```

```
BEGIN WITH Ф DO BEGIN
```

```
  (* ВЫЧИСЛИМ СНАЧАЛА В ПЕРЕМЕННОЙ S ПЛОЩАДЬ  
    ФИГУРЫ ЕДИНИЧНОГО РАЗМЕРА *)
```

```
  CASE ВИД OF
```

```
    ПРЯМОУГОЛЬНИК: S:=ОТНОШЕНИЕ-СТОРОН;
```

```
    КРУГ:S:=PI;
```

```
    ПРАВ-МНОГОУГОЛЬНИК:
```

```
      S:=ЧИСЛО-СТОРОН/2*SIN(2*PI/ЧИСЛО-СТОРОН)
```

```
  END (* CASE *);
```

```
  ПЛОЩАДЬ:=S*SQR(РАЗМЕР);
```

```
END END;
```

Полезно представлять себе способ размещения в памяти ЭВМ компонент записи. В большинстве реализаций паскаль-машины все варианты записи накладываются в памяти друг на друга, поскольку они никогда не бывают нужны одновременно. Тем самым под запись с вариантами требуется объем памяти, необходимый для хранения самого большого варианта, а не всех вариантов сразу. Здесь

X	Y	НАКЛОН	РАЗМЕР	ВИД	ОТНОШЕНИЕ_СТОРОН
					ЧИСЛО_СТОРОН

показано отображение в память записи ФИГУРА. Память ЭВМ на этом рисунке простирается по горизонтали, расположенные одна над другой компоненты ОТНОШЕНИЕ-СТОРОН и ЧИСЛО-СТОРОН символизируют то, что эти компоненты расположены в одном месте памяти.

4.7. Точность машинных вычислений

В этом разделе мы познакомимся со способом представления вещественных чисел в ЭВМ и с тем, как он влияет на программирование. Принципиальная трудность работы с вещественными числами является следствием того факта, что любая переменная в памяти ЭВМ может принимать лишь конечное число значений, тогда как даже на конечном отрезке содержится бесконечно много вещественных чисел. Поэтому вещественные числа не могут быть представлены в машине точно.

Практически во всех современных ЭВМ вещественные числа записываются в так называемом формате с плавающей точкой. При этом отдельно хранится *мантисса* M , представляющая собой число, по модулю не превосходящее 1, и *порядок* p , показывающий, на какую степень основания принятой системы счисления a , надо умножить мантиссу. Таким образом, пара (p, M) задает вещественное число x , вычисляемое по формуле

$$x = M \cdot a^p.$$

Такое представление чисел широко используется, например, в физике. Скажем заряд электрона в кулонах обычно записывается в виде $1.6 \cdot 10^{-19}$.

В ЭВМ основание a обычно равно 2 или 16. Для записи мантиссы используется фиксированное число цифр k , т. е.

$$M = MM/a^k,$$

где MM — целое число, $|MM| \leq a^k$. Диапазон изменения порядка тоже ограничен:

$$|p| \leq p_{\max}.$$

Таким образом, представление чисел в ЭВМ обладает следующими свойствами:

- 1) в ЭВМ невозможно представить очень большие по модулю и очень малые (близкие к нулю) ненулевые числа;
- 2) произвольное вещественное число, попадающее в допустимый интервал, может быть записано с некоторой погрешностью; относительная погрешность примерно постоянна и равна a^{-k} , будем обозначать эту величину символом δ .

Мы будем использовать две характеристики точности — абсолютную погрешность, равную абсолютной величине разности точного числа и приближенного, и относительную погрешность, равную отношению абсолютной погрешности к точному значению.

ЭВМ БЭСМ-6 имеет следующие характеристики точности: диапазон представимых чисел — примерно от $0.3 \cdot 10^{-19}$ до 10^{19} , относительная погрешность δ приближенно равна 10^{-12} , т. е. ман-

тисса имеет 12 значащих десятичных цифр. С такой же погрешностью (в наихудшем случае) выполняются все операции над вещественными числами. В ЕС ЭВМ диапазон чисел шире: от 10^{-78} до $4 \cdot 10^{75}$, а точность составляет примерно 16—17 значащих десятичных цифр.

Столь малая погрешность, казалось бы, достаточная для всех практических расчетов, может иногда катастрофически возрасти, и результаты машинного расчета могут получиться совсем неправильными. Рассмотрим ряд таких ситуаций:

```
TYPE ТОЧКА=RECORD X,Y:REAL END;
```

```
(* X,Y — КООРДИНАТЫ ТОЧКИ НА ПЛОСКОСТИ *)
```

```
FUNCTION DIST(P:ТОЧКА):REAL;
```

```
(* РАССТОЯНИЕ ТОЧКИ P ОТ НАЧАЛА КООРДИНАТ *)  
BEGIN WITH P DO DIST:=SQRT(X*X+Y*Y) END;
```

У этой функции, конечно, есть пределы применимости. Естественные границы определяются условием, что исходные данные и результат должны попадать в диапазон представимых чисел. Эта функция, однако, при работе на БЭСМ-6 не сможет получить результат, когда X или Y будут порядка $4 \cdot 10^9$ (или больше). При этом исходные координаты и искомое расстояние еще весьма далеки от границы, однако, промежуточные величины — квадраты координат и их сумма — превысят 10^{19} и окажутся непредставимыми в ЭВМ, что приведет к аварийному окончанию программы. Если же обе координаты будут меньше 10^{-10} , то при возведении их в квадрат должно получиться число, меньшее, чем $0.3 \cdot 10^{-19}$. Однако ЭВМ не может работать со столь малыми числами, поэтому в качестве результата получится 0 и относительная погрешность достигнет 1!

Избавиться от очень больших промежуточных величин можно, например, вынеся за знак квадратного корня наибольшую по модулю координату:

```
FUNCTION DIST2(P:ТОЧКА):REAL;
```

```
(* РАССТОЯНИЕ ТОЧКИ P ОТ НАЧАЛА КООРДИНАТ *)
```

```
BEGIN WITH P DO
```

```
IF (X=0.0) AND (Y=0.0) THEN DIST2:=0.0 ELSE
```

```
IF ABS(X)>ABS(Y)
```

```
THEN DIST2:=ABS(X)*SQRT(1+SQR(Y/X))
```

```
ELSE DIST2:=ABS(Y)*SQRT(1+SQR(X/Y))
```

```
END;
```

Эта более сложная функция вычисляет расстояние с относительной погрешностью порядка 10^{-12} (на БЭСМ-6) для всех точек для которых X , Y и результат могут быть записаны в машинное слово.

Дефекты исходной функции DIST были вызваны первым свойством машинного представления чисел — ограниченностью их диапазона. Это свойство, в отличие от второго, имеет место и для целых

чисел — диапазон целых чисел, с которым может работать ЭВМ, ограничен (на БЭСМ-6 нижняя и верхняя границы диапазона целых чисел составляют примерно -10^{12} и 10^{12} , на ЕС ЭВМ соответственно $-2 \cdot 10^9$ и $2 \cdot 10^9$). Следовательно, в программах для целочисленных расчетов также следует избегать больших промежуточных результатов. Поэтому, например, следует признать неудачной программу вычисления числа сочетаний из N элементов по M непосредственно по формуле $N!/(M!(N-M)!)$. Входящие в эту формулу факториалы значительно больше требуемого числа сочетаний, и это вызовет неправильную работу программы задолго до того, как число сочетаний выйдет за допустимые пределы.

В следующих примерах рассмотрим следствия второго свойства — ограниченной точности записи вещественных чисел в ЭВМ и действий с ними.

```
TYPE ТОЧКА=RECORD X,Y:REAL END;
```

```
(* X,Y — КООРДИНАТЫ ТОЧКИ НА ПЛОСКОСТИ *)
```

```
FUNCTION ПЛОЩ_ТРАП(A,B:ТОЧКА):REAL;
```

```
(* ПЛОЩАДЬ ТРАПЕЦИИ, ЗАКЛЮЧЕННОЙ МЕЖДУ  
ОТРЕЗКОМ АВ И ОСЬЮ АБСЦИСС.
```

```
ОСНОВАНИЯ ТРАПЕЦИИ — ПЕРПЕНДИКУЛЯРЫ,  
ОПУЩЕННЫЕ ИЗ ТОЧЕК А И В НА ОСЬ АБСЦИСС *)
```

```
BEGIN ПЛОЩ_ТРАП:=(A.Y+B.Y)/2*(B.X-A.X) END;
```

```
FUNCTION ПЛОЩ_ТРЕУГ(A,B,C:ТОЧКА):REAL;
```

```
(* ПЛОЩАДЬ ТРЕУГОЛЬНИКА ABC *)
```

```
BEGIN ПЛОЩ_ТРЕУГ:=ABS(ПЛОЩ_ТРАП(A,B)+
```

```
ПЛОЩ_ТРАП(B,C) -ПЛОЩ_ТРАП(A,C))
```

```
END;
```

В этих функциях может происходить вычитание близких чисел — действие, приводящее к потере точности (наряду со сложением почти противоположных чисел). Рассмотрим сначала функцию ПЛОЩ_ТРАП. Пусть точки A и B лежат в первом квадранте близко друг к другу, тогда $A.X$ и $B.X$ — положительные числа, близкие друг к другу и к некоторому числу X_0 . Относительная погрешность задания $A.X$ и $B.X$ примерно δ , а абсолютная — δX_0 . Абсолютная погрешность их разности может в наихудшем случае равняться сумме абсолютных погрешностей операндов, т. е. $2\delta X_0$ (плюс погрешность выполнения операции вычитания, которой в данном случае пренебрежем). Относительная погрешность составит $2\delta X_0 / \text{ABS}(B.X - A.X)$, т. е. в $2X_0 / \text{ABS}(B.X - A.X)$ раз больше теоретически минимальной относительной погрешности δ . Если точки близки (в сравнении с расстоянием до оси ординат), то отношение $X_0 / \text{ABS}(B.X - A.X)$ велико. Последующие действия не изменят существенно эту погрешность, они добавят к ней погрешность порядка δ . Таким образом, результат функции ПЛОЩ_ТРАП может иметь

погрешность, существенно превосходящую δ . Однако эту программу нельзя улучшить. Дело в том, что если бы даже вычисления выполнялись абсолютно точно, а погрешности были бы вызваны только неточным заданием исходных данных (а эту погрешность нельзя устранить, перенесшая программу), то погрешность результата все равно была бы того же порядка, что получается в функции ПЛОЩ-ТРАП.

Иначе обстоит дело в функции ПЛОЩ-ТРЕУГ. Рассмотрим случай маленького треугольника, все вершины которого лежат вблизи точки (X_0, Y_0) первого квадранта, и вычислим, какая погрешность в площади возникает из-за неточности записи исходных данных — координат вершин (это естественная мера погрешности, с которой следует сравнить погрешность, даваемую функцией ПЛОЩ-ТРЕУГ). Все исходные координаты заданы с относительной погрешностью δ , которой соответствует абсолютная погрешность δX_0 для X -координат и δY_0 для Y -координат. Если изменить X -координату какой-либо вершины на δX_0 , то высота треугольника, опущенная из этой вершины, изменится на меньшую или равную величину, а площадь изменится не больше, чем на $\delta X_0 L$, где L — длина противоположной стороны. Складывая погрешности, вызванные изменением всех X - и Y -координат, получим следующую верхнюю оценку погрешности площади:

$$3\delta(X_0 + Y_0) \cdot L_{max},$$

где L_{max} — длина максимальной стороны треугольника.

Посмотрим, достигается ли такая точность в функции ПЛОЩ-ТРЕУГ. Используемая в ней функция ПЛОЩ-ТРАП(A, B) дает результат с относительной погрешностью $2\delta X_0 / \text{ABS}(B \cdot X - A \cdot X)$. Умножив это число на величину площади — примерно $Y_0(B \cdot X - A \cdot X)$, найдем абсолютную погрешность — $2\delta X_0 Y_0$. Такую же погрешность дадут два других обращения к ПЛОЩ-ТРАП, что приведет к погрешности суммарной площади $6\delta X_0 Y_0$. Отношение этой оценки к полученной ранее теоретической границе равно $2X_0 Y_0 / ((X_0 + Y_0) \cdot L_{max})$. Это отношение много больше единицы в случае маленького треугольника (в сравнении с расстоянием до осей координат), т. е. когда X_0, Y_0 много больше L_{max} .

Источник роста погрешности — то, что маленькая площадь треугольника получается как разность близких больших площадей трапеций. Чтобы избавиться от этого, придется подставить вместо обращений к ПЛОЩ-ТРАП формулы для площадей трапеций и сократить одинаковые члены, входящие с разным знаком. Выполнить некоторые преобразования, придем к следующей программе:

```
FUNCTION ПЛОЩТР2(A,B,C:ТОЧКА):REAL;
  (* ПЛОЩАДЬ ТРЕУГОЛЬНИКА ABC *)
```

```
BEGIN ПЛОЩП2:=ABS(0.5*((B.Y-A.Y)*(C.X-A.X)
      -(B.X-A.X)*(C.Y-A.Y)))
END;
```

Эта функция обеспечивает погрешность, отличающуюся от теоретической границы только постоянным множителем.

В следующем примере основной вклад в погрешность результата вносит погрешность выполнения операций, которая была несущественна в предыдущих программах:

```
FUNCTION HARM(N:INTEGER):REAL;
  (* СУММА ПЕРВЫХ N ЧЛЕНОВ ГАРМОНИЧЕСКОГО
  РЯДА *)
VAR S:REAL; K:INTEGER;
BEGIN S:=0;
  FOR K:=1 TO N DO S:=S+1/K;
  HARM:=S;
END;
```

Погрешность результата складывается из погрешностей слагаемых и погрешности выполнения всех сложений, которые на каждом шаге равны δS . В этой программе мы складываем сначала большие слагаемые, поэтому частичная сумма S быстро достигает больших значений, соответственно на каждом шаге в результат вносится большая погрешность.

Рассмотрим крайний, искусственный пример подобной ситуации. Пусть мы выполняем вычисления всего с четырьмя значащими цифрами и хотим вычислить выражение $55.55 + 0.001 + 0.001 + \dots$ (сто слагаемых по 0.001). Если вычислять сумму в том порядке, как написано, то сначала надо вычислить $55.55 + 0.001 = 55.551$. Это число нужно округлить до четырех цифр (мы не можем хранить пятую цифру), что даст 55.55. Понятно, что, сколько ни прибавляй 0.001, число 55.55 таким и останется. Однако можно сначала сложить все 0.001 (это даст 0.1) и затем прибавить сумму к 55.55. Такой порядок суммирования обеспечит малость частичных сумм и, как следствие, меньшую суммарную погрешность (в данном случае получается точный результат).

При сложении положительных чисел наилучший с точки зрения точности порядок суммирования — по возрастанию слагаемых. В рассматриваемой функции HARM для этого следует цикл суммирования записать наоборот:

```
FOR K:=N DOWNT0 1 DO S:=S+1/K;
```

Задача. Докажите, что функция

```
FUNCTION КОРЕНЬ-КВАДРАТНЫЙ(A: REAL): REAL;
  (* A>0 — ЗНАЧЕНИЕ, ИЗ КОТОРОГО ИЗВЛЕКАЕТСЯ
```

```

КОРЕНЬ *)
VAR ТЕК,ПРЕД: REAL;
(* ТЕК — ТЕКУЩЕЕ ПРИБЛИЖЕНИЕ К КОРНЮ
   ПРЕД — ПРЕДЫДУЩЕЕ ПРИБЛИЖЕНИЕ *)
BEGIN
  ТЕК:=А+1;
  REPEAT
    ПРЕД:=ТЕК;
    ТЕК:=0.5*(ПРЕД+А/ПРЕД);
  UNTIL ТЕК>=ПРЕД;
  КОРЕНЬ КВАДРАТНЫЙ:=ПРЕД;
END;

```

на любой ЭВМ вычисляет корень квадратный с относительной погрешностью порядка δ . Тем самым мы получаем машинно-независимый способ максимально точного извлечения корня.

У к а з а н и е. Воспользуйтесь конечностью множества представимых в ЭВМ вещественных чисел, а также тем, что для любого $X > A$

$$0.5 \cdot (X + A/X) < X$$

при условии точного выполнения арифметических операций.

4.8. Динамические переменные и указатели

Между объектами реального мира, поведение которых моделируют программы, существуют разнообразные, постоянно меняющиеся связи. Одни объекты могут исчезнуть, другие — появиться. Так, люди рождаются и умирают, порывают с кем-то из старых друзей и заводят новых. Ясно, что, если в программе мы хотим моделировать группы с переменным числом объектов, связи между которыми подвержены изменению, нужны языковые средства для установления, изменения и разрыва связей между модельными объектами, а также для порождения и уничтожения объектов. Для этой цели в паскале служат динамические переменные и указатели.

Прежде чем переходить собственно к изложению нового материала, попытаемся проиллюстрировать недостаточность ранее изученных языковых средств. Рассмотрим задачу о считалке. Пусть N людей встают в круг и получают номера, считая по часовой стрелке, 1, 2, ..., N . Затем, начиная с первого, также по часовой стрелке отсчитывается M -й человек. (Поскольку люди стоят по кругу, то при счете за N -м следует первый.) Этот M -й выходит из круга, после чего, начиная со следующего, снова отсчитывается M -й человек, и так до тех пор, пока из всего круга не останется один человек. Требуется определить его номер.

Для решения задачи промоделируем ход считалки:

```
FUNCTION СЧИТАЛКА1(N,M: INTEGER): INTEGER;  
(* ФУНКЦИЯ ОПРЕДЕЛЯЕТ, КТО ИЗ N ЛЮДЕЙ  
ОСТАНЕТСЯ ПОСЛЕ ЗАВЕРШЕНИЯ СЧИТАЛКИ, ЕСЛИ  
ВЫБЫВАЕТ КАЖДЫЙ М-Й *)  
CONST ПРЕДЕЛ=100;  
(* СЧИТАЕМ, ЧТО ЛЮДЕЙ БУДЕТ НЕ БОЛЬШЕ 100 *)  
VAR КАНДИДАТЫ: ARRAY[1..ПРЕДЕЛ] OF BOOLEAN;  
(* ЗНАЧЕНИЕМ FALSE ПОМЕЧАЕМ ВЫБЫВШИХ *)  
J,K,L: INTEGER;  
BEGIN  
  FOR L:=1 TO N DO КАНДИДАТЫ[L]:=TRUE;  
  (* ПОКА НЕ ВЫБЫЛ НИКТО *)  
  L:=N;  
  (* КАЖДЫЙ РАЗ ОТСЧЕТ ЛЮДЕЙ БУДЕМ НАЧИНАТЬ  
  С ЧЕЛОВЕКА СЛЕДУЮЩЕГО ЗА L-M.  
  СНАЧАЛА ЭТО ДОЛЖЕН БЫТЬ НОМЕР 1 *)  
  FOR K:=N DOWNT0 1 DO BEGIN  
    (* N РАЗ ПОВТОРИМ СЧИТАЛКУ *)  
    FOR J:=1 TO M DO  
      (* ЦИКЛ ОТСЧЕТА М-ГО ЧЕЛОВЕКА *)  
      REPEAT (* ПРОПУСКАЕМ ВЫШЕДШИХ ИЗ КРУГА *)  
        L:=(L MOD N)+1  
      UNTIL КАНДИДАТЫ[L];  
      (* L-Й ЧЕЛОВЕК ВЫХОДИТ ИЗ КРУГА *)  
      КАНДИДАТЫ[L]:=FALSE;  
    END;  
    (* НОМЕР ВЫШЕДШЕГО ИЗ КРУГА ПОСЛЕДНИМ —  
    ИСКОМЫЙ *)  
    СЧИТАЛКА1:=L  
  END;  
END;
```

Основной недостаток функции СЧИТАЛКА1 в том, что мы только помечаем отвергнутых кандидатов, но не выводим их из круга. Поэтому в процессе счета мы вынуждены раз за разом идти сквозь строй отвергнутых кандидатов.

Чтобы иметь возможность устанавливать и разрывать связи между объектами в процессе выполнения программы, нужен новый вид значений — адреса, по которым можно обращаться к объектам. Переменные, значениями которых являются адреса, называются указателями. Указатели, как и другие объекты языка паскаль, различаются по типам. Указательный тип определяется так:

TYPE <тип указателей>=↑<тип указуемых объектов>;

Переменные-указатели описываются обычным образом. Определим, например, тип КАНДИДАТ, характеризующий человека, участвующего в считалке, и опишем указатели на объекты этого типа:

TYPE АДРЕС-КАНДИДАТА = ↑КАНДИДАТ;

КАНДИДАТ = RECORD

НОМЕР: INTEGER;

СЛЕДУЮЩИЙ: АДРЕС-КАНДИДАТА

END;

VAR ПЕРВЫЙ, ТЕКУЩИЙ, НОВЫЙ: АДРЕС-КАНДИДАТА;

В записи хранятся номер кандидата и адрес человека, стоящего в круге следующим.

До сих пор мы могли обращаться к переменным только по именам. Назовем такие переменные статическими. Переменные, обращение к которым производится по адресам, будем называть динамическими. Для обозначения динамической переменной служит конструкция

(указатель)↑

Пр и м е р.

ПЕРВЫЙ↑.НОМЕР:=1;

ПЕРВЫЙ↑.СЛЕДУЮЩИЙ↑.НОМЕР:=2

Подчеркнем, что ПЕРВЫЙ — это обычная статическая переменная, значением которой является адрес. Адрес хранится и в компоненте ПЕРВЫЙ↑.СЛЕДУЮЩИЙ. В то же время ПЕРВЫЙ↑ и ПЕРВЫЙ↑.СЛЕДУЮЩИЙ↑ — это динамические переменные (записи с двумя компонентами).

Чтобы создать динамическую переменную, нужно сообщить паскаль-машине тип переменной и в ответ получить адрес созданной переменной. Этой цели служит стандартная процедура

NEW ((указатель))

По типу указателя паскаль-машина определит тип динамической переменной, создаст ее и присвоит указателю ее адрес. Это — единственный способ получения новых адресных значений. Адреса можно также присваивать и сравнивать на равенство.

Поясним сказанное аналогией. Уподобим переменную квартиру. Пусть мы хотим получить от паскаль-машины новую квартиру. Обращаясь к процедуре NEW, мы как бы предлагаем паскаль-машине бланк, по виду которого паскаль-машина определяет, какая квартира нужна. Затем квартира выделяется (разумеется, при наличии ресурсов) и паскаль-машина записывает на нашем бланке адрес. Квартира выделена, но она пуста, вернее, завалена мусором. Отправившись по имеющемуся адресу, мы сможем должным образом

обставить квартиру (снабдить динамическую переменную значением). Квартира доступна, если хоть где-то в доступном месте (например, в статической переменной) записан ее адрес. Адреса можно хранить и в компонентах динамических переменных (записывать их на стенах ранее выделенных квартир). Это позволяет иметь доступ к сколь угодно большому числу динамических переменных (в рамках ресурсов паскаль-машины).

Если какая-то динамическая переменная больше не нужна, можно вернуть ее паскаль-машине, обратившись к процедуре

DISPOSE (указатель)

Нужно помнить, однако, что, если где-то остался адрес возвращенной переменной (квартиры), возможен конфликт между старыми и новыми квартиросъемщиками (после того как паскаль-машина вновь пустит освобожденную квартиру в оборот). Иными словами, пользоваться процедурой DISPOSE нужно очень осмотрительно.

Промоделируем теперь ход считалки, используя указатели:

```
FUNCTION СЧИТАЛКА (N,M: INTEGER): INTEGER;  
TYPE АДРЕС-КАНДИДАТА = ↑КАНДИДАТ;  
  КАНДИДАТ = RECORD  
    НОМЕР: INTEGER;  
    СЛЕДУЮЩИЙ: АДРЕС-КАНДИДАТА  
  END;  
VAR ПЕРВЫЙ, ТЕКУЩИЙ, НОВЫЙ: АДРЕС-КАНДИДАТА;  
  J, K: INTEGER;  
BEGIN  
  (* СОЗДАДИМ ЗАПИСЬ ДЛЯ ПЕРВОГО КАНДИДАТА *)  
  NEW(ПЕРВЫЙ); ПЕРВЫЙ↑.НОМЕР := 1;  
  ТЕКУЩИЙ := ПЕРВЫЙ;  
  (* ВЫСТРОИМ В ЦЕПОЧКУ ОСТАЛЬНЫХ  
    КАНДИДАТОВ *)  
  FOR K := 2 TO N DO BEGIN  
    NEW(НОВЫЙ);  
    НОВЫЙ↑.НОМЕР := K;  
    ТЕКУЩИЙ↑.СЛЕДУЮЩИЙ := НОВЫЙ;  
    ТЕКУЩИЙ := НОВЫЙ  
  END;  
  (* ПОСЛЕ ОКОНЧАНИЯ ЦИКЛА УКАЗАТЕЛЬ  
    ТЕКУЩИЙ  
    ХРАНИТ АДРЕС N-ГО КАНДИДАТА. ЗАМКНЕМ  
    КРУГ *)  
  ТЕКУЩИЙ↑.СЛЕДУЮЩИЙ := ПЕРВЫЙ;  
  (* ПОВТОРИМ СЧИТАЛКУ N-1 РАЗ *)  
  FOR K := N DOWNTO 2 DO BEGIN
```



```

(* ОТСЧИТАЕМ М-ГО ЧЕЛОВЕКА *)
FOR J:=1 TO M-1 DO ТЕКУЩИЙ:=
ТЕКУЩИЙ↑.СЛЕДУЮЩИЙ;
(* УДАЛИМ ЭЛЕМЕНТ, СЛЕДУЮЩИЙ ЗА ТЕКУЩИМ *)
ТЕКУЩИЙ↑.СЛЕДУЮЩИЙ:=
ТЕКУЩИЙ↑.СЛЕДУЮЩИЙ↑.СЛЕДУЮЩИЙ
END;
(* В КРУГЕ ОСТАЛСЯ ОДИН ЧЕЛОВЕК.
ЕГО НОМЕР — ИСКОМЫЙ *)
СЧИТАЛКА:=ТЕКУЩИЙ↑.НОМЕР
END

```

В этом примере записи, характеризующие кандидатов, были связаны в кольцо — компонента СЛЕДУЮЩИЙ записи с номером N хранила адрес первой записи. Такая структура называется кольцевым списком. Часто используется и чуть иная структура, называемая линейным списком или просто списком, в которой компонента связи последней записи содержит специальное значение — константу NIL. Это значение показывает, что список кончился и дальше элементов нет. Значение NIL — это адрес, показывающий в никуда; ни одна динамическая переменная этим значением адресоваться не может. Значение NIL можно присвоить указателю любого типа.

Следующая серия примеров призвана проиллюстрировать технику работы со списками. Предполагается, что всем написанным далее процедурам предшествуют определения типов:

```

TYPE АДРЕС-ЭЛЕМЕНТА=↑ЭЛЕМЕНТ-СПИСКА;
ЭЛЕМЕНТ-СПИСКА=RECORD
    ЗНАЧЕНИЕ: REAL;
    СВЯЗЬ: АДРЕС-ЭЛЕМЕНТА
END;

```

Напишем функцию, вычисляющую сумму компонент ЗНАЧЕНИЕ элементов списка:

```

FUNCTION СУММА-ЗНАЧЕНИЙ
(ПЕРВЫЙ: АДРЕС-ЭЛЕМЕНТА): REAL;
(* ПЕРВЫЙ СОДЕРЖИТ АДРЕС ПЕРВОГО ЭЛЕМЕНТА
СПИСКА *)
VAR СУММА: REAL;
BEGIN
    СУММА:=0;
    WHILE ПЕРВЫЙ <> NIL DO BEGIN
        СУММА:=СУММА+ПЕРВЫЙ↑.ЗНАЧЕНИЕ;
        (* ПЕРЕЙДЕМ К СЛЕДУЮЩЕМУ ЭЛЕМЕНТУ
СПИСКА *)
    END

```

```

ПЕРВЫЙ:=ПЕРВЫЙ↑.СВЯЗЬ
END;
СУММА-ЗНАЧЕНИЙ:=СУММА
END;

```

Теперь напомним процедуру для вставки в список перед данным элементом нового элемента. Предполагается, что данный элемент — не первый в списке:

```

PROCEDURE ВСТАВКА(ПЕРВЫЙ,ДАННЫЙ,НОВЫЙ:
АДРЕС-ЭЛЕМЕНТА);
(* ПЕРВЫЙ — УКАЗАТЕЛЬ НА ПЕРВЫЙ ЭЛЕМЕНТ
СПИСКА
ДАННЫЙ — УКАЗАТЕЛЬ НА ЭЛЕМЕНТ, ПЕРЕД
КОТОРЫМ ВЫПОЛНЯЕТСЯ ВСТАВКА,
НОВЫЙ — УКАЗАТЕЛЬ НА ВСТАВЛЯЕМЫЙ
ЭЛЕМЕНТ *)
BEGIN
(* НАЙДЕМ ЭЛЕМЕНТ, ПРЕДШЕСТВУЮЩИЙ
ДАННОМУ *)
WHILE ПЕРВЫЙ↑.СВЯЗЬ<> ДАННЫЙ DO
ПЕРВЫЙ:=ПЕРВЫЙ↑.СВЯЗЬ;
(* ТЕПЕРЬ ПЕРВЫЙ↑.СВЯЗЬ=ДАННЫЙ *)
НОВЫЙ↑.СВЯЗЬ:=ДАННЫЙ;
ПЕРВЫЙ↑.СВЯЗЬ:=НОВЫЙ
END;

```

Напомним функцию, связывающую элементы списка в обратном порядке (бывший первый элемент становится последним, бывший последний — первым); результатом функции будет адрес бывшего последнего (а в конце работы — первого) элемента списка:

```

FUNCTION НАОБОРОТ(ПЕРВЫЙ: АДРЕС-ЭЛЕМЕНТА):
АДРЕС-ЭЛЕМЕНТА;
VAR ПРЕДЫДУЩИЙ,ТЕКУЩИЙ,СЛЕДУЮЩИЙ:
АДРЕС-ЭЛЕМЕНТА;
BEGIN
ПРЕДЫДУЩИЙ:=NIL; ТЕКУЩИЙ:=ПЕРВЫЙ;
WHILE ТЕКУЩИЙ <> NIL DO BEGIN
(* ЗАПОМНИМ АДРЕС СЛЕДУЮЩЕГО ЭЛЕМЕНТА
СПИСКА *)
СЛЕДУЮЩИЙ:=ТЕКУЩИЙ↑.СВЯЗЬ;
(* ПОМЕСТИМ В КОМПОНЕНТУ СВЯЗЬ ТЕКУЩЕГО
ЭЛЕМЕНТА АДРЕС ПРЕДЫДУЩЕГО ЭЛЕМЕНТА
СПИСКА *)
ТЕКУЩИЙ↑.СВЯЗЬ:=ПРЕДЫДУЩИЙ;
(* В ЧАСТИ СПИСКА ОТ ПЕРВОГО ЭЛЕМЕНТА ДО

```

```

    ТЕКУЩЕГО СВЯЗИ ПЕРЕСТАВЛЕНЫ НАОБОРОТ.
    ПРОДВИНЕМСЯ К СЛЕДУЮЩЕМУ ЭЛЕМЕНТУ
    СПИСКА *)
    ПРЕДЫДУЩИЙ:=ТЕКУЩИЙ;
    ТЕКУЩИЙ:=СЛЕДУЮЩИЙ
END;
    (* СПИСОК ПРОЙДЕН ДО КОНЦА. ПЕРЕМЕННАЯ
    ПРЕДЫДУЩИЙ ХРАНИТ АДРЕС БЫВШЕГО
    ПОСЛЕДНЕГО ЭЛЕМЕНТА СПИСКА *)
    НАОБОРОТ:=ПРЕДЫДУЩИЙ
END;

```

Напишем функцию, копирующую исходный список; ее результат — адрес первого элемента нового списка:

```

FUNCTION КОПИЯ(СТАРЫЙ: АДРЕС-ЭЛЕМЕНТА):
    АДРЕС-ЭЛЕМЕНТА;
    (* СТАРЫЙ — УКАЗАТЕЛЬ НА ПЕРВЫЙ ЭЛЕМЕНТ
    КОПИРУЕМОГО СПИСКА *)
    VAR НОВ-ПЕРВЫЙ, ТЕКУЩИЙ, НОВЫЙ:
    АДРЕС-ЭЛЕМЕНТА;
    BEGIN
        (* СОЗДАДИМ ФИКТИВНЫЙ ЭЛЕМЕНТ, КОТОРЫЙ
        БУДЕТ СТОЯТЬ ПЕРЕД ПЕРВЫМ ЭЛЕМЕНТОМ
        НОВОГО СПИСКА *)
        NEW(НОВ-ПЕРВЫЙ); ТЕКУЩИЙ:=НОВ-ПЕРВЫЙ;
        WHILE СТАРЫЙ <> NIL DO BEGIN
            (* СКОПИРУЕМ ТЕКУЩИЙ ЭЛЕМЕНТ *)
            NEW(НОВЫЙ); НОВЫЙ↑.ЗНАЧЕНИЕ:=
            СТАРЫЙ↑.ЗНАЧЕНИЕ;
            (* ПРИСОЕДИНИМ НОВЫЙ ЭЛЕМЕНТ К КОПИИ *)
            ТЕКУЩИЙ↑.СВЯЗЬ:=НОВЫЙ;
            (* ПРОДВИНЕМСЯ ПО ОБОИМ СПИСКАМ *)
            СТАРЫЙ:=СТАРЫЙ↑.СВЯЗЬ;
            ТЕКУЩИЙ:=ТЕКУЩИЙ↑.СВЯЗЬ;
        END;
        (* ОТМЕТИМ КОНЕЦ СКОПИРОВАННОГО СПИСКА *)
        ТЕКУЩИЙ↑.СВЯЗЬ:=NIL;
        (* ПРИСВОИМ ИМЕНИ ФУНКЦИИ ЗНАЧЕНИЕ—
        РЕЗУЛЬТАТ *)
        КОПИЯ:=НОВ-ПЕРВЫЙ↑.СВЯЗЬ;
        (* ВЕРНЕМ СТАВШИЙ НЕНУЖНЫМ ФИКТИВНЫЙ
        УЗЕЛ *)
        DISPOSE(НОВ-ПЕРВЫЙ);
    END;

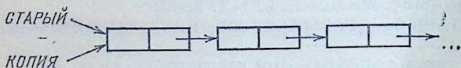
```


указателя **НОВЫЙ**, а компонента **СЛЕДУЮЩИЙ** — посредством указателя **ТЕКУЩИЙ**.

В данном случае по-разному обращаться к одной динамической переменной удобно. Хуже, когда такие обращения производятся неосознанно, в результате ошибки. Пусть, например, вместо обращения к функции **КОПИЯ** мы решили ограничиться присваиванием указателю:

КОПИЯ:=СТАРЫЙ

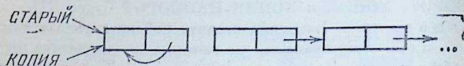
Мы получили бы следующую картину:



(сравните ее с результатом работы функции копирования списка).
Выполнив присваивание

КОПИЯ↑.СВЯЗЬ:=КОПИЯ

мы удалили бы из списка все элементы, кроме первого:



Доступ ко второму и последующим элементам списка был бы утерян, хотя указатель **СТАРЫЙ** мы не трогали.

Похожие ситуации встречаются в повседневной жизни, когда продают два билета на одно место. Если бы далее события развивались как в программировании, произошло бы следующее. Пришедший первым спокойно садится на указанное в билете место. Пришедший вторым также спокойно усаживается сверху, при этом первый бесследно исчезает. И быть может, его будут горько оплакивать, недоумевая, куда он мог деться. А виновой всему — два билета на одно место...

З а м е ч а н и е 3. Функция **СЧИТАЛКА** посредством указателей и динамических переменных идеально моделирует ход считалки и выбывание отвергнутых кандидатов. Однако внимательный читатель, наверняка, заметил, что ее работу можно ускорить. Действительно, когда в круге останется немного людей, а **М** велико, функция много раз обойдет всех, прежде чем остановится перед удаляемым человеком. Если же записать цикл отсчета **М**-го человека в виде

```
FOR J:=1 TO (M-1) MOD K DO
  ТЕКУЩИЙ:=ТЕКУЩИЙ.↑СЛЕДУЮЩИЙ
```

функция уже не будет делать лишних оборотов, а результат получится тем же самым.

Мы не сделали указанного усовершенствования в основной части раздела по двум причинам. Во-первых, чтобы не отвлекаться от основной темы. Во-вторых, существует еще более эффективное решение задачи о считалке. Напомним, что в задаче требуется определить номер человека, который останется в круге последним. Обозначим искомый номер через $U(M, N)$. Можно показать, что числа $U(M, N)$ удовлетворяют рекуррентному соотношению

$$U(M, N) = (M + U(M, N-1) - 1) \bmod N + 1$$

Действительно, первым выйдет из круга человек с номером

$$M_1 = (M - 1) \bmod N + 1$$

(докажите!). Перенумеруем теперь оставшихся людей по кругу, начиная с $(M_1 + 1)$ -го, числами $1, 2, \dots, N-1$. В новой нумерации номер «победителя считалки» по определению равен $U(M, N-1)$, а в первоначальной — $(M_1 + U(M, N-1) - 1) \bmod N + 1$. Поэтому

$$U(M, N) = ((M - 1) \bmod N + 1 + U(M, N-1) - 1) \bmod N + 1$$

Осталось заметить, что внутреннее вычисление остатка можно убрать.

Очевидно, что $U(M, 1) = 1$. Значит, $U(M, N)$ относительно N является рекуррентной последовательностью первого порядка. Для вычисления ее элементов применим развитый в разд. 3.3 механизм. Небольшая тонкость состоит в том, что в данном случае удобнее вычислять не $U(M, N)$, а $U(M, N) - 1$ — тогда мы избавимся от вычитаний и прибавлений единицы.

```

FUNCTION ЕЩЕ-ОДНА-СЧИТАЛКА(N, M: INTEGER):
INTEGER;
(* ФУНКЦИЯ ВЫЧИСЛЯЕТ N-Й ЭЛЕМЕНТ
РЕКУРРЕНТНОЙ ПОСЛЕДОВАТЕЛЬНОСТИ U(M, K),
ЗАДАННОЙ СООТНОШЕНИЯМИ:
U(M, 1) = 1,
U(M, K) = (M + U(M, K-1) - 1) MOD K + 1, K > 1 *)
VAR C: INTEGER;
(* C — УМЕНЬШЕННЫЙ НА ЕДИНИЦУ
ОЧЕРЕДНОЙ ЭЛЕМЕНТ ПОСЛЕДОВАТЕЛЬНОСТИ *)
BEGIN
  C := 0;
  FOR K := 2 TO N DO C := (C + M) MOD K;
  ЕЩЕ-ОДНА-СЧИТАЛКА := C + 1
END;
```


Функция ЕЩЕ-ОДНА-СЧИТАЛКА получилась более короткой и экономной в смысле расходования как времени, так и памяти. Правда, догадаться до нее труднее, чем до функции СЧИТАЛКА. Но, как говорится, чем раньше начнешь писать программу, тем длиннее она получится.

З а м е ч а н и е 4. Если посредством обращения к стандартной процедуре NEW создается динамическая переменная, которая является записью с вариантами, причем известно, что эта переменная на протяжении всего своего существования будет представлять лишь какой-то один вариант записи, можно сообщить об этом паскаль-машине. Для этого следует указать в обращении к процедуре NEW значение дискриминанта для нужного варианта:

NEW(⟨указатель⟩, ⟨значение дискриминанта⟩)

Такая инструкция в сравнении с обращением без значения дискриминанта может дать экономию памяти, поскольку память будет отведена только для одного (быть может, не самого большого) варианта записи.

Если данный вариант записи в свою очередь содержит вариантную часть и заранее известно, какой из подвариантов будет постоянно использоваться, можно указать значение и второго дискриминанта:

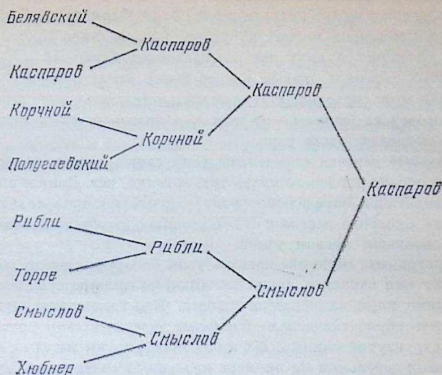
NEW(⟨указатель⟩, ⟨значение дискриминанта 1⟩,
⟨значение дискриминанта 2⟩)

и т. д.

При освобождении с помощью процедуры DISPOSE памяти, занимаемой записью с вариантами, в качестве параметров DISPOSE следует обязательно указывать те же значения дискриминантов, что были в обращении к процедуре NEW при создании этой записи.

4.9. Бинарные деревья

Даже людям, далеким от спорта, знакома формула «проигравший выбывает». По этой формуле проводятся соревнования за различные кубки, она, например, регламентирует заключительный этап отбора претендента на матч с чемпионом мира по шахматам. Многим знаком и способ представления информации о кубковых состязаниях — так называемая матчевая сетка. Вот как она выглядит для цикла претендентских матчей по шахматам 1984—85 гг.:

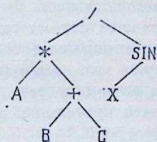


В программировании подобная структура данных называется бинарным деревом. Узлы дерева хранят фамилии победителей матчей, ветви связывают каждого победителя с участниками матча. Самый правый на рисунке узел «Каспаров» описывает победителя претендентского цикла и называется корнем. Все дерево как бы вырастает из него. Победители четвертьфинальных и полуфинальных матчей описываются корневыми узлами соответствующих под-деревьев, ведь они (победители) — сильнейшие в своей части матчевой сетки.

К сожалению, спортивная аналогия страдает некоторыми изъянами. Во-первых, изображенное дерево является «слишком правильным» — в общем случае из узлов бинарного дерева может исходить 0, 1 или 2 ветви. Во-вторых, в отличие от спортивных изданий в литературе по программированию деревья принято изображать корнем вверх. Например, алгебраическое выражение

$$A * (B + C) / \sin(X)$$

обычно представляют в ЭВМ в виде бинарного дерева



Определим теперь бинарное дерево более формально. *Бинарное дерево* есть конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня.

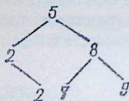
Левая ветвь, исходящая из произвольного узла, ведет в корень левого поддерева (если оно не пусто), а правая ветвь — в корень правого поддерева этого узла.

Бинарные деревья определены рекурсивно, в терминах самих деревьев, однако порочного круга тут, конечно, нет. Данное определение аналогично определению рекуррентной последовательности, поскольку бинарное дерево с $N > 0$ узлами определяется через деревья с меньшим числом узлов.

В программах на языке паскаль узлы бинарного дерева обычно описывают как записи с двумя полями-указателями и одним или несколькими информационными полями. Простота описания и эффективность обработки делают бинарные деревья весьма употребительной структурой данных. Мы продемонстрируем методы работы с бинарными деревьями на примере программы сортировки (упорядочения) последовательности натуральных чисел. Элементы этой последовательности следует прочитать и напечатать в порядке возрастания. Признак конца последовательности — число 0. В процессе работы программа будет строить бинарное дерево, каждый узел которого содержит одно из сортируемых чисел и указатели на левое и правое поддеревья, причем все числа в узлах левого поддерева будут меньше, чем в корне, а в узлах правого — больше или равны. Например, последовательность чисел

5 2 8 7 2 9

приведет к созданию такого дерева:



После того как последовательность будет прочитана и дерево полностью построено, останется напечатать значения, хранящиеся в узлах, в таком рекурсивно определяемом порядке: сначала все значения из левого поддерева узла, затем значение из самого узла и, наконец, значения из правого поддерева. Из способа построения дерева следует, что элементы прочитанной последовательности напечатаются в порядке возрастания.

Обход всех узлов дерева и печать хранящихся в них значений — действия довольно хитрые, ведь каждый узел содержит указатели на два поддерева, т. е. является разветвлением. Спускаясь по левой ветви, мы должны запомнить адрес разветвления, чтобы потом вернуться в узел, напечатать хранящееся в нем значение и отправиться по правой ветви. Адреса разветвлений будем запоминать в массиве и извлекать их оттуда по принципу «последним запомнили — первым вспомнили»:

```

PROGRAM СОРТИРОВКА (INPUT, OUTPUT);
CONST ПРИЗНАК-КОНЦА=0;
TYPE АДРЕС-УЗЛА=↑УЗЕЛ;
  УЗЕЛ=RECORD
    ЗНАЧЕНИЕ: INTEGER;
    ЛЕВЫЙ, ПРАВЫЙ: АДРЕС-УЗЛА
  END;
VAR ОЧЕРЕДНОЕ-ЧИСЛО: INTEGER;
    КОРЕНЬ: АДРЕС-УЗЛА;
PROCEDURE ВСТАВКА(КОРЕНЬ: АДРЕС-УЗЛА,
  ЧИСЛО: INTEGER);
(* ПРОЦЕДУРА ДОБАВЛЯЕТ К НЕПУСТОМУ ДЕРЕВУ
  С ДАННЫМ КОРНЕМ УЗЕЛ, ХРАНЯЩИЙ ДАННОЕ
  ЧИСЛО,
  ПРИЧЕМ ДЛЯ КАЖДОГО УЗЛА ОСТАЕТСЯ
  СПРАВЕДЛИВЫМ УТВЕРЖДЕНИЕ, ЧТО ВСЕ ЗНАЧЕНИЯ
  В ЛЕВОМ ПОДДЕРЕВЕ МЕНЬШЕ, ЧЕМ В УЗЛЕ,
  А В ПРАВОМ НЕ МЕНЬШЕ *)
VAR ТЕКУЩИЙ, ПРЕДЫДУЩИЙ: АДРЕС-УЗЛА;
(* ТЕКУЩИЙ ИЩЕТ В ДЕРЕВЕ ПУСТУЮ ССЫЛКУ,
  А ПРЕДЫДУЩИЙ ХРАНИТ АДРЕС УЗЛА,
  В КОТОРОМ ЭТА ССЫЛКА НАЙДЕНА *)
BEGIN
  ТЕКУЩИЙ:=КОРЕНЬ;
  WHILE ТЕКУЩИЙ <> NIL DO BEGIN
    ПРЕДЫДУЩИЙ:=ТЕКУЩИЙ;
    IF ЧИСЛО<ТЕКУЩИЙ↑.ЗНАЧЕНИЕ THEN
      ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ЛЕВЫЙ
    ELSE ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ПРАВЫЙ
  END; (* WHILE *)
  NEW(ТЕКУЩИЙ);
  WITH ТЕКУЩИЙ↑ DO BEGIN
    ЗНАЧЕНИЕ:=ЧИСЛО;
    ЛЕВЫЙ:=NIL; ПРАВЫЙ:=NIL
  END; (* WITH *)
  (* «ПРИВЯЖЕМ» НОВЫЙ УЗЕЛ К ДЕРЕВУ *)

```

```

IF ЧИСЛО < ПРЕДЫДУЩИЙ↑.ЗНАЧЕНИЕ THEN
    ПРЕДЫДУЩИЙ↑.ЛЕВЫЙ:=ТЕКУЩИЙ
ELSE ПРЕДЫДУЩИЙ↑.ПРАВЫЙ:=ТЕКУЩИЙ
END; (* ПРОЦЕДУРЫ ВСТАВКА *)
PROCEDURE ОБХОД-ДЕРЕВА (КОРЕНЬ: АДРЕС-УЗЛА);
(* ПРОЦЕДУРА ОБХОДИТ УЗЛЫ ДЕРЕВА В ПОРЯДКЕ
    ЛЕВОЕ ПОДДЕРЕВО, УЗЕЛ, ПРАВОЕ ПОДДЕРЕВО
    И ПЕЧАТАЕТ ХРАНЯЩИЕСЯ В УЗЛАХ ЗНАЧЕНИЯ *)
LABEL 13;
(* МЕТКА ДЛЯ ВЫХОДА ИЗ ФУНКЦИИ
    ПРИ ПЕРЕПОЛНЕНИИ ХРАНИЛИЩА АДРЕСОВ
    РАЗВЕТВЛЕНИЙ *)
CONST ПРЕДЕЛ=100;
(* МАКСИМАЛЬНОЕ ЧИСЛО АДРЕСОВ
    РАЗВЕТВЛЕНИЙ,
    КОТОРЫЕ МОГУТ ХРАНИТЬСЯ ОДНОВРЕМЕННО *)
VAR ТЕКУЩИЙ: АДРЕС-УЗЛА;
    ХРАНИЛИЩЕ-АДРЕСОВ: RECORD
        НОМЕР: 0..ПРЕДЕЛ;
        АДРЕСА: ARRAY [1..ПРЕДЕЛ] OF АДРЕС-УЗЛА
    END;
BEGIN
    ТЕКУЩИЙ:=КОРЕНЬ;
    WITH ХРАНИЛИЩЕ-АДРЕСОВ DO BEGIN
        НОМЕР:=0;
        (* ПРОДОЛЖАЕМ ОБХОД, ПОКА ЛИБО ЕСТЬ КУДА
            СПУСКАТЬСЯ, Т. Е. ТЕКУЩИЙ <> NIL,
            ЛИБО ОСТАЛИСЬ НЕОБРАБОТАННЫЕ
            РАЗВЕТВЛЕНИЯ,
            Т. Е. НОМЕР <> 0 *)
        WHILE (ТЕКУЩИЙ <> NIL) OR (НОМЕР <> 0) DO
            IF ТЕКУЩИЙ <> NIL THEN BEGIN
                (* СПУСТИМСЯ, ЗАПОМНИВ АДРЕС
                    РАЗВЕТВЛЕНИЯ *)
                IF НОМЕР=ПРЕДЕЛ THEN BEGIN
                    WRITELN;
                    WRITELN(' ***СООБЩАЕТ ОБХОД' ,
                        (' ДЕРЕВА ');
                    WRITELN(' УВЕЛИЧЬТЕ РАЗМЕР ');
                    WRITELN(' МАССИВА АДРЕСА ');
                    GOTO 13
                END; (* IF *)
                НОМЕР:=НОМЕР+1;
                АДРЕСА[НОМЕР]:=ТЕКУЩИЙ;
                ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ЛЕВЫЙ
            
```

```

END
ELSE BEGIN
  (* ТЕКУЩИЙ=NIL, Т. Е. ОБХОД ПОДДЕРЕВ
    ОКОНЧЕН; ИЗВЛЕЧЕМ ПОСЛЕДНИЙ ИЗ
    ЗАПОМНЕННЫХ АДРЕСОВ
    РАЗВЕТВЛЕНИЯ,
    НАПЕЧАТАЕМ ЗНАЧЕНИЕ
    ИЗ ЭТОГО УЗЛА
    И СПУСТИМСЯ ПО ПРАВОЙ ВЕТВИ *)
  ТЕКУЩИЙ:=АДРЕСА[НОМЕР];
  НОМЕР:=НОМЕР-1;
  WRITE(ТЕКУЩИЙ↑.ЗНАЧЕНИЕ);
  ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ПРАВЫЙ
END (* IF И WHILE *)
END; (* WITH *)
13:END; (* ПРОЦЕДУРЫ ОБХОД-ДЕРЕВА *)
BEGIN
  (* ГЛАВНАЯ ПРОГРАММА *)
  WRITELN(' СОРТИРОВКА ЧИСЕЛ В ПОРЯДКЕ',
  (' ВОЗРАСТАНИЯ');
  WRITELN(' ИСХОДНАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ:');
  (* ФОРМИРОВАНИЕ ДЕРЕВА *)
  КОРЕНЬ:=NIL;
  (* СНАЧАЛА СОЗДАДИМ КОРНЕВОЙ УЗЕЛ *)
  READ(ОЧЕРЕДНОЕ-ЧИСЛО);
  IF ОЧЕРЕДНОЕ-ЧИСЛО <> ПРИЗНАК-КОНЦА THEN
  BEGIN
    WRITE(ОЧЕРЕДНОЕ-ЧИСЛО);
    NEW(КОРЕНЬ);
    WITH КОРЕНЬ↑ DO BEGIN
      ЗНАЧЕНИЕ:=ОЧЕРЕДНОЕ-ЧИСЛО;
      ЛЕВЫЙ:=NIL; ПРАВЫЙ:=NIL
    END; (* WITH *)
    READ(ОЧЕРЕДНОЕ-ЧИСЛО);
    (* ЦИКЛ ВСТАВКИ ЭЛЕМЕНТОВ
      ПОСЛЕДОВАТЕЛЬНОСТИ В НЕПУСТОЕ ДЕРЕВО *)
    WHILE ОЧЕРЕДНОЕ-ЧИСЛО <>
    ПРИЗНАК-КОНЦА DO BEGIN
      WRITE(ОЧЕРЕДНОЕ-ЧИСЛО);
      ВСТАВКА(КОРЕНЬ, ОЧЕРЕДНОЕ-ЧИСЛО);
      READ(ОЧЕРЕДНОЕ-ЧИСЛО)
    END (* WHILE *)
  END; (* IF *)
  WRITELN;
  (* ДЕРЕВО СФОРМИРОВАНО. ОБОЙДЕМ ЕГО

```



```

И НАПЕЧАТАЕМ ХРАНЯЩИЕСЯ В УЗЛАХ
ЗНАЧЕНИЯ *)
WRITELN(' УПОРЯДОЧЕННАЯ ',
' ПОСЛЕДОВАТЕЛЬНОСТЬ:');
ОБХОД-ДЕРЕВА (КОРЕНЬ)
END.

```

Если элементы исходной последовательности распределены случайно, получившееся бинарное дерево будет достаточно хорошо сбалансированным, т. е. левое поддерево любого узла будет содержать примерно столько же узлов, сколько и правое. При этом время, необходимое для вставки нового узла, по порядку величины не превышает $\log_2 N$, где N — число элементов последовательности, а вся сортировка выполнится за время, пропорциональное $N \log_2 N$. Вставка в бинарное дерево — хороший метод сортировки. Далее, процедуру ВСТАВКА нетрудно переделать так, чтобы с ее помощью можно было осуществлять поиск с вставкой, т. е. искать в бинарном дереве данное значение, а в случае неудачи поиска добавлять узел с этим значением. Приводимая ниже функция работает при тех же предположениях, что и процедура ВСТАВКА; ее результат — адрес узла дерева, содержащего данное число:

```

FUNCTION ПОИСК-И-ВСТАВКА(
  КОРЕНЬ: АДРЕС-УЗЛА; ЧИСЛО: INTEGER);
  АДРЕС-УЗЛА;
LABEL 99;
(* МЕТКА ДЛЯ ВЫХОДА ИЗ ФУНКЦИИ ПРИ УДАЧНОМ
ПОИСКЕ *)
VAR ТЕКУЩИЙ, ПРЕДЫДУЩИЙ: АДРЕС-УЗЛА;
BEGIN
  ТЕКУЩИЙ:=КОРЕНЬ;
  WHILE ТЕКУЩИЙ <> NIL DO BEGIN
    ПРЕДЫДУЩИЙ:=ТЕКУЩИЙ;
    IF ЧИСЛО < ТЕКУЩИЙ↑.ЗНАЧЕНИЕ THEN
      ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ЛЕВЫЙ
    ELSE
      IF ЧИСЛО > ТЕКУЩИЙ↑.ЗНАЧЕНИЕ THEN
        ТЕКУЩИЙ:=ТЕКУЩИЙ↑.ПРАВЫЙ
      ELSE (* НАШЛИ УЗЕЛ С ДАННЫМ ЗНАЧЕНИЕМ *)
        GOTO 99
  END; (* WHILE *)
  NEW(ТЕКУЩИЙ);
  WITH ТЕКУЩИЙ↑ DO BEGIN
    ЗНАЧЕНИЕ:=ЧИСЛО;
    ЛЕВЫЙ:=NIL; ПРАВЫЙ:=N
  END; (* WITH *)

```

```

IF ЧИСЛО <ПРЕДЫДУЩИЙ↑.ЗНАЧЕНИЕ THEN
  ПРЕДЫДУЩИЙ↑.ЛЕВЫЙ:=ТЕКУЩИЙ
ELSE ПРЕДЫДУЩИЙ↑.ПРАВЫЙ:=ТЕКУЩИЙ;
99: ПОИСК-И-ВСТАВКА:=ТЕКУЩИЙ
END;

```

Если дерево хорошо сбалансировано, время поиска данного значения пропорционально $\log_2 N$, тогда как последовательный поиск (см. разд. 3.5) требует времени, пропорционального N , где N — количество хранящихся значений. Поиск с вставкой по дереву — один из самых быстрых методов поиска посредством сравнений, допускающий, кроме того, эффективное добавление новых значений. Еще одно важное достоинство состоит в том, что, если дерево получено путем многократных применений функции ПОИСК-И-ВСТАВКА, легко напечатать хранящиеся в нем значения в порядке возрастания — достаточно применить процедуру ОБХОД-ДЕРЕВА.

Вернемся к программе СОРТИРОВКА. Самое сложное место в ней — процедура ОБХОД-ДЕРЕВА. Докажем утверждение о цикле, составляющем основу этой процедуры, из которого будет следовать корректность предложенного способа обхода дерева.

У т в о р ж д е н и е. Пусть перед проверкой условий, входящих в заголовок цикла, указатель ТЕКУЩИЙ указывает на корень бинарного дерева с $n \geq 0$ узлами, а массив АДРЕСА хранит $m \geq 0$ указателей. Тогда в результате нескольких выполнений цикла при отсутствии выхода за границу массива АДРЕСА 1) узлы данного бинарного дерева будут пройдены в рекурсивно определяемом порядке: левое поддерево, корень, правое поддерево, 2) в массиве АДРЕСА останутся те же m указателей, 3) паскаль-машина вновь окажется перед заголовком цикла со значением ТЕКУЩИЙ = NIL.

(Под прохождением данного узла понимается выполнение инструкции

WRITE (ТЕКУЩИЙ↑.ЗНАЧЕНИЕ)

когда ТЕКУЩИЙ хранит адрес узла. В общем случае значение можно не только печатать, но и выполнять с ним самые разнообразные действия — процедура ОБХОД-ДЕРЕВА останется корректной.)

Д о к а з а т е л ь с т в о. Воспользуемся методом математической индукции. Пусть сначала $n=0$. Тогда паскаль-машина с самого начала находится в пущном состоянии, поскольку ТЕКУЩИЙ = NIL. Пусть теперь $n > 0$, т. е. в начальный момент ТЕКУЩИЙ $\neq$ NIL. Тогда в результате выполнения условной инструкции указатель ТЕКУЩИЙ будет сохранен в $(m+1)$ -м элементе массива АДРЕСА, а паскаль-машина окажется перед заголовком цикла со значением ТЕКУЩИЙ, указывающим на левое поддерево дан-

ного корня. Поскольку оно содержит меньше, чем n узлов, его обход выполнится должным образом, паскаль-машина вернется к заголовку цикла при ТЕКУЩИЙ=NIL, а массив АДРЕСА по-прежнему будет содержать $m+1 > 0$ значений. Затем после входа в цикл в результате выполнения условной инструкции указатель ТЕКУЩИЙ вновь получит первоначальное значение; в массиве АДРЕСА останется m указателей; будет пройден корневой узел и паскаль-машина окажется перед заголовком цикла со значением ТЕКУЩИЙ, указывающим на правое поддерево данного корня. Правое поддерево содержит меньше, чем n узлов, поэтому оно будет пройдено должным образом. Утверждение доказано.

Чтобы теперь убедиться в правильности процедуры ОБХОД-ДЕРЕВА, достаточно применить доказанное утверждение к ситуации, возникающей в начале работы процедуры, когда указатель ТЕКУЩИЙ указывает на корень дерева, а массив АДРЕСА не хранит ни одного указателя.

4.10. Рекурсивные процедуры

Процедура называется рекурсивной, если она прямо или косвенно (т. е. посредством других процедур) обращается к самой себе.

Рекурсивные процедуры наиболее естественно применять для обработки рекурсивно определяемых структур данных, таких, например, как деревья. Вот как можно переписать процедуру ОБХОД-ДЕРЕВА из разд. 4.9:

```
PROCEDURE РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА (КОРЕНЬ:
    АДРЕС-УЗЛА);
BEGIN
    IF КОРЕНЬ <> NIL THEN BEGIN
        РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА
            (КОРЕНЬ↑.ЛЕВЫЙ);
        WRITE (КОРЕНЬ↑.ЗНАЧЕНИЕ);
        РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА
            (КОРЕНЬ↑.ПРАВЫЙ)
    END
END;
```

Текст этой процедуры, по существу, повторяет определение порядка обхода узлов дерева, данное в разд. 4.9, и в отличие от процедуры ОБХОД-ДЕРЕВА не требует доказательства правильности — она (правильность) очевидна. Смысл употребления рекурсивных процедур — в облегчении написания, понимания и модификации программ. Например, если понадобится обойти узлы дерева в порядке (левое поддерево, правое поддерево, корень),

в процедуре РЕКУРСИВНЫЙ_ОБХОД_ДЕРЕВА достаточно поменять местами две инструкции:

```
WRITE (КОРЕНЬ↑.ЗНАЧЕНИЕ)
```

и

```
РЕКУРСИВНЫЙ_ОБХОД_ДЕРЕВА  
(КОРЕНЬ↑.ПРАВЫЙ)
```

Переделать соответствующим образом нерекурсивную процедуру ОБХОД_ДЕРЕВА значительно сложнее.

Поясним, как работает процедура РЕКУРСИВНЫЙ_ОБХОД_ДЕРЕВА. Если дерево не пусто, она порождает помощника — свою копию — и посылает его для обхода левого поддерева, сама же остается ждать. Помощник вернулся — процедура печатает значение из корневого узла, порождает нового помощника и посылает его на правому поддереву. Дождавшись возвращения помощника, процедура завершает работу. Таким образом, во время обхода дерева в паскаль-машине одновременно существует несколько копий процедуры, из которых одна активна (проверяет дерево на пустоту, порождает новую копию или печатает значение), а все остальные пассивны — ждут завершения работы порожденных ими копий. В каждой из них параметр КОРЕНЬ имеет свое значение и указывает на корень того поддерева, которое поручено обойти данной копии. Тем самым адреса разветвлений запоминаются автоматически — всю «бухгалтерию» паскаль-машина берет на себя.

Мы видим, что рекурсивные процедуры — средство, удобное для человека, но довольно накладное для ЭВМ, и пользоваться этим средством нужно только тогда, когда выигрыш для человека действительно велик. Приведем пример неразумного, на наш взгляд, употребления рекурсии. Сначала запишем нерекурсивный вариант функции, подсчитывающей число элементов списка:

```
TYPE АДРЕС-ЭЛЕМЕНТА = ↑ЭЛЕМЕНТ-СПИСКА;  
ЭЛЕМЕНТ-СПИСКА = RECORD  
    ЗНАЧЕНИЕ: REAL;  
    СВЯЗЬ: АДРЕС-ЭЛЕМЕНТА  
END;  
...  
FUNCTION ДЛИНА1 (ПЕРВЫЙ: АДРЕС-ЭЛЕМЕНТА):  
    INTEGER;  
(* ПЕРВЫЙ СОДЕРЖИТ АДРЕС ПЕРВОГО ЭЛЕМЕНТА  
    СПИСКА *)  
VAR ДЛИНА: INTEGER;  
BEGIN  
    ДЛИНА := 0;  
    WHILE ПЕРВЫЙ <> NIL DO BEGIN
```

```

ДЛИНА:=ДЛИНА+1;
ПЕРВЫЙ:=ПЕРВЫЙ↑.СВЯЗЬ
END;
ДЛИНА1:=ДЛИНА
END;
А вот как выглядит рекурсивное решение:
FUNCTION ДЛИНА2(ПЕРВЫЙ: АДРЕС-ЭЛЕМЕНТА):
INTEGER;
BEGIN
  IF ПЕРВЫЙ=NIL THEN ДЛИНА2:=0
  ELSE ДЛИНА2:=1+ДЛИНА2(ПЕРВЫЙ↑.СВЯЗЬ)
END;

```

Приведем телефонную аналогию, поясняющую работу этих функций. Пусть имеется N человек, причем k -й ($0 < k < N$) знает только телефон следующего, $(k+1)$ -го человека, а N -й знает только, что он последний. Пусть далее некто, знающий телефон первого человека, решил выяснить, чему равно N . Если он будет действовать не рекурсивно в соответствии с функцией ДЛИНА1, он запастется бумагой, на которой будет вести подсчеты (переменная ДЛИНА), напишет на ней 0 и позвонит первому. Затем он зачеркнет 0, напишет 1 и спросит у первого телефон второго человека, повесит трубку, наберет номер второго, исправит 1 на 2, спросит телефон третьего и т. д., пока не доберется до человека, который сообщит, что он последний. Число, записанное в этот момент на бумаге, и будет служить ответом.

Если же упомянутый некто захочет действовать рекурсивно в соответствии с функцией ДЛИНА2, он позвонит первому и прикажет: «Алло, не вешайте трубку! Сообщите мне, сколько вас». Поскольку первый не знает, сколько их, а трубку вешать нельзя, ему придется пойти к соседям и позвонить второму, сказав те же слова и оставшись ждать ответа. Второй также отправится к соседям и т. д., пока, наконец, последний не сообщит предпоследнему, что он один. Предпоследний прибавит к ответу последнего 1 и скажет, что их двое, и т. д. Наконец, заждавшийся первый услышит в трубке голос второго, прибавит к услышанному числу 1 и пойдет к себе домой, чтобы сообщить результат.

При не рекурсивном способе в каждый момент соединена только одна пара абонентов, при рекурсивном — до N пар. Второй способ хуже по двум причинам: он медленнее в 3—5 раз и, что более важно, для длинных списков (содержащих порядка 10 000 элементов) он не будет работать, поскольку паскаль-машина исчерпает пространство для хранения копий функции ДЛИНА2 (возвращаясь к телефонной аналогии, можно сказать, что АТС будет перегружена и к очередному человеку дозвониться не удастся).

Рассмотрим теперь задачу, в которой употребление рекурсивной процедуры оказывается весьма уместным. Головоломка «Ханойская башня» известна многим. Имеется три стержня, назовем их А, В, С. На стержне А, подобно детской пирамидке, надеты кольца в порядке убывания диаметров. За один ход разрешается переложить верхнее кольцо с любого стержня на любой другой с одним ограничением: большее кольцо нельзя класть на меньшее. Требуется, чтобы в конце концов все кольца оказались надеты на стержень В в порядке убывания диаметров. Например, если колец три, к цели ведет такая последовательность перекладываний:

А—>В
 А—>С
 В—>С
 А—>В
 С—>А
 С—>В
 А—>В

Пусть колец N. Будем рассуждать так. Пусть мы сумели переложить N—1 колец с А на С, используя стержень В как промежуточный. После этого нам остается переложить самое большое кольцо с А на В и затем переложить N—1 колец с С на В, используя как промежуточный уже стержень А. В свою очередь, чтобы переложить N—1 колец, надо научиться сначала перекладывать N—2 колец и т. д. В результате напрашивается рекурсивное решение:

```

PROCEDURE ХАНОЙ(N: INTEGER);
TYPE СТЕРЖЕНЬ=CHAR;
PROCEDURE ПЕРЕЛОЖИТЬ(N: INTEGER;
  ИСХОДНЫЙ, ЦЕЛЕВОЙ, ПРОМЕЖУТОЧНЫЙ:
  СТЕРЖЕНЬ);
(* ПРОЦЕДУРА ПЕЧАТАЕТ ПОСЛЕДОВАТЕЛЬНОСТЬ
  ПЕРЕКЛАДЫВАНИЙ, НЕОБХОДИМЫХ
  ДЛЯ ПЕРЕМЕЩЕНИЯ
  N КОЛЕЦ С ИСХОДНОГО СТЕРЖНЯ
  НА ЦЕЛЕВОЙ С ПОМОЩЬЮ ПРОМЕЖУТОЧНОГО *)
BEGIN
  IF N>0 THEN BEGIN
    ПЕРЕЛОЖИТЬ(N — 1, ИСХОДНЫЙ,
      ПРОМЕЖУТОЧНЫЙ, ЦЕЛЕВОЙ);
    WRITELN(' ', ИСХОДНЫЙ, '—>', ЦЕЛЕВОЙ);
    ПЕРЕЛОЖИТЬ(N—1, ПРОМЕЖУТОЧНЫЙ,
      ЦЕЛЕВОЙ, ИСХОДНЫЙ)
  END
END; (* ПРОЦЕДУРЫ ПЕРЕЛОЖИТЬ *)
  
```


BEGIN

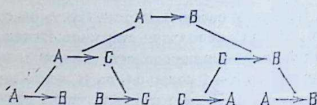
ПЕРЕЛОЖИТЬ (N, 'A', 'B', 'C')

END;

Рекурсивная процедура ПЕРЕЛОЖИТЬ дает изящное решение головоломки. Попытаемся, однако, преобразовать рекурсивное решение в нерекурсивное, хотя бы для того, чтобы получить программу, по которой может действовать человек. (Рекурсивная программа не дает прямого ответа даже на вопрос: куда переложить первое кольцо?) Обратим прежде всего внимание на сходство процедур ПЕРЕЛОЖИТЬ и РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА. Действительно, поставим в соответствие каждому выполнению процедуры

WRITELN (' ', ИСХОДНЫЙ, '—>', ЦЕЛЕВОЙ)

узел бинарного дерева. Первый из двух рекурсивных вызовов будет соответствовать переходу к левому поддереву, второй — правому поддереву. Например, для $N=3$ получается дерево:



В такой трактовке процедуры ПЕРЕЛОЖИТЬ и РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА делают одно и то же — обходят бинарное дерево в порядке (левое поддерево, узел, правое поддерево) и печатают значения, хранящиеся в узлах. В принципе для избавления от рекурсии можно воспользоваться процедурой, похожей на ОБХОД-ДЕРЕВА, но тогда понадобится массив для хранения адресов разветвлений, а главное, придется на самом деле строить дерево. В данном же случае дерево важно только как мысленная конструкция, помогающая найти нерекурсивное решение.

Запишем узлы дерева в порядке их обхода. Так, при $N=3$ самый левый узел получит номер 1, корень — номер 4, самый правый узел — номер 7. Запишем также уровни, на которых расположены узлы, — снизу вверх, начиная с 1. Поскольку при каждом рекурсивном вызове процедуры ПЕРЕЛОЖИТЬ число колец уменьшается на 1, дерево будет содержать N уровней, на K -м уровне будет 2^{N-K} узлов, а всего в дереве будет $2^N - 1$ узлов.

Теперь рассмотрим, как происходит обход дерева. Узлы, расположенные на нижнем уровне, имеют номера 1, 3, ..., $2^{N-1} - 1$, поэтому при обходе мы через раз будем попадать на уровень 1. Спуск на этот уровень происходит так: шаг по правой ветви и несколько (быть может 0) шагов по левым ветвям. Для подъема с нижнего

уровня нужно идти по правым ветвям вверх, пока это возможно, затем сделать шаг вверх по левой ветви. Как следует из текста процедуры ПЕРЕЛОЖИТЬ, при движении по левой ветви происходит перестановка второго и третьего параметров, при движении по правой — первого и третьего. Поставим в соответствие параметрам процедуры ПЕРЕЛОЖИТЬ три переменные типа СТЕРЖЕНЬ; назовем их X, Y, Z. Перестановке параметров при рекурсивном вызове будет соответствовать обмен значений этих переменных, а инструкции

```
WRITELN(' ', ИСХОДНЫЙ, '—>', ЦЕЛЕВОЙ)
```

инструкции

```
WRITELN(' ', X, '—>', Y)
```

Идея нерекурсивного решения состоит в том, чтобы последовательно перебрать номера от 1 до $2^N - 1$, выяснить, на каких уровнях дерева находятся узлы с этими номерами, и при переходе к очередному номеру обменять значения переменных X, Y и Z в соответствии со способом построения дерева. Например, в узел 5 $C \rightarrow A$ мы попадаем, спускаясь из корня; поэтому при переходе к номеру 5 нужно сначала обменять значения X и Z, затем Y и Z. Нетрудно сообразить, что итоговая перестановка значений X, Y и Z, возникающая при перемещении из предыдущего узла в текущий, определяется только четностью номеров уровней этих узлов. Получаем следующий набросок процедуры:

```
PROCEDURE ХАНОЙ2(N: INTEGER);
VAR X, Y, Z: СТЕРЖЕНЬ;
    M: INTEGER;
    ТЕКУЩИЙ-ЧЕТНЫЙ, ПРЕДЫДУЩИЙ-ЧЕТНЫЙ:
        BOOLEAN;
BEGIN
    ПОДХОДЯЩИЕ НАЧАЛЬНЫЕ УСТАНОВКИ
    ПЕРЕМЕННЫХ
    X, Y, Z, ПРЕДЫДУЩИЙ-ЧЕТНЫЙ:
    FOR M:=1 TO (2 В СТЕПЕНИ N)-1 DO BEGIN
        ТЕКУЩИЙ-ЧЕТНЫЙ:=УЗЕЛ НОМЕР M
        НАХОДИТСЯ НА ЧЕТНОМ УРОВНЕ;
        IF ТЕКУЩИЙ-ЧЕТНЫЙ THEN ОБМЕНЯТЬ
            ЗНАЧЕНИЯ (Y, Z)
        ELSE
            IF ПРЕДЫДУЩИЙ-ЧЕТНЫЙ THEN
                ОБМЕНЯТЬ ЗНАЧЕНИЯ (X, Z)
            ELSE BEGIN
                ОБМЕНЯТЬ ЗНАЧЕНИЯ (X, Z);
```

```

        ОБМЕНЯТЬ ЗНАЧЕНИЯ (Y, Z)
    END;
    WRITELN(' ', X, '—>', Y);
    ПРЕДЫДУЩИЙ-ЧЕТНЫЙ:=ТЕКУЩИЙ-ЧЕТНЫЙ
END
END;

```

Для придания процедуре окончательного вида полезно заметить, что номер уровня узла M равен номеру самой правой единицы в двоичном представлении M. Этот факт можно доказать по индукции. А для правильного выбора начальных установок можно мыслить переход к узлу номер 1 как спуск из фиктивного узла, расположенного левее и выше корня:

```

PROCEDURE ХАНОЙ2(N: INTEGER);
TYPE СТЕРЖЕНЬ=CHAR;
VAR X, Y, Z, T: СТЕРЖЕНЬ;
    M, K: INTEGER;
    ТЕКУЩИЙ-ЧЕТНЫЙ, ПРЕДЫДУЩИЙ-ЧЕТНЫЙ:
        BOOLEAN;
FUNCTION СТЕПЕНЬ(M, N: INTEGER): INTEGER;
    VAR P: INTEGER;
    BEGIN
        P:=1;
        WHILE N <>0 DO BEGIN
            WHILE NOT ODD(N) DO BEGIN
                M:=M*M; N:=N DIV 2
            END;
            P:=P*M; N:=N-1
        END;
        СТЕПЕНЬ:=P
    END; (* ФУНКЦИИ СТЕПЕНЬ *)
BEGIN
    X:='C'; Y:='B'; Z:='A';
    ПРЕДЫДУЩИЙ_ЧЕТНЫЙ:=ODD(N);
    FOR M:=1 TO СТЕПЕНЬ(2, N)-1 DO BEGIN
        (* ТЕКУЩИЙ-ЧЕТНЫЙ:=УЗЕЛ M НАХОДИТСЯ
            НА ЧЕТНОМ УРОВНЕ *)
        ТЕКУЩИЙ-ЧЕТНЫЙ:=FALSE; K:=M;
        WHILE NOT ODD(K) DO BEGIN
            ТЕКУЩИЙ-ЧЕТНЫЙ:=NOT ТЕКУЩИЙ-ЧЕТНЫЙ;
            K:=K DIV 2
        END;
        IF ТЕКУЩИЙ-ЧЕТНЫЙ THEN
            (* ОБМЕНЯТЬ ЗНАЧЕНИЯ (Y, Z) *)
            BEGIN T:=Y; Y:=Z; Z:=T END
    END

```



```

ELSE
  IF ПРЕДЫДУЩИЙ-ЧЕТНЫЙ THEN
    (* ОБМЕНЯТЬ (X, Z) *)
    BEGIN T:=X; X:=Z; Z:=T END
  ELSE (* ОБМЕНЯТЬ ЗНАЧЕНИЯ (X, Z), ЗАТЕМ (Y,
    Z) *)
    BEGIN T:=X; X:=Z; Z:=Y; Y:=T END;
  WRITELN(' ', X, '—>', Y);
  ПРЕДЫДУЩИЙ-ЧЕТНЫЙ:=ТЕКУЩИЙ-ЧЕТНЫЙ
END
END;

```

Нам потребовались довольно длинные рассуждения, но зато теперь мы знаем, что при четном N верхнее кольцо надо класть на стержень C , а при нечетном — на B . В целом же рекурсивный вариант, конечно, предпочтительнее, поскольку его гораздо проще написать и понять.

З а м е ч а н и е. Рекурсия — один из способов многократного выполнения одной и той же последовательности инструкций. При составлении программ следует позаботиться, чтобы цепочка рекурсивных вызовов не оказалась бесконечной, так как в противном случае произойдет аварийное окончание работы паскаль-машины.

4.11. Процедуры в качестве параметров

Пусть требуется проделать одни и те же действия со всеми узлами бинарного дерева, скажем напечатать или просуммировать хранящиеся в них значения. Одно из возможных решений состоит в том, чтобы записать несколько процедур, аналогичных процедуре РЕКУРСИВНЫЙ-ОБХОД-ДЕРЕВА. В каждой процедуре на месте инструкции

```
WRITE(КОРЕНЬ↑.ЗНАЧЕНИЕ)
```

следует поставить одну или несколько инструкций, реализующих нужные действия. Недостаток такого решения проявится, однако, если мы захотим во всех процедурах согласованным образом изменить порядок обхода — нам придется исправлять программу в нескольких местах. Если же воспользоваться имеющейся в языке паскаль возможностью передавать процедуры в качестве параметров, указанной трудности можно избежать.

Занишем фрагмент программы, в котором печатаются и суммируются значения, хранящиеся в узлах бинарного дерева (предполагается, что само дерево уже построено):

```
PROGRAM ПАРАМЕТРЫ(INPUT, OUTPUT);
```

...

```

TYPE АДРЕС-УЗЛА=↑УЗЕЛ;
УЗЕЛ=RECORD
    ЗНАЧЕНИЕ: INTEGER;
    ЛЕВЫЙ, ПРАВЫЙ: АДРЕС-УЗЛА
END;
VAR СУММА: INTEGER;
    КОРЕНЬ: АДРЕС-УЗЛА;
PROCEDURE ПЕЧАТЬ-ЗНАЧЕНИЯ(АДРЕС:АДРЕС-УЗЛА);
BEGIN
    WRITE(АДРЕС↑.ЗНАЧЕНИЕ)
END;
PROCEDURE ПРИБАВИТЬ-ЗНАЧЕНИЕ(АДРЕС:
    АДРЕС-УЗЛА);
BEGIN
    * СУММА:=СУММА+АДРЕС↑.ЗНАЧЕНИЕ
END;
PROCEDURE ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА(
    КОРЕНЬ: АДРЕС-УЗЛА;
    PROCEDURE ДЕЙСТВИЕ(АДРЕС:АДРЕС-УЗЛА));
BEGIN
    IF КОРЕНЬ <> NIL THEN BEGIN
        ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА(КОРЕНЬ↑.ЛЕВЫЙ,
            ДЕЙСТВИЕ);
        ДЕЙСТВИЕ(КОРЕНЬ);
        ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА(КОРЕНЬ↑.ПРАВЫЙ,
            ДЕЙСТВИЕ)
    END
END;
BEGIN (* НАЧАЛО ГЛАВНОЙ ПРОГРАММЫ *)
    ***
    (* ПРЕДПОЛАГАЕТСЯ, ЧТО В ЭТОМ МЕСТЕ СТОЯТ
        ИНСТРУКЦИИ ДЛЯ СОЗДАНИЯ БИНАРНОГО ДЕРЕВА *)
    ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА(КОРЕНЬ,
        ПЕЧАТЬ-ЗНАЧЕНИЙ);
    (* В РЕЗУЛЬТАТЕ ЭТОГО ОБРАЩЕНИЯ БУДУТ
        НАПЕЧАТАНЫ ЗНАЧЕНИЯ, ХРАНЯЩИЕСЯ В УЗЛАХ
        ДЕРЕВА *)
    WRITELN;
    СУММА:=0;
    ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА(КОРЕНЬ,
        ПРИБАВИТЬ-ЗНАЧЕНИЕ);
    WRITELN(' СУММА ЗНАЧЕНИЙ В УЗЛАХ ДЕРЕВА',
        ' РАВНА', СУММА)
END.

```

У процедуры **ОБОБЩЕННЫЙ-ОБХОД-ДЕРЕВА** два параметра: адрес корневого узла дерева и процедура, обрабатывающая узел. Как именно обрабатывается узел, для обхода дерева безразлично. Его дело — обратиться к этой процедуре в инструкции

ДЕЙСТВИЕ (КОРЕНЬ)

и передать процедуру рекурсивно для обработки поддеревьев.

Мы видим, что правила записи процедуры-формального параметра такие же, как у заголовка процедуры. Обращение к подобной процедуре записывается обычным образом, а передача в качестве фактического параметра сходна с передачей параметров других типов.

Функции, как и другие процедуры, можно передавать как параметры. Естественными примерами, где такая возможность полезна, являются вычисление определенных интегралов, решение уравнений, поиск экстремумов. Напишем функцию, отыскивающую решение уравнения $F(X)=0$ на заданном отрезке (A,B) ; функция действует методом деления пополам в предположении, что $F(A) \leq 0$, а $F(B) \geq 0$:

```
PROGRAM РЕШЕНИЕ-УРАВНЕНИЙ (OUTPUT);
FUNCTION КВАДРАТИЧНАЯ (X: REAL): REAL;
BEGIN
    КВАДРАТИЧНАЯ := X*X-2
END;
FUNCTION КУБИЧНАЯ (X: REAL): REAL;
BEGIN
    КУБИЧНАЯ := X*X*X-2
END;
FUNCTION НУЛЬ-ФУНКЦИИ (A,B: REAL;
    FUNCTION F (X: REAL): REAL;
    ПОГРЕШНОСТЬ: REAL): REAL;
BEGIN
    IF (F(A)>0) OR (F(B)<0) THEN BEGIN
        WRITELN (' *** СООБЩАЕТ НУЛЬ-ФУНКЦИИ');
        WRITELN (' НЕПРАВИЛЬНО ВЫБРАНЫ КОНЦЫ' ,
            ' ОТРЕЗКА');
        WRITELN (' A=',A,' B=',B);
        WRITELN (' F(A)=',F(A),' F(B)=',F(B));
        END
    ELSE BEGIN
        WHILE ABS(B-A)>ПОГРЕШНОСТЬ DO
            IF F((A+B)/2)<0
                THEN A:=(A+B)/2 ELSE B:=(A+B)/2;
        НУЛЬ-ФУНКЦИИ:=A;
```



```

END; (* НУЛЬ-ФУНКЦИИ *)
BEGIN (* ГЛАВНАЯ ПРОГРАММА *)
  WRITELN(' РЕШЕНИЕ УРАВНЕНИЯ  $X^2 - 2 = 0$ ');
  WRITE(' НА ОТРЕЗКЕ (0,2) С ТОЧНОСТЬЮ 0.0001 ЕСТЬ
  WRITELN(НУЛЬ-ФУНКЦИИ(0,2,КВАДРАТИЧНАЯ,
  0.0001):10:4);
  (* ДЛЯ ПЕЧАТИ РЕШЕНИЯ УРАВНЕНИЯ ОТВЕЛИ
  10 ПОЗИЦИЙ
  В МАНТИССЕ БУДУТ НАПЕЧАТАНЫ
  ЧЕТЫРЕ ЦИФРЫ *)
  WRITELN(' РЕШЕНИЕ УРАВНЕНИЯ  $X^3 - 2 = 0$ ');
  WRITE(' НА ОТРЕЗКЕ (0,2) С ТОЧНОСТЬЮ 0.0001 ЕСТЬ
  WRITELN(НУЛЬ-ФУНКЦИИ(0,2,КУБИЧНАЯ,0.0001):10:4);
END.

```

Программа напечатает:

```

РЕШЕНИЕ УРАВНЕНИЯ  $X^2 - 2 = 0$ 
НА ОТРЕЗКЕ (0,2) С ТОЧНОСТЬЮ 0.0001 ЕСТЬ   1.4142
РЕШЕНИЕ УРАВНЕНИЯ  $X^3 - 2 = 0$ 
НА ОТРЕЗКЕ (0,2) С ТОЧНОСТЬЮ 0.0001 ЕСТЬ   1.2599

```

З а м е ч а н и е. Стандартные процедуры (функции) не могут выступать в качестве фактических параметров.

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

В формулировках задач в фигурных скобках указывается сложность задачи в условных единицах. Оценка сложности 0 означает, что для решения этой задачи нужно составить программу примерно из 25 строк по 30 литер в строке. Увеличение сложности на 10 означает примерно удвоение требуемых усилий. Сложная задача не обязательно требует составления длинной программы. Буква И перед оценкой сложности означает, что задача исследовательская, т. е. для ее решения недостаточно составить одну программу, а понадобится составить несколько программ и поэкспериментировать с ними.

В процессе решения задачи целесообразно составлять отчет, который должен включать:

- 1) формулировку задачи;
- 2) описание способа задания входных данных, формы выходных данных и спецификацию программы (т. е. описание того, каким будет результат программы при заданных входных данных);
- 3) словесное описание алгоритма решения;
- 4) оценку времени работы программы;
- 5) распечатку программы и результата ее работы;
- 6) результаты дополнительных исследований, если они предусмотрены условием задачи.

В некоторых задачах нужен датчик случайных чисел. Подробную информацию о том, как получать на ЭВМ случайные числа, можно найти в книгах [6, п. 4.5; 12, т. 2, п. 3.2].

Переборные задачи

П1. {4} Для заданных натуральных k, l, m, n ($k, l, m \geq 2$) найти все решения в натуральных числах уравнения $x^k + y^l = z^m$, удовлетворяющие условию $z^m \leq n$. Составить по возможности эффективную программу. Исследовать случай $k=2, l=3, m=5, n=10^{10}$. (В задаче M764 [14] утверждается, что это уравнению имеет бесконечно много решений в натуральных числах.)

П2. {13} Для какой-либо головоломки типа кубика Рубика имеющей $10^3 \dots 10^6$ состояний, найти наименьшее число ходов, которое любое состояние можно привести к начальному. Пример таких головоломок: сборка креста на одной грани кубика Рубика, игра 8, т. е. игра 15 на поле 3×3 . Ряд интересных головоломок описан в [17].

П3. {16} *Задача о рюкзаке*. Имеется несколько видов предметов. Каждый вид характеризуется массой одного предмета и его стоимостью (натуральные числа). Найти, какие предметы следует положить в рюкзак, чтобы общая масса не превышала заданной границы (порядка 1000), а общая стоимость была максимальной. Использовать метод динамического программирования.

П4. {12} Имеется n городов. Некоторые из них соединены дорогами известной длины. Для заданного города (столицы) найти кратчайшие маршруты в остальные города и их длины.

П5. Найти все натуральные числа, равные сумме k -х степеней своих цифр:

а) {5} $k=1..5$;

б) {15} $k=1..10$.

П6. {17} Дана последовательность цифр. Найти все способы расставить между ними знаки арифметических операций $+$, $-$, $*$, $:$, так, чтобы в результате выполнения этих операций получилось 100. Операции выполняются последовательно слева направо. Если между цифрами не стоит знака операции, то эти цифры объединяются в многозначное число. Исследовать последовательности цифр 123456789 и 987654321. Пример одного из решений: $100 = 1 * 2 * 3 - 4 * 5 - 6 + 7 + 89$.

Вариант {9}: то же самое, но многозначные числа не допускаются.

П7. {15} Напечатать все существенно различные разложения заданного натурального числа на заданное количество слагаемых. Разложения, отличающиеся только порядком слагаемых, существенно различными не считаются.

Сортировка

S1. {12} Запрограммировать какой-либо $O(N \cdot \log N)$ -метод сортировки массива. Сравнить фактические времена работы этого метода с $O(N^2)$ -методом для $N=2,3,4,5,10,100,1000$. В [12, т. 2, п. 5.2] можно познакомиться с теорией сортировки и различными алгоритмами, там же имеется анализ времени работы алгоритмов сортировки.

Игры

И1. Написать программу для решения шахматных задач методом полного перебора. Над этой программой должны работать не менее четырех человек.

Задача разбивается на следующие части, над каждой из которых может работать один человек (однако необходимо совместное обсуждение общих решений):

- а) {0} ввод и преобразование во внутреннее представление условия задачи;
- б) {3} построение таблиц возможных ходов на пустой доске;
- в) {20} нахождение возможных ходов в данной позиции;
- г) {20} управление перебором;
- д) {10} вывод на печать решения и промежуточной отладочной информации.

В [1] приводятся программы на алголе для решения шахматных задач (двух-, трех- и четырехходовок) и даются рекомендации, полезные при самостоятельном составлении подобных программ.

И2. {20} Провести полный анализ какой-либо конечной игры, содержащей от 10^3 до 10^6 позиций, например «Волки и овцы» на обычной шахматной доске.

И3. {13} Написать программу, моделирующую игру «Жизнь» и печатающую состояния. Предусмотреть возможность печати не всех состояний. Обеспечить возможно более компактный способ задания начальной позиции [16, гл. 2].

Системное программирование

С1. Реализовать язык программирования, позволяющий выполнять вычисления в символьном виде. Объектами, с которыми работает программа, являются формулы, включающие переменные, а также целые и рациональные числа произвольного размера. Набор операций включает арифметические операции и дифференцирование. Можно запрограммировать и другие действия над формулами, например вычисление предела.

Эта задача разбивается на следующие части, каждая из которых может быть запрограммирована одним программистом независимо от остальных.

- а) {10} Лексический анализ — ввод чисел, идентификаторов, знаков операций.
- б) {20} Синтаксический анализ программ и построение внутреннего представления программ.
- в) {10} Интерпретация внутреннего представления.
- г) {35} Подпрограммы для выполнения содержательных действий над формулами и числами. Этот пункт можно разделить между несколькими программистами.
- д) {20} Упрощение формул.
- е) {10} Вывод на печать формул и чисел.

О методах лексического и синтаксического анализа см. [7].

С2. {20} Не используя стандартную процедуру WRITE для вещественных чисел, перевести заданное вещественное число в де-

сятичную систему. Обеспечить как можно более высокую точность перевода. Исследовать точность программы. Алгоритмы перевода чисел рассмотрены в [12, т. 2, п. 4.4].

С3. {10} Вычислить результат арифметического выражения заданного в обратной польской записи и состоящего из целых чисел и знаков операций $+$, $-$, $*$, $/$.

С4. {13} Преобразовать обычную запись арифметического выражения, состоящего из однобуквенных идентификаторов, знаков арифметических операций и скобок, в польскую запись. В польской записи знаки операций записываются после операндов, скобки не используются. Более подробно см., например, [7, § 11.2].

Машинная графика и геометрия

Г1. Написать программу, которая будет печатать

а) {8} линии уровня функции $z=f(x,y)$;

б) {12} линию, заданную в параметрическом виде $x=f(t)$
 $y=g(t)$.

Кроме графика выдать на печать вспомогательную информацию — оси координат, обозначения на осях, диапазоны изменения переменных и т. д. Подобную программу см. в [8, программа 6.2].

Г2. {20} Дано четное число точек на плоскости, никакие три из которых не лежат на одной прямой. Медианой этого множества точек называется прямая, соединяющая две точки множества такая, что с обеих сторон от нее лежит равное число точек. Подсчитать число медиан для заданного множества точек. Порождая точки случайным образом, вычислить распределение числа медиан, а также среднее и максимальное число медиан.

Г3. {10} Имеется n жучков на плоскости (даны их координаты). Каждый из них с постоянной скоростью, равной 1, ползет к следующему (n -й ползет к 1-му). Если расстояние между жучками становится не больше 1, преследуемый жучок считается съеденным. Напечатать и (или) нарисовать эволюцию системы.

Арифметика многократной точности

Алгоритмы, полезные при решении задач А1 и А2, приведены в [12, т. 2, п. 4.3].

А1. Реализовать арифметические действия над целыми числами произвольной величины.

а) {12} Вычислить $2000!$, F_{5000} (число Фибоначчи), $3^{20\,000}$.

б) {125} Разложить на простые сомножители несколько случайно выбранных 20-значных чисел (см. также [12, т. 2, п. 4.5.4]).

А2. Реализовать арифметические действия над вещественными числами произвольной точности.

а) {15} Вычислить с помощью разложений в ряд $\sqrt{2}$, e , π , $\ln 2$ с 2000 знаками после запятой. Все эти константы должны вычисляться почти одной и той же программой.

б) {25} Вычислить $\lg 2$ с 2000 знаков после запятой.

A3. {10} Найти два 1000-значных натуральных числа, последние 1000 цифр квадрата которых совпадают с самими числами.

A4. {11} Натуральное число называется палиндромом, если оно читается одинаково с обеих сторон, например 171. Возьмем любое число. Если оно не палиндром, то перевернем его и сложим с исходным числом. Если получился не палиндром, то сделаем с ним то же и т. д., пока не получится палиндром. Для всех чисел от 1 до 1000 найти, через сколько шагов получится палиндром и какой.

Теория чисел

Ч1. {12} *Обратная теорема Ферма.* Малая теорема Ферма утверждает, что для всякого натурального a , если число p простое, то a^p сравнимо с a по модулю p . Для $a=2,3,5,7$ найти все числа p , $1 < p \leq 100\,000$, не удовлетворяющие обратному утверждению.

Ч2. Проверить, существуют ли нечетные совершенные числа, меньшие чем

а) {10} 200 000;

б) {15} 10^6 ;

в) {140} 10^{12} .

Обратить внимание на обоснование правильности программы.

Работа с текстами

T1. {22} Дан текст на русском языке длиной около 5000 слов. Напечатать 100 наиболее часто встречающихся слов и число их появлений. Для хранения слов использовать массив или список, упорядоченный по частоте слов. Измерить среднюю длину операции поиска. Оценить, дает ли преимущество использование бинарного дерева, как в программе из [8, п. 6.2, 8.4].

T2. {10} Написать интерпретатор машины Поста. Предусмотреть два режима: печать состояния после каждого шага и печать только результата.

T3. {20} Имеется словарь примерно из 100 слов. Вводятся слова, в которых могут быть допущены следующие ошибки:

а) переставлены две соседние буквы;

б) искажена одна буква;

в) пропущена одна буква;

г) вставлена лишняя буква.

Найти в словаре все слова, из которых могло получиться введенное слово в результате не более чем одной ошибки.

T4. {10} *Спиральное кодирование.* Чтобы закодировать текст, записывают его в матрицу $n \times n$ (по строкам), где n нечетно, а затем прочитывают по спирали, начиная с центра. Уметь кодировать и декодировать сообщения.

T5. {5} Написать программу, печатающую все способы переноса данного слова, отвечающие правилам орфографии.

Вычислительные методы

B1. {20} *Задача трех тел.* Составить программу для расчета движения в плоскости трех притягивающих материальных точек. Предусмотреть контроль погрешности расчетов. Рассчитать движение трех точек с массой соответственно 3, 4, 5, первоначально покоящихся в вершинах прямоугольного треугольника со сторонами 3, 4, 5 (каждая точка лежит напротив стороны, длина которой совпадает с массой точки). Известно, что примерно через 70 единиц времени эта система распадется на пару связанных точек и одну, удаляющуюся от них в бесконечность. Эта задача сводится к решению системы дифференциальных уравнений, о методах решения которых см. [13].

B2. {12} Исследовать погрешность нескольких алгоритмов обращения матриц [16, гл. 21].

B3. {25} Написать программу для нахождения вещественных решений алгебраических уравнений с вещественными коэффициентами. В отчете нужно обосновать, что программа находит все вещественные решения.

Разные задачи

P1. {8} Напечатать заданное целое число словами.

P2. {12} Напечатать календарь на заданный год. Обратит внимание на оформление выдачи.

P3. {30} Написать программу, печатающую свой собственный текст [16, гл. 9].

P4. {11} Расположив натуральные числа от 1 до 128^2 на плоскости по спирали, отметить звездочками простые числа. Напечатать картинку, содержащую звездочки и число 1 [16, гл. 15]. Можно использовать только один массив из 128^2 элементов. Простые числа следует вычислять по методу решета Эратосфена.

P5. {15} *Стохастические автоматы.* В каждой целочисленной точке прямой находится лампочка, которая может гореть или нет. Обозначим лампочку в точке k через $\Pi(k)$. Состояние лампочек в момент $t+1$ определяется по состоянию в момент t следующим образом. Если в момент t лампочки $\Pi(k)$ и $\Pi(k-1)$ горят, то в момент $t+1$ лампочка $\Pi(k)$ горит. Если же хотя бы одна из $\Pi(k)$ и $\Pi(k-1)$ не горит в момент t , то $\Pi(k)$ в момент $t+1$ горит с вероят-

постью θ и не горит с вероятностью $1 - \theta$, где θ — заданное число. В начальный момент все лампочки не горят. Вычислить функцию $P(\theta)$ — среднюю долю горящих лампочек через очень большое время. Вычислить число θ^* , такое что при θ , превышающих θ^* , $P(\theta) = 1$, а при $\theta < \theta^*$ $P(\theta) < 1$. Обратить внимание на оценку погрешности ответа. (Известно, что $0.09 < \theta^* < 0.328$.)

Р6. {5} Многочлены задаются в виде наборов пар (коэффициент, степень). Перемножить многочлены. И в данных, и в результате показатели степеней должны возрастать.

Р7. {5} Дан связный граф. Выделить из него дерево, содержащее все вершины графа.

ЗАКЛЮЧЕНИЕ

Богат и многообразен мир программирования для электронных вычислительных машин. ЭВМ все глубже проникают в науку, производство, повседневную жизнь. Каковы же основные тенденции в развитии ЭВМ и важнейшие области их применения?

В развитии электронно-вычислительной техники нам представляются главными следующие четыре линии.

1. Супер-ЭВМ (с быстродействием свыше 100 млн. операций в секунду).

2. Микропроцессоры — миниатюрные блоки для построения вычислительных машин. На кремниевой пластинке со стороной около 1 см располагается несколько тысяч электронных компонентов.

3. Персональные ЭВМ (ПЭВМ). Благодаря успехам микропроцессорной технологии стало возможным изготавливать довольно дешевые ЭВМ, уместающиеся на рабочем столе, но отнюдь не являющиеся игрушками — их быстродействие 10—100 тыс. операций в секунду. Персональные ЭВМ — действительно массовые машины. По данным на конец 1985 г. в школах Франции было установлено около 120 тыс. ПЭВМ, а в домах американцев — около 13 млн. ПЭВМ.

4. Сети ЭВМ. Когда ЭВМ соединены друг с другом линиями связи, они фактически приобретают новые качества. Прежде всего это относится к персональным ЭВМ, владельцы которых получают доступ к огромным массивам информации и сложным программам, хранящимся в памяти супер-ЭВМ.

Успехи микропроцессорной технологии существенно повлияли на характер общения человека и ЭВМ. Хотя сейчас, пожалуй, самым распространенным устройством ввода информации является клавиатура терминала, все большее значение приобретают устройства речевого ввода и графические планшеты. Типичные устройства вывода информации — графопостроитель, дисплей, звукосинтезатор, устройство синтеза речи.

Основными, на наш взгляд, являются следующие области применения вычислительных машин.

1. *Машинная графика.* Телевидение помогает нам увидеть дальние страны. Машинная графика способна сделать зримыми проектируемую дорогу или жилой район, тренировочный полет на самолете или в космическом корабле, лицо человека после пластической операции, взаимодействие молекул и многое другое. ЭВМ, снабженная цветным графическим дисплеем и графическим программным обеспечением, — это окно в сказочный мир, богатство которого определяется богатством нашей фантазии и программистским мастерством. Без машинной графики человек бредет по миру программ как бы в темноте, догадываясь о происходящих событиях только по косвенной, числовой информации. Видимо, не будет преувеличением сравнить развитие машинной графики с восходом солнца над программистским миром.

2. *Системы автоматизации проектирования.* Все более сложными становятся проектируемые машины, все больше конструкторского труда, поиска нужно на их создание. С другой стороны, все быстрее происходит моральное старение техники, значит, требуется быстрее создавать проекты. Одно из основных средств преодоления указанного противоречия — внедрение систем автоматизации проектирования (САПР). САПР — это сложные программные комплексы, которые в значительной степени избавляют конструктора от рутинной работы. В подобных системах применяются хорошо зарекомендовавшие себя способы поиска удачных конструкторских решений, что ускоряет разработку и повышает ее качество. Системы автоматизации проектирования позволяют (с помощью машинной графики) увидеть создаваемую машину, проанализировать ее функционирование в разных режимах, менять параметры машины, если что-то в ее работе конструктора не устраивает.

3. *Управление роботами.* Современные роботы — это сложнейшие механизмы, падающие осязанием, зрением, манипуляторами, положение которых контролируется с точностью до долей миллиметра, а также средствами передвижения. И задачи на роботов возлагаются все более сложные — деятельность в составе гибких (перестраиваемых) производств, работа под водой, на поверхности других планет, в обстановке, полностью «рассчитать» которую заранее невозможно. Управление роботами — передний край современного программирования. Много вбирает в себя эта область — представление знаний в ЭВМ, механизмы получения нового знания, распознавание речи и зрительных образов, решение задач оптимального управления. Внедрение роботов в сочетании с развитием систем автоматизации проектирования в принципе открывает путь к сквозной автоматизации — от конструкторского замысла до готового изделия.

Программирование роботов — одна из важнейших ветвей научного направления, получившего название «искусственный интеллект».

4. *Экспертные системы* — другая стремительно развивающаяся ветвь исследований по искусственному интеллекту. Экспертная система — это программа, способная решать трудные, по общему мнению, задачи, требующие специальных знаний в какой-либо области. Экспертная система, диагностирующая бактериальные инфекции, способна помочь врачу; система, определяющая способы борьбы с заболеваниями растений, — отличное подспорье и для профессионального агронома, и для садовода-любителя. Экспертные системы дадут хороший совет космонавту и химику, комбайнеру и юристу. В подобных системах аккумулируются знания ведущих специалистов. Это могут быть и общие правила, и примеры действий в определенных обстоятельствах. Экспертные системы наделяются также средствами для самостоятельного вывода новых правил на основе имеющейся информации. Обычно такая система, как врач в больнице, задает человеку вопросы, на основании его ответов приходит к определенному заключению и выдает свои рекомендации. В свою очередь человек может узнать, почему экспертная система сделала тот или иной вывод, и, если ему покажется, что система на ложном пути, он может направить ее поиск по другому руслу.

5. *Математическое моделирование, вычислительный эксперимент*. Если в начале своего развития ЭВМ в соответствии с названием занимались в основном вычислениями, то теперь эта область применения — далеко не самая массовая. Однако важность численных расчетов нисколько не уменьшилась. С помощью ЭВМ происходит управление движением космических аппаратов, рассчитываются характеристики будущих самолетов, предсказывается погода. Особую роль играет вычислительный эксперимент. Если известны законы развития какого-либо процесса и эти законы выражены в математической форме, т. е. если создана математическая модель, можно написать программу, разворачивающую исследуемый процесс во времени при различных начальных условиях, можно экспериментировать с процессом. С помощью математической модели и вычислительного эксперимента иногда удается открывать новые, ранее неизвестные явления. Вычислительный эксперимент во много раз дешевле обычного эксперимента, и выполнить его намного проще. Например, строительство экспериментальной термоядерной установки обходится в миллионы рублей, длится годами, а характерное время протекания исследуемых явлений — доли секунды. ЭВМ позволяет «остановить мгновение» — растянуть моделируемый процесс во времени, чтобы разобраться в тонкостях его протекания. Иногда, например при проверке космологических гипотез

вычислительный эксперимент — вообще единственное доступное средство.

6. *Обработка текстов* — одна из самых популярных областей использования ЭВМ. Программы, называемые редакторами текстов, помогают создавать и исправлять любой текст. Совокупность программных и аппаратных средств служит для облегчения подготовки публикаций, они берут на себя многие стандартные операции — нумеруют страницы, выравнивают текст по правому краю, переносят слова, исправляют орфографические ошибки. Общаться с подобной системой особенно просто и удобно, если она дополнена средствами для распознавания речи. Подготовленный на ЭВМ текст может быть направлен в типографию, где фотонаборные автоматы отпечатают конечный продукт, например газету или книгу. Системы обработки текстов — одно из важнейших средств повышения производительности конторского труда, перехода к безбумажному делопроизводству, что сулит огромную экономию и людских, и материальных ресурсов.

7. *Электронные бланки* — еще одно средство компьютеризации конторского труда. Человек, их использующий, видит на экране терминала привычную картину — бланк, разделенный на клетки. Он вписывает в клетки нужную информацию, однако клетки эти «волшебные». В них могут располагаться не только числа и тексты, но и формулы, что позволяет легко реализовать бухгалтерские расчеты. Стоит изменить число в какой-нибудь клетке, и произойдет автоматический перерасчет всего бланка, что гарантирует его непротиворечивость. Электронные бланки — пример реализации очень важной программистской идеи: работать по принципу «что вижу, то имею».

8. *База данных* — это целостная совокупность данных, относящихся к какой-либо области. Это могут быть сведения о промышленном изделии или целом заводе, электронная библиотека или справочная система. Большие базы данных вмещают миллиарды единиц информации. С помощью информационно-поисковых систем можно найти в этом море сведений то, что нас интересует, например все журнальные публикации по какой-либо теме за определенный период или данные о людях, проработавших на предприятии более 20 лет. Базы данных — неотъемлемая часть систем автоматизации проектирования, экспертных систем, электронных бланков.

9. *Персональные ЭВМ, ЭВМ в быту*. ПЭВМ, входящие в сеть вычислительных машин, способны во многом изменить уклад домашней жизни. Прежде всего это касается информационного обеспечения — библиотечная и справочная информация, хранящаяся в базах данных на больших ЭВМ, будет всегда «под рукой». Доступными будут и сложные программы, такие как экспертные системы

или программы для вычислительных экспериментов. Более простые программы — средства машинной графики, системы обработки текстов, электронные бланки — могут функционировать на самой ПЭВМ.

Большую помощь ПЭВМ могут оказать в обучении, прежде всего потому, что способны предоставить модель любого изучаемого явления, модель, с которой можно активно работать, изменяя ее параметры, рассматривая с разных точек зрения. На экране цветного дисплея можно наблюдать самые эффектные химические реакции, не боясь отравиться или взорваться. Доступными, зримыми становятся молекулы белка и далекие планеты, решения дифференциальных уравнений и законы оптики.

ПЭВМ способны помочь и в повседневных домашних делах. С их помощью можно оформить заказ на покупку или забронировать билет на самолет. ПЭВМ могут взять на себя управление бытовой техникой — вовремя разбудить нас, включить телевизор, проследить за работой кухонной плиты. Микрокомпьютер, встроенный в фотоаппарат, поможет даже начинающему фотографу делать качественные снимки.

Многое, очень многое могут ЭВМ. Но при том условии, что найдутся люди — программисты, — способные научить машины рисовать, говорить, управлять роботами, давать советы. Авторы будут рады, если книга поможет кому-то из читателей приобщиться к миру программирования для электронных вычислительных машин.

Приложение 1

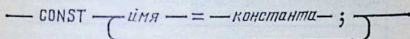
СИНТАКСИЧЕСКИЕ ДИАГРАММЫ ЯЗЫКА ПАСКАЛЬ

Синтаксические диаграммы описывают правила записи программ на языке паскаль. Каждая диаграмма задает структуру той или иной конструкции языка. Диаграмма имеет номер и название и состоит из надписей, которые соединены между собой линиями. Среди линий диаграммы выделяются вход и выход. Вход расположен слева, под названием, выход — справа внизу. Каждый путь на диаграмме от входа к выходу соответствует одному из допустимых вариантов данной конструкции языка. Допускается только движение по прямой или плавные повороты. Запрещено проходить острые углы, а также одну и ту же линию в двух направлениях.

Если на пути движения встречается надпись, состоящая из больших букв, цифр или знаков, то она включается в программу в том виде, как она записана на диаграмме. Надписи, сделанные малыми буквами — это названия других диаграмм из приложения 1. Вместо таких надписей следует подставлять конкретные тексты, структура которых задается соответствующими диаграммами.

Рассмотрим, например, диаграмму

4. определение констант



Согласно этой диаграмме определение констант должно начинаться словом CONST. Затем вместо надписи «Имя» нужно подставить конкретное имя, например ПИ. Структура имен описана на диаграмме 31. После имени пишется знак «=». Вместо надписи «константа» нужно подставить определенную константу (диаграмма 27), например 3.14159. Затем, поставив «;», можно вернуться по нижней линии, начав описание следующей константы. Повторив этот цикл произвольное число раз, можно направиться к выходу, закончив на этом определение констант.

Отметим, что при помощи диаграмм трудно отобразить все правила записи программ на языке паскаль. Из диаграмм не следует, например, что нельзя употреблять имена, совпадающие по написанию с синтаксическими словами, нельзя использовать в инструкциях имена, которые не были описаны выше и т. д. Не от-

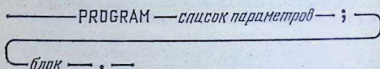
ражены также правила употребления пробелов и комментариев. Эти правила сводятся к следующему.

1. Нельзя ставить пробелы внутри синтаксических слов, имен, числовых констант, символов $:=$, $<=$, $>=$, $<$, $>$, ...

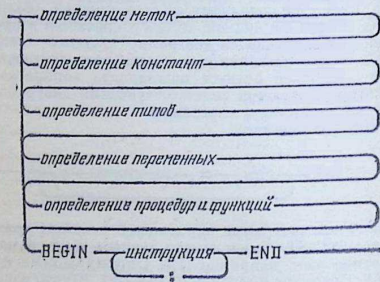
2. Пробел обязателен между подряд стоящими синтаксическими словами, именами, числами в любых сочетаниях; в этих случаях пробел служит разделителем. Внутри текстовой константы пробел воспринимается не как разделитель, а как часть константы. В остальных местах пробелы можно употреблять произвольно. Везде, где может стоять пробел (кроме текстовых констант), разрешены также последовательность пробелов, переход на новую строку, комментарий, а также любое их сочетание.

Несколько замечаний к диаграммам. Диаграмма 31 соответствует нескольким понятиям языка паскаль. В диаграммах 27, 28 встречается надпись «литера не кавычка», для которой нет диаграммы в приложении 1. Имеется в виду произвольная литера, допустимая в данной реализации паскаль-машины, кроме одиночной кавычки. В диаграмме 34 «литера» — это произвольная литера, допустимая в данной реализации паскаль-машины. В некоторых реализациях (диаграмма 33) русские буквы не допускаются.

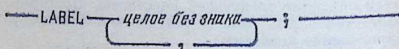
1. программа



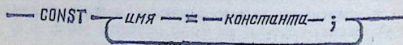
2. блок



3. определение меток



4. определение констант



5. определение типов

TYPE — имя — = — тип — ;

6. определение переменных

VAR — имя : — тип ;

7. определение процедур и функций

заголовок процедуры ; блок
заголовок функции FORWARD ;

8. заголовок процедуры

PROCEDURE — имя — список параметров —

9. заголовок функции

FUNCTION — имя — список параметров — : — имя —

10. список параметров

(— имя : — имя)
VAR
заголовок функции
заголовок процедуры

11. инструкция

целое без знака :
инструкция присваивания
вызов процедуры
условная инструкция
инструкция выбора
инструкция цикла
инструкция with
инструкция перехода
BEGIN — инструкция — END
;

12. инструкция присваивания

— *имя переменной* — := — *выражение* —

13. вызов процедуры

— имя процедуры — (выражение) —

14. условная инструкция

— IF — *выражение* — THEN — *инструкция* —
— ELSE — *инструкция* —

15. инструкция выбора

— CASE — выражение — DF —
 { константа : инструкция } END —
 { , : }

16. инструкция цикла

WHILE — выражение — DO — инструкция
REPEAT — инструкция — UNTIL — выражение
FOR — имя переменной — := — выражение
TO — выражение — DO — инструкция
DOWNTO

17. инструкция with

— WITH — переменная — DO — инструкция —

18. инструкция перехода

— ГOTO — целое без знака —

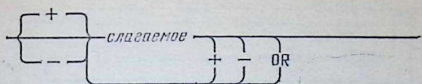
19. выражение

простое выражение

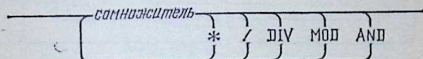
= < > <= >= > IN

простое выражение

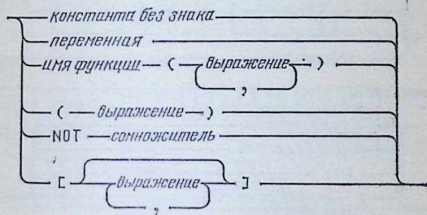
20. простое выражение



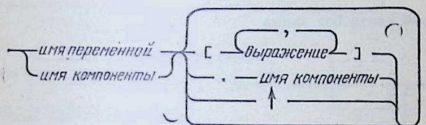
21. слагаемое



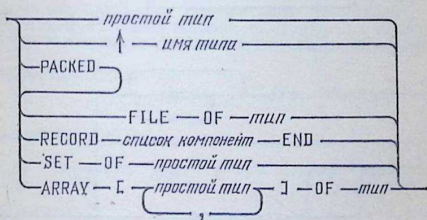
22. множитель



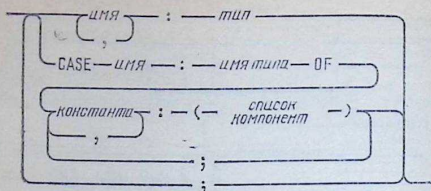
23. переменная



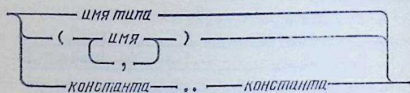
24. тип



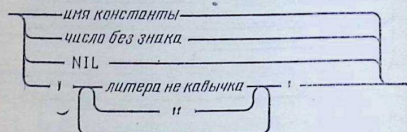
25. список компонент



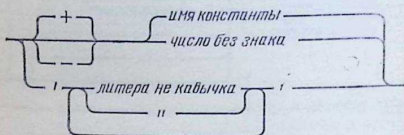
26. простой тип



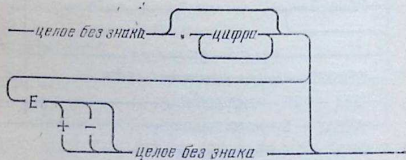
27. константа



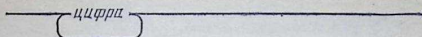
28. константа без знака



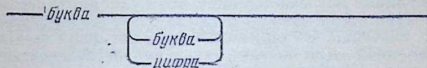
29. число без знака



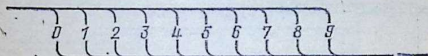
30. целое без знака



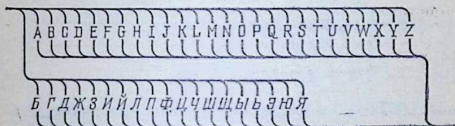
31. имя, имя константы, имя типа, имя переменной, имя процедуры, имя функции, имя компоненты



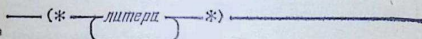
32. цифра



33. буква



34. комментарий



Приложение 2

ПРИМЕР ПРОГРАММЫ

Программа СЧЕТ предназначена для вычисления функций от одной переменной, заданных аналитически (формулой), при определенных значениях этой переменной. Программа читает данные из файла INPUT. Описание функции имеет вид:

$F(X) = \langle \text{формула} \rangle;$

например:

$F(X) = \text{SQRT}(2 * X / 9.8);$

Значение переменной задается при помощи формулы, не содержащей X:

$X = \langle \text{формула} \rangle;$

Примеры.

$X=0$;

$X=2.15$;

$X=(1+\text{SQRT}(5))/2$;

В формулах допускаются арифметические операции (+, -, *, /, ↑ (возведение в целую степень)), круглые скобки, стандартные функции (SIN, COS, LN, EXP, ABS, SQRT). Аргументы стандартных функций заключаются в скобки.

Если в данных встречается описание функции, то оно запоминается. Если функция не зависит от X, то сразу же вычисляется и печатается ее значение. За описанием функции могут следовать значения переменной. Для каждого значения переменной вычисляется и выводится значение функции. Данные могут содержать несколько описателей функций.

Пример данных для программы СЧЕТ.

$F(X) = ((2 * X - 3) * X + 0.5) * X - 10 * X + 4$;

$X=1$;

$X=0$;

$X=-1.5$;

$F(X) = \text{SQRT}(5.0 - X \uparrow 2)$;

$X=2$;

$X=5$;

$F(X) = \text{SIN}(2X) / \text{COS}(2X)$;

$X=0.5$;

$X=3.14$;

$F(X) = 1 + 1/2 + 1/3 + 1/4 + 1/5$;

В файл OUTPUT в результате работы программы СЧЕТ будет выведен текст:

$F(X) = ((2 * X - 3) * X + 0.5) * X - 10 * X + 4$;

$X=1$;

$F(1.000E+00) = 6.500E+00$

$X=0$;

$F(0.000E+00) = 4.000E+00$

$X=-1.5$;

$F(-1.500E+00) = 4.038E+01$

$F(X) = \text{SQRT}(5.0 - X \uparrow 2)$;

$X=2$;

$F(2.000E+00) = 1.000E+00$

$X=5$;

***SQRT(-2.000E+01)

$F(X) = \text{SIN}(2X)$

***СИНТАКСИЧЕСКАЯ ОШИБКА В ДАННЫХ.

ОБНАРУЖЕНА В ПРОЦЕДУРЕ

ЭЛЕМЕНТАРНАЯ-ФОРМУЛА

НЕТ РАЗДЕЛИТЕЛЯ МЕЖДУ ОПЕРАНДАМИ

$X=0.5$;

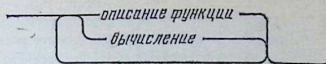
$X=3.14$;

$F(X) = 1 + 1/2 + 1/3 + 1/4 + 1/5$;

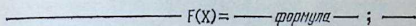
$F(X) = 2.283E+00$

Данные для программы СЧЕТ должны обладать определенной структурой, которую можно описать при помощи синтаксических диаграмм.

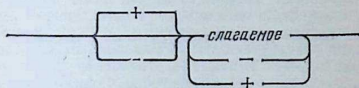
1. данные



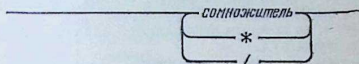
2. описание функции



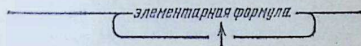
3. формула



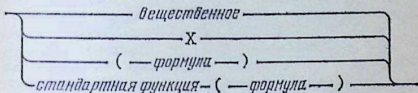
4. слагаемое



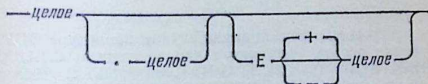
5. сумножитель



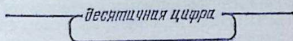
6. элементарная формула



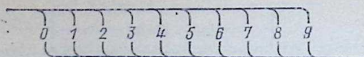
7. вещественное



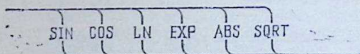
8. целое



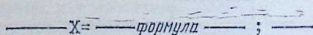
9. десятичная цифра



10. стандартная функция



11. вычисление

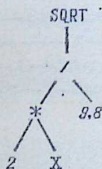


Кроме того, данные должны удовлетворять следующим требованиям:

- каждая инструкция начинается с новой строки;
- пробелы и переходы на новую строку допускаются везде, кроме чисел и имен элементарных функций;
- формулы, по которым вычисляются значения переменной X, не должны содержать X.

Идея программы состоит в следующем. Каждая формула, встретившаяся в данных, преобразуется в дерево и в таком виде хранится в памяти паскаль-машины. Концевые узлы дерева (листья) представляют числовые константы и вхождения переменной X, имеющиеся в формуле. Остальные узлы представляют арифметические операции или стандартные функции. Поддеревья этих узлов представляют соответственно операнды арифметических операций или аргументы функций.

Пример. Дерево формулы $SQRT(2 * X / 9.8)$



Для построения дерева формулы служит процедура ФОРМУЛА. В построении дерева участвуют также процедуры СЛАГАЕМОЕ, СОМНОЖИТЕЛЬ, ЭЛЕМЕНТАРНАЯ-ФОРМУЛА. Используются еще процедуры ВЕЩЕСТВЕННОЕ и ЦЕЛОЕ для чтения констант, процедура СТАНДАРТНАЯ-ФУНКЦИЯ для чтения имени стандартной функции, процедуры ПРОБЕЛЫ и ДАЛЬШЕ для чтения файла INPUT до очередной литеры, отличной от пробела. К процедуре ФОРМУЛА обращаются процедуры ОПИСАНИЕ-ФУНКЦИИ и ВЫЧИСЛЕНИЕ, к которым в свою очередь обращается главная программа.

Перечисленные выше процедуры (кроме ПРОБЕЛЫ и ДАЛЬШЕ) сопоставляют текст, читаемый из файла INPUT с соответствующей синтаксической диаграммой (главная программа соответствует диаграмме «данные»), и выполняют необходимые действия, связанные с построением дерева или вычислением функций. Считается, что при обращении к любой из этих процедур буферная переменная INPUT[†] содержит первую литеру той конструкции данных, для которой предназначена процедура. Если в данных обнаруживается ошибка, то происходит обращение к процедуре ОШБ, которая анализирует причину ошибки, выдает диагностическое сообщение и прочитывает файл INPUT до ближайшей литеры «;».

Для вычисления значения функции, представленной в виде дерева, при заданном X служит процедура ЗНАЧ. Она осуществляет рекурсивный обход дерева (в порядке левое поддерево, правое поддерево, корень), выполняя при этом стандартные функции и операции. Таким образом, в первую очередь выполняются операции и функции, расположенные на нижних уровнях дерева.

Для того чтобы обеспечить привычный порядок действий при вычислении, необходимо позаботиться о соответствующей структуре дерева при его построении. В корне дерева должна находиться та операция или функция, которая выполняется в последнюю очередь. Это же правило относится и к любому поддереву.

Припишем функциям и операциям приоритеты:

- 1 — сложение и вычитание,
- 2 — умножение и деление,
- 3 — возведение в степень,
- 4 — стандартные функции.

Тогда порядок действия можно определить правилами.

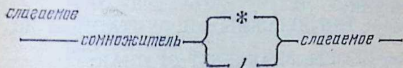
1. Действия в скобках выполняются раньше, чем действия вне скобок.

На одном уровне скобок

2. Действия с большим приоритетом выполняются раньше действий с меньшим приоритетом.

3. Действия с одинаковым приоритетом выполняются слева направо.

Соблюдение первых двух правил обеспечивается структурой совокупности диаграмм (и соответствующих процедур) ФОРМУЛА, СЛАГАЕМОЕ, СОМНОЖИТЕЛЬ, ЭЛЕМЕНТАРНАЯ-ФОРМУЛА. Соблюдение третьего правила обеспечивается внутренней структурой диаграмм «формула», «слагаемое», «сомножитель» и способом построения дерева в соответствующих процедурах. Так, при разработке программы пришлось отказаться от более простых диаграмм, например,



которые, хотя и правильно отражают синтаксис формул, но не обеспечивают соблюдение третьего правила.

В программе СЧЕТ используется возможность предварительного описания процедур (описание FORWARD). В данном случае оно облегчает составление программы методом пошаговой детализации. Скажем, при написании процедуры для обработки описания функции

$F(X) = \langle \text{формула} \rangle;$

ясно, что нужно сначала проверить, правильно ли записаны литеры «F(X)=», затем проанализировать формулу и построить для нее дерево и, наконец, проверить, есть ли после формулы точка с запятой. Таким образом, хотелось бы воспользоваться процедурой ФОРМУЛА, не приводя пока ее полного описания. В языке паскаль использование процедур и функций до описания их заголовков запрещено. При помощи описания FORWARD можно, не нарушая этого запрета, располагать сами процедуры в том порядке, который удобен при их написании и облегчает понимание программы в целом. Следует обратить внимание на то, что, если имело место предварительное описание процедуры, то при ее окончательной записи нельзя задавать параметры. В программе СЧЕТ они указаны (для удобства) в виде комментариев.

```
PROGRAM СЧЕТ (INPUT, OUTPUT);
CONST ДЛИНА-ИМЕНИ-ПРОЦ=24;
TYPE ТИП-УЗЛА=(ОПЕРАЦИЯ, ФУНКЦИЯ, КОНСТАНТА,
ПЕРЕМЕННАЯ);
ИМЯ ФУНКЦИИ=(FSIN, FCOS, FLN, FEXP, FABS,
FSQRT, МИНУС);
(*МИНУС СООТВЕТСТВУЕТ УНАРНОМУ
МИНУСУ*)
ДЕРЕВО=↑УЗЕЛ;
УЗЕЛ=RECORD CASE ТИП : ТИП УЗЛА OF
ОПЕРАЦИЯ: (ЗНАК-ОПЕРАЦИИ: CHAR;
ОП1, ОП2: ДЕРЕВО);
ФУНКЦИЯ: (ИМЯ: ИМЯ-ФУНКЦИИ;
АРГУМЕНТ: ДЕРЕВО);
КОНСТАНТА: (ЗНАЧЕНИЕ: REAL)
ПЕРЕМЕННАЯ: ( )
END;
ТИП-ЛЕКСЕМЫ=(Л-НАЧАЛО, Л-ИМЯ-ФУНК, Л-ПЕРЕМ,
Л-КОНСТ,
Л-ОТКРС, Л-ЗАКРС, Л-ОШ-ИМЯ, Л-ОШ-КОНСТ, Л-ИНСТР);
ИМЯ-ПРОЦ=PACKED ARRAY
[1..ДЛИНА-ИМЕНИ-ПРОЦ] OF CHAR;
VAR СИНТ-ОШ,
(*ПРИЗНАК СИНТАКСИЧЕСКОЙ ОШИБКИ В ДАННЫХ *)
F-ОШ,
(* ПРИЗНАК ОШИБКИ В ОПИСАТЕЛЕ ФУНКЦИИ *)
ВЫЧ-ОШ,
(* ПРИЗНАК ОШИБКИ ПРИ ВЫЧИСЛЕНИИ ФУНКЦИИ *)
ЗАВИСИТ-ОТ-X (* =TRUE, ЕСЛИ ФУНКЦИЯ ЗАВИСИТ
ОТ X *)
:BOOLEAN;
ЛЕКСЕМА: ТИП-ЛЕКСЕМЫ (*ПРОЧИТАННОЙ ИЗ
ДАННЫХ *);
БУКВЫ, ЦИФРЫ, ЗНАКИ: SET OF CHAR;
Д: ДЕРЕВО
(* ПОСЛЕДНЕГО ПРОЧИТАННОГО ОПИСАТЕЛЯ
ФУНКЦИИ *);
X (* ЗНАЧЕНИЕ ПЕРЕМЕННОЙ X *),
РЕЗ: REAL (* ЗНАЧЕНИЕ F(X) *);
(* СТЕПЕНЬ *)
FUNCTION СТЕПЕНЬ (M : REAL, N : INTEGER): REAL;
(* ВЫЧИСЛЕНИЕ M В СТЕПЕНИ N, N >= 0 *)
VAR P: REAL;
```

```

BEGIN P:=1;
  WHILE N>0 DO BEGIN
    WHILE NOT ODD(N) DO BEGIN
      N:=N DIV 2; M:=M*M
    END;
    N:=N-1; P:=P*M
  END; СТЕПЕНЬ:=P
END (*FUNCTION СТЕПЕНЬ *);
(* ЗНАЧ *)
PROCEDURE ЗНАЧ (Д : ДЕРЕВО; X:REAL; VAR РЕЗ:REAL);
(* НА ВХОДЕ: Д — ДЕРЕВО ФОРМУЛЫ,
  X — ЗНАЧЕНИЕ ПЕРЕМЕННОЙ *)
(* НА ВЫХОДЕ: РЕЗ — РЕЗУЛЬТАТ ВЫЧИСЛЕНИЯ
  ФОРМУЛЫ *)
LABEL 99;
VAR P1, P2: REAL;
BEGIN WITH Д DO
  CASE ТИП OF
    ОПЕРАЦИЯ: BEGIN ЗНАЧ(ОП1, X, P1);
      IF ВЫЧ-ОШ THEN GOTO 99;
      ЗНАЧ(ОП2, X, P2);
      IF ВЫЧ-ОШ THEN GOTO 99;
      CASE ЗНАК-ОПЕРАЦИИ OF
        '+': РЕЗ:=P1+P2;
        '-': РЕЗ:=P1-P2;
        '*': РЕЗ:=P1*P2;
        '/': IF P2=0 THEN BEGIN ВЫЧ-ОШ:=TRUE;
          WRITELN('**** ДЕЛЕНИЕ НА 0'); GOTO 99
        END
        ELSE РЕЗ:=P1/P2;
      '+↑': IF P2>0 THEN РЕЗ:=СТЕПЕНЬ(P1, ROUND(P2))
        ELSE IF P1=0 THEN BEGIN ВЫЧ-ОШ:=TRUE;
          WRITELN('**** В СТЕПЕНИ ', P2:10); GOTO 99
        END
        ELSE РЕЗ:=1/СТЕПЕНЬ(P1, ROUND(-P2))
      END (*CASE ЗНАК ОПЕРАЦИИ *)
    END (*ОПЕРАЦИЯ *);
  ФУНКЦИЯ: BEGIN ЗНАЧ(АРГУМЕНТ, X, P1);
    IF ВЫЧ-ОШ THEN GOTO 99;
    CASE ИМЯ OF
      FSIN: РЕЗ:=SIN (P1);
      FCOS: РЕЗ:=COS (P1);
      FLN: IF P2<=0 THEN BEGIN ВЫЧ-ОШ:=TRUE;
        WRITELN('****LN(', P1 : 10, ')'); GOTO 99
      END
      ELSE РЕЗ:=LN(P1);
      FEXP: РЕЗ:=EXP (P1);
      FABS: РЕЗ:=ABS (P1);
      FSQRT: IF P1<0 THEN BEGIN ВЫЧ-ОШ:=
        TRUE;
        WRITELN('****SQRT(', P1 : 10, ')');
        GOTO 99
      END
      ELSE РЕЗ:=SQRT (P1);
    МИНУС: РЕЗ:= -P1
  END (*CASE ИМЯ *)
END (*ФУНКЦИЯ *);

```

```

    ПЕРЕМЕННАЯ: РЕЗ:=X;
    КОНСТАНТА: РЕЗ:=ЗНАЧЕНИЕ
END (* CASE ТИП *);
99: END (* PROCEDURE ЗНАЧ *);
(* ПРОБЕЛЫ *)
PROCEDURE ПРОБЕЛЫ;
BEGIN (* ПРОПУСТИТЬ ПРОБЕЛЫ В ФАЙЛЕ INPUT *)
    WHILE NOT EOF(INPUT) AND (INPUT↑=' ') DO BEGIN
        IF EOLN(INPUT)
            THEN WRITELN ELSE WRITE(' '); GET(INPUT)
    END
END (* PROCEDURE ПРОБЕЛЫ *);
(* ДАЛЬШЕ *)
PROCEDURE ДАЛЬШЕ;
(* ЧТЕНИЕ ИЗ ФАЙЛА INPUT ДО СЛЕДУЮЩЕЙ ЛИТЕРЫ,
    ОТЛИЧНОЙ ОТ ПРОБЕЛА, НЕ СЧИТАЯ ТЕКУЩЕЙ *)
BEGIN WRITE(INPUT↑); GET(INPUT); ПРОБЕЛЫ
END (* PROCEDURE ДАЛЬШЕ *);
(* ОШБ *)
PROCEDURE ОШБ (ПР: ИМЯ-ПРОЦ);
(* ПЕЧАТЬ СООБЩЕНИЯ ОБ ОШИБКЕ. ХАРАКТЕР
    ОШИБКИ ОПРЕДЕЛЯЕТСЯ ПО ЗНАЧЕНИЯМ
    ПЕРЕМЕННЫХ
    ЛЕКСЕМА И INPUT↑*)
LABEL 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 99;
BEGIN СИНТ-ОШ:=TRUE; WRITELN(INPUT↑);
    WRITELN(' ***СИНТАКСИЧЕСКАЯ ОШИБКА В',
        ' ДАННЫХ');
    WRITELN(' ОБНАРУЖЕНА В ПРОЦЕДУРЕ ');
    WRITELN(' ', ПР);
    IF EOF(INPUT) THEN GOTO 9;
    CASE ЛЕКСЕМА OF
        Л-ИНСТР: GOTO 10;
        Л-ОШ-ИМЯ: GOTO 7;
        Л-ОШ-КОНСТ: GOTO 11;
        Л-НАЧАЛО: IF INPUT↑=')' THEN GOTO 3
                    ELSE IF INPUT↑ IN ЗНАКИ THEN GOTO 4
                    ELSE IF INPUT↑=';' THEN GOTO 6
                    ELSE GOTO 8;
        Л-ИМЯ-ФУНК: IF INPUT↑ IN БУКВЫ+ЦИФРЫ+ЗНАКИ+
                    [';', ' ', ')']
                    THEN GOTO 1
                    ELSE GOTO 8;

        Л-ПЕРЕМ,
        Л-КОНСТ,
        Л-ЗАКРС: IF INPUT↑ IN БУКВЫ+ЦИФРЫ+['('] THEN
                    GOTO 2
                    ELSE IF (INPUT↑=')') OR (INPUT↑=';') THEN
                    GOTO 3
                    ELSE GOTO 8;

        Л-ЗНАК: IF INPUT↑ IN ЗНАКИ+[';', ' '] THEN GOTO 4
                    ELSE GOTO 8;

        Л-ОТКРС: IF INPUT↑=')' THEN GOTO 5
                    ELSE IF INPUT↑ IN ЗНАКИ THEN GOTO 4
                    ELSE IF INPUT↑=';' THEN GOTO 3
                    ELSE GOTO 8
    END

```



```

END (* CASE ЛЕКСЕМА *);
WRITELN('**** ОШИБКА В ПРОГРАММЕ СЧЕТ ');
GOTO 99;
(* СООБЩЕНИЯ О СИНТАКСИЧЕСКИХ ОШИБКАХ *)
1:WRITELN(' АРГУМЕНТ ФУНКЦИИ НЕ В СКОБКАХ');
GOTO 99;
2:WRITELN(' НЕТ РАЗДЕЛИТЕЛЯ МЕЖДУ ',
' ОПЕРАНДАМИ'); GOTO 99;
3:WRITELN(' НЕСООТВЕТСТВИЕ СКОБОК'); GOTO 99;
4:WRITELN(' ОТСУТСТВУЕТ ОПЕРАНД АРИФМ. ',
' ОПЕРАЦИИ');
GOTO 99;
5:WRITELN(' ПУСТЫЕ СКОБКИ'); GOTO 99;
6:WRITELN(' ПУСТАЯ ФОРМУЛА'); GOTO 99;
7:WRITELN(' НЕВЕРНОЕ ИМЯ ФУНКЦИИ'); GOTO 99;
8:WRITELN(' НЕДОПУСТИМАЯ ЛИТЕРА'); GOTO 99;
9:WRITELN(' НЕТ; ПОСЛЕ ИНСТРУКЦИИ'); GOTO 99;
10:WRITELN(' НЕДОПУСТИМАЯ ИНСТРУКЦИЯ'); GOTO 99;
11:WRITELN(' ОШИБКА В ВЕЩ. КОНСТАНТЕ'); GOTO 99;
99:WHILE NOT EOF(INPUT) AND (INPUT↑ <> ';')
DO GET (INPUT);
IF NOT EOF(INPUT) THEN READLN
END (* PROCEDURE ОШЕ *);
(* ПРОЦЕДУРЫ ОПИСАНИЕ-ФУНКЦИИ, ФОРМУЛА,
СЛАГАЕМОЕ,
СОМНОЖИТЕЛЬ, ЭЛЕМЕНТАРНАЯ-ФОРМУЛА ЧИТАЮТ
ТЕКСТ ИЗ ФАЙЛА INPUT И СОПОСТАВЛЯЮТ
ЕГО С СООТВЕТСТВУЮЩЕЙ СИНТАКСИЧЕСКОЙ
КОНСТРУКЦИЕЙ. НА ВЫХОДЕ:
ЕСЛИ НЕ БЫЛО ОШИБОК, ТО Д — ДЕРЕВО
КОНСТРУКЦИИ, СИНТ-ОШ=FALSE.
ЕСЛИ НАЙДЕНА ОШИБКА, ТО СИНТ-ОШ=TRUE *)
(* ОПИСАНИЕ- ФУНКЦИИ *)
PROCEDURE ФОРМУЛА (VAR Д:ДЕРЕВО); FORWARD;
PROCEDURE ОПИСАНИЕ-ФУНКЦИИ (VAR Д:ДЕРЕВО);
LABEL 99;
VAR ПР:ИМЯ-ПРОЦ;
I: INTEGER; F: PACKED ARRAY [1..4] OF CHAR;
BEGIN ПР:=' ОПИСАНИЕ-ФУНКЦИИ ';
F:=' ';
FOR I:=1 TO 4 DO BEGIN ДАЛЬШЕ; F[I]:=INPUT↑ END;
IF F<>'(X)=' THEN BEGIN ОШБ(ПР); GOTO 99 END;
ДАЛЬШЕ; ЛЕКСЕМА:=Л-НАЧАЛО;
ФОРМУЛА (Д); IF СИНТ-ОШ THEN GOTO 99;
IF EOF(INPUT) OR (INPUT↑ < > ';') THEN
BEGIN ОШБ(ПР); GOTO 99 END;
READLN; WRITELN;
99: END (* PROCEDURE ОПИСАНИЕ- ФУНКЦИИ *);
(* ФОРМУЛА *)
PROCEDURE СЛАГАЕМОЕ (VAR Д:ДЕРЕВО); FORWARD;
PROCEDURE ФОРМУЛА (* VAR Д:ДЕРЕВО *);
LABEL 99;
VAR ПР:ИМЯ-ПРОЦ; Д1, Д2: ДЕРЕВО; ЗНАК:CHAR;
BEGIN ПР:=' ФОРМУЛА ';
ЗНАК:='+';

```

```

IF NOT EOF(INPUT) AND ((INPUT↑='+') OR
(INPUT↑='−'))
  THEN BEGIN ЗНАК:=INPUT↑; ДАЛЬШЕ;
  ЛЕКСЕМА:=Л-ЗНАК
END;
СЛАГАЕМОЕ(Д); IF СИНТ-ОШ THEN GOTO 99;
IF ЗНАК='−' THEN BEGIN (* УНАРНЫЙ МИНУС *)
  Д1:=Д; NEW(Д, ФУНКЦИЯ);
  Д↑.ТИП:=ФУНКЦИЯ;
  Д↑.ИМЯ:=МИНУС;
  Д↑.АРГУМЕНТ:=Д1
END;
WHILE NOT EOF(INPUT)
  AND ((INPUT↑='+') OR (INPUT↑='−'))
  DO BEGIN (* Д — ДЕРЕВО ПРОЧИТАННОЙ ЧАСТИ
  ФОРМУЛЫ *)
    ЗНАК:=INPUT↑; ДАЛЬШЕ; ЛЕКСЕМА:=Л-ЗНАК;
    СЛАГАЕМОЕ(Д2); IF СИНТ-ОШ THEN GOTO 99;
    Д1:=Д; NEW(Д, ОПЕРАЦИЯ);
    Д↑.ТИП:=ОПЕРАЦИЯ;
    Д↑.ЗНАК-ОПЕРАЦИИ:=ЗНАК;
    Д↑.ОП1:=Д1; Д↑.ОП2:=Д2;
  END;
99: END (* PROCEDURE ФОРМУЛА *);
PROCEDURE СОМНОЖИТЕЛЬ (VAR Д:ДЕРЕВО); FORWARD;
(* СЛАГАЕМОЕ *)
PROCEDURE СЛАГАЕМОЕ (* VAR Д : ДЕРЕВО *);
LABEL 99;
VAR ПР : ИМЯ-ПРОЦ; Д1,Д2:ДЕРЕВО; ЗНАК:CHAR;
BEGIN ПР:='СЛАГАЕМОЕ';
  СОМНОЖИТЕЛЬ(Д); IF СИНТ-ОШ THEN GOTO 99;
  WHILE NOT EOF(INPUT)
    AND((INPUT↑='*') OR (INPUT↑='/'))
    DO BEGIN (* Д — ДЕРЕВО ПРОЧИТАННОЙ ЧАСТИ
    СЛАГАЕМОГО *)
      ЗНАК:=INPUT↑; ДАЛЬШЕ; ЛЕКСЕМА:=Л-ЗНАК;
      СОМНОЖИТЕЛЬ(Д2); IF СИНТ-ОШ THEN GOTO 99;
      Д1:=Д; NEW(Д, ОПЕРАЦИЯ);
      Д↑.ТИП:=ОПЕРАЦИЯ;
      Д↑.ЗНАК-ОПЕРАЦИИ:=ЗНАК;
      Д↑.ОП1:=Д1; Д↑.ОП2:=Д2;
    END;
99: END (* PROCEDURE СЛАГАЕМОЕ *);
(* СОМНОЖИТЕЛЬ *)
PROCEDURE ЭЛЕМЕНТАРНАЯ_ФОРМУЛА (VAR Д:ДЕРЕВО);
FORWARD;
PROCEDURE СОМНОЖИТЕЛЬ (* VAR Д:ДЕРЕВО *);
LABEL 99;
VAR ПР:ИМЯ-ПРОЦ; Д1, Д2 : ДЕРЕВО;
BEGIN ПР:='СОМНОЖИТЕЛЬ';
  ЭЛЕМЕНТАРНАЯ-ФОРМУЛА(Д); IF СИНТ-ОШ THEN GOTO
  99;
  WHILE NOT EOF(INPUT) AND (INPUT↑='↑')
  DO BEGIN
    (*Д — ДЕРЕВО ПРОЧИТАННОЙ ЧАСТИ
    СОМНОЖИТЕЛЯ*)

```

```

ДАЛЬШЕ; ЛЕКСЕМА:=Л-ЗНАК;
ЭЛЕМЕНТАРНАЯ-ФОРМУЛА (Д2);
IF СИНТ-ОШ THEN GOTO 99;
Д1:=Д; NEW (Д, ОПЕРАЦИЯ);
Д↑.ТИП:=ОПЕРАЦИЯ;
Д↑.ЗНАК-ОПЕРАЦИИ:='↑';
Д↑.ОП1:=Д1; Д↑.ОП2:=Д2;
END;
99: END (*PROCEDURE СОМНОЖИТЕЛЬ *);
(* ЭЛЕМЕНТАРНАЯ_ФОРМУЛА *)
PROCEDURE ЭЛЕМЕНТАРНАЯ-ФОРМУЛА (*VAR Д:
ДЕРЕВО *);
LABEL 99;
VAR ПР:ИМЯ-ПРОЦ; ИМЯ-Ф: ИМЯ-ФУНКЦИИ;
BEGIN ПР:='ЭЛЕМЕНТАРНАЯ-ФОРМУЛА';
IF EOF(INPUT) THEN ОШБ(ПР)
ELSE IF (INPUT↑>='0') AND (INPUT↑ <='9') THEN
BEGIN
(* КОНСТАНТА *)
NEW (Д, КОНСТАНТА);
Д↑.ТИП:=КОНСТАНТА;
ВЕЩЕСТВЕННОЕ(Д↑.ЗНАЧЕНИЕ);
IF СИНТ-ОШ THEN GOTO 99;
ЛЕКСЕМА:=Л-КОНСТ END
ELSE IF INPUT↑='X' THEN BEGIN (*ПЕРЕМЕННАЯ *)
NEW (Д, ПЕРЕМЕННАЯ); Д↑.ТИП:=ПЕРЕМЕННАЯ;
ЗАВИСИТ-ОТ-Х:=TRUE; ЛЕКСЕМА:=Л-ПЕРЕМ;
ДАЛЬШЕ END
ELSE IF INPUT↑='(' THEN BEGIN (*ФОРМУЛА В
СКОБКАХ *)
ЛЕКСЕМА:=Л-ОТКРС; ДАЛЬШЕ;
ФОРМУЛА (Д); IF СИНТ-ОШ THEN GOTO 99;
IF EOF(INPUT) OR (INPUT↑<>'') THEN
BEGIN ОШБ(ПР); GOTO 99 END; ДАЛЬШЕ;
END
ELSE BEGIN (*ОБРАЩЕНИЕ К ФУНКЦИИ *)
СТАНДАРТНАЯ-ФУНКЦИЯ(ИМЯ-Ф);
IF СИНТ-ОШ THEN GOTO 99;
IF EOF(INPUT) OR (INPUT↑<>'(') THEN
BEGIN ОШБ(ПР); GOTO 99 END;
ДАЛЬШЕ; ЛЕКСЕМА:=Л-ОТКРС;
NEW (Д, ФУНКЦИЯ);
Д↑.ТИП:=ФУНКЦИЯ;
Д↑.ИМЯ:=ИМЯ-Ф;
ФОРМУЛА (Д↑.АРГУМЕНТ); IF СИНТ-ОШ THEN GOTO 99;
IF EOF(INPUT) OR (INPUT↑<>'') THEN
BEGIN ОШБ(ПР); GOTO 99 END;
ДАЛЬШЕ; ЛЕКСЕМА:=Л-ЗАКРС
END;
99: END (* PROCEDURE ЭЛЕМЕНТАРНАЯ-ФОРМУЛА *);
(* ВЕЩЕСТВЕННОЕ *)
PROCEDURE ЦЕЛОЕ (VAR X: REAL); FORWARD;
PROCEDURE ВЕЩЕСТВЕННОЕ (* VAR X:REAL *);
(* ПРОЧИТАТЬ ВЕЩЕСТВЕННОЕ ЧИСЛО,
ЗНАЧЕНИЕ ЗАНЕСТИ В X *)
LABEL 99;

```



```

VAR D, E: REAL; ЗНАК: CHAR;
BEGIN ЦЕЛОЕ (X); IF СИНТ-ОШ THEN GOTO 99;
(* X = ЦЕЛАЯ ЧАСТЬ *)
IF NOT EOF(INPUT) AND (INPUT↑='.') THEN BEGIN
  WRITE(INPUT↑); GET(INPUT); D:=0.1;
  WHILE NOT EOF(INPUT)
    AND (INPUT↑ >='0') AND (INPUT↑ <='9') DO
    BEGIN X:=X+D*(ORD(INPUT↑)-ORD('0'));
      D:=D*0.1
    END; (*ПРОЧИТАЛИ ЦЕЛую И ДРОБНую ЧАСТИ
      ЧИСЛА*)
END;
IF NOT EOF(INPUT) AND (INPUT↑='E') THEN BEGIN
  WRITE(INPUT↑); GET(INPUT); ЗНАК:='+';
  IF NOT EOF(INPUT)
    AND ((INPUT↑='+') OR (INPUT↑='-'))
    THEN BEGIN ЗНАК:=INPUT↑; WRITE(INPUT↑); GET
      (INPUT)
    END;
  ЦЕЛОЕ (E); IF СИНТ-ОШ THEN GOTO 99;
  IF ЗНАК='+' THEN X:=X*СТЕПЕНЬ(10, ROUND(E))
    ELSE X:=X*СТЕПЕНЬ(0.1, ROUND(E));
END;
99: END (* PROCEDURE ВЕЩЕСТВЕННОЕ *);
(* ЦЕЛОЕ*)
PROCEDURE ЦЕЛОЕ (*VAR X: REAL *);
(* ПРОЧИТАТЬ ЦЕЛОЕ ЧИСЛО, ЗНАЧЕНИЕ ЗАНЕСТИ В X *)
VAR ПР:ИМЯ-ПРОЦ; ЦИФРА: BOOLEAN;
BEGIN ПР:=' ЦЕЛОЕ ';
  X:=0; ЦИФРА:=FALSE;
  WHILE NOT EOF(INPUT)
    AND (INPUT↑ >='0') AND (INPUT↑ <='9')
    DO BEGIN X:=X*10+ORD(INPUT↑) - ORD('0');
      WRITE(INPUT↑); GET(INPUT); ЦИФРА:=TRUE
    END;
  IF NOT ЦИФРА
    THEN BEGIN ЛЕКСЕМА:=Л-ОШ-КОНСТ; ОШБ(ПР) END
  END (* PROCEDURE ЦЕЛОЕ *);
(* СТАНДАРТНАЯ ФУНКЦИЯ *)
PROCEDURE СТАНДАРТНАЯ-ФУНКЦИЯ
(*VAR ИМЯ: ИМЯ-ФУНКЦИИ *);
(*ПРОЧИТАТЬ ИМЯ ЭЛЕМЕНТАРНОЙ ФУНКЦИИ
  И ЗАНЕСТИ
  В ПЕРЕМЕННую ИМЯ СООТВЕТСТВУЮЩЕЕ
  ЗНАЧЕНИЕ *);
CONST L=6;
TYPE СЛОВО =PACKED ARRAY [1..L] OF CHAR;
VAR ПР: ИМЯ-ПРОЦ; С: СЛОВО; I: INTEGER;
ИМЕНА: ARRAY [FSIN..МИНУС] OF СЛОВО;
BEGIN ПР:='СТАНДАРТНАЯ ФУНКЦИЯ ';
ИМЕНА[FSIN]:='SIN'; ИМЕНА[FCOS]:='COS ';
ИМЕНА[FLN]:='LN'; ИМЕНА[FEXP]:='EXP ';
ИМЕНА[FABS]:='ABS '; ИМЕНА[FSQRT]:='SQRT ';
С:=' '; I:=1;
(*ПРОЧИТАТЬ ИМЯ ФУНКЦИИ В ПЕРЕМЕННую С *)
  WHILE NOT EOF(INPUT)

```

```

AND (INPUT↑ IN БУКВЫ) AND (I<=L) DO
BEGIN C[I]:=INPUT↑; WRITE (INPUT↑); GET (INPUT);
I:=I+1
END;
IF I=1 THEN ОШБ(ПР) (* ИМЯ НЕ ПРОЧИТАНО *)
ELSE BEGIN (* НАЙТИ ИМЯ В МАССИВЕ ИМЕНА *)
ИМЯ:=FSIN; ИМЕНА [МИНУС]:=C;
(* ЗАНЕСЛИ ИСКОМОЕ ИМЯ В ПОСЛЕДНИЙ
ЭЛЕМЕНТ МАССИВА *)
WHILE C < > ИМЕНА [ИМЯ] DO ИМЯ:=СУСС(ИМЯ);
IF ИМЯ=МИНУС THEN BEGIN (* ИМЯ НЕ НАЙДЕНО *)
ЛЕКСЕМА:=Л-ОШ-ИМЯ; ОШБ(ПР) END
ELSE BEGIN ЛЕКСЕМА:=Л-ИМЯ-ФУНК; ПРОБЕЛЫ END
END
END (* PROCEDURE СТАНДАРТНАЯ-ФУНКЦИЯ *);
(*ВЫЧИСЛЕНИЕ *)
PROCEDURE ВЫЧИСЛЕНИЕ;
LABEL 99;
VAR ПР:ИМЯ-ПРОЦ; Д1: ДЕРЕВО;
BEGIN ПР:='ВЫЧИСЛЕНИЕ'; ДАЛЬШЕ;
IF EOF(INPUT) OR (INPUT↑ < > '=') THEN
BEGIN ОШБ(ПР); GOTO 99 END;
ДАЛЬШЕ; ФОРМУЛА(Д1);
IF СИНТ-ОШ THEN GOTO 99;
IF EOF(INPUT) OR (INPUT↑ < > ';') THEN
BEGIN ОШБ(ПР); GOTO 99 END;
Writeln(INPUT↑); READLN;
IF ЗАВИСИТ-ОТ-Х THEN BEGIN
Writeln(' ****ФОРМУЛА ЗАВИСИТ ОТ Х'); GOTO 99
END;
ЗНАЧ(Д1, 0, Х);
IF F-ОШ OR ВЫЧ-ОШ THEN GOTO 99;
ЗНАЧ(Д, Х, РЕЗ); IF ВЫЧ-ОШ THEN GOTO 99;
Writeln(' F(' , Х : 10, ')=' , РЕЗ : 10);
99: END (*PROCEDURE ВЫЧИСЛЕНИЕ *);
(*PROGRAM СЧЕТ *)
BEGIN (*УСТАНОВКА ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ *)
БУКВЫ:=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
'V', 'W', 'X', 'Y', 'Z'];
ЦИФРЫ:=['1'..'9'];
ЗНАКИ:=['+', '-', '*', '/', '↑'];
F-ОШ:=TRUE;
(*ОБРАБОТКА ДАННЫХ *)
WHILE NOT EOF(INPUT) DO BEGIN ПРОБЕЛЫ;
ЛЕКСЕМА:=Л-ИНСТР;
СИНТ-ОШ:=FALSE; ВЫЧ-ОШ:=FALSE;
ЗАВИСИТ-ОТ-Х:=FALSE;
IF INPUT↑='F' THEN BEGIN ОПИСАНИЕ_ФУНКЦИИ(Д);
F-ОШ:=СИНТ-ОШ;
IF NOT СИНТ-ОШ AND NOT ЗАВИСИТ-ОТ-Х THEN
BEGIN
ЗНАЧ(Д, 0, РЕЗ);
IF NOT ВЫЧ_ОШ THEN Writeln(' F(X)=' , РЕЗ : 10);
END
END
END

```

```

ELSE IF INPUT↑='X' THEN ВЫЧИСЛЕНИЕ
ELSE ОШБ('СЧЕТ');
END (* WHILE *)
END.

```

Приложение 8

РАБОТА С ТРАНСЛЯТОРАМИ

В этом приложении описана работа с паскаль-трансляторами на ЕС ЭВМ серии Ряд-2 и на персональных ЭВМ ЕС1840, «Ямаха» и им подобных. Приведен минимум сведений, необходимый для того, чтобы ввести программу в машину и выполнить ее. Для более подробного знакомства с конкретной паскаль-системой следует обратиться к документации по математическому обеспечению конкретной ЭВМ.

Использование транслятора Паскаль-8000, версия 2.0 на ЕС ЭВМ в операционной системе СВМ в подсистеме ПДО. Работа с программой состоит из двух этапов — трансляции и выполнения. В обоих случаях к виртуальной машине, на которой ведется работа, должен быть подключен диск, содержащий транслятор с паскаля.

Паскаль-программы должны храниться в файлах ПДО с типом PASCAL. Новую программу или программу, в которую были внесены изменения, необходимо оттранслировать. Для этого следует исполнить команду

PASCAL {имя программы}

В качестве имени программы следует указать имя файла ПДО типа PASCAL, в котором хранится текст программы. Транслятор выдает на дисплей (а также в файл типа LISTING) сообщения об ошибках, если они были обнаружены.

Если при трансляции программы ошибок не обнаружено, программу можно выполнить. Перед этим следует

- 1) подключить библиотеку вспомогательных программ:

GLOBAL TXTLIB PASCAL

2) описать внешние файлы, с которыми будет работать программа. Для каждого внешнего файла следует исполнить команду

FILEDEF {паскаль-имя файла} {описание файла ПДО}

Здесь {паскаль-имя файла} — имя файла, как оно записано в программе (внутри скобок в ее заголовке). Стандартные файлы INPUT и OUTPUT представляют исключение. В команде FILEDEF вместо паскаль-имени следует указывать SYSIN — для файла INPUT, SYSPRINT — для файла OUTPUT. Кроме того, можно вообще не задавать команду FILEDEF для файлов INPUT и OUTPUT, в этом случае данные будут вводиться с клавиатуры терминала и выводиться на экран. Таким образом, в простейшем случае можно обойтись вообще без команд FILEDEF.

Из различных возможных способов описания файла ПДО в команде FILEDEF упомянем два наиболее употребительных. Если мы хотим, чтобы файл располагался на диске, то в качестве описания файла ПДО нужно указать

DISK <имя файла ПДО> <тип файла ПДО>

Если же требуется осуществлять вывод на экран дисплея или ввод с клавиатуры, то в качестве описания файла ПДО указывается одно слово

TERM

Собственно выполнение программы начинается по команде

LOAD <имя программы> (START)

Здесь <имя программы> — то же, что в команде PASCAL.

П р и м е р. Пусть мы хотим создать программу в файле TEST PASCAL. Для создания программы воспользуемся, например, редактором XEDIT:

XEDIT TEST PASCAL

С помощью команд редактора введем программу:

```
PROGRAM EXAMPLE(INPUT, OUTPUT, FA);  
VAR FA : TEXT;
```

```
...  
END.
```

Заключив редактирование, оттранслируем программу.

PASCAL TEST

Теперь выполним программу. Пусть мы хотим, чтобы исходные данные (файл INPUT) вводились с терминала, результат (файл OUTPUT) записывался в файл EXAMPLE RESULT, а паскаль-файл FA отображался на дисплей. Тогда команду FILEDEF SYSIN можно не исполнять:

```
GLOBAL TXTLIB PASCAL  
FILEDEF SYSPRINT DISK EXAMPLE RESULT  
FILEDEF FA TERM  
LOAD TEST(START)
```

Если программа при выполнении зациклилась, ее можно остановить, введя команду

DUMP

Выполнение программы будет прервано и на дисплей будут выданы текущие значения переменных программы, по которым можно попытаться найти место ошибки. Такая же информация выводится на дисплей в случае обнаружения ошибки при выполнении программы.

Язык Паскаль-8000 несколько отличается от описанного в книге:

- 1) В именах нельзя использовать русские буквы.
- 2) Вместо открывающей и закрывающей квадратных скобок в программах можно использовать комбинации, соответственно, (и .); для того же, чтобы использовать настоящие квадратные скобки, следует перед началом работы исполнить команды

```
SET OUTPUT AD [  
SET OUTPUT BD ]  
SET INPUT [ AD  
SET INPUT ] BD
```

Если предполагается работа только с паскалем, целесообразно поместить эти команды, а также команду GLOBAL в файл PROFILE EXEC. В этом случае операционная система будет автоматически исполнять все эти команды, и их не надо будет вводить в каждом сеансе работы.

Использование системы Турбо-паскаль на ПЭВМ. Система Турбо-паскаль имеется на ПЭВМ многих типов. Она обеспечивает весьма быструю трансляцию программ.

Прежде чем начать работу с системой, следует сделать ее доступной. Если система Турбо-паскаль находится на встроенном в ЭВМ несъемном диске, следует с помощью команды CD сделать текущей директорию, содержащую систему, например,

CD\TURBO

Если система записана на дискете, следует установить дискету в дисковод и настроиться на работу с этим дисководом, например, с помощью команды

A:

После этого можно вызвать систему, набрав на клавиатуре TURBO

Система задаст вопрос

Include error messages (Y/N)?

на который, как правило следует ответить утвердительно, Y. Это означает, что сообщения об ошибках будут выдаваться в виде текстов. Если же ответить N, то будут выдаваться только номера ошибок.

Затем на экране появится «основное меню» системы Турбо-паскаль, содержащее команды, которые в данный момент можно выполнить. Для того чтобы выполнить команду, нужно ввести с клавиатуры одну букву, которая выделена в меню как заглавная.

Типичная последовательность действий такова:

1. Нажимаем на клавишу W (Workfile — рабочий файл). В ответ на вопрос системы вводим имя файла, содержащего текст программы. Если мы создаем новую программу, то следует ввести имя файла, в котором мы хотим запомнить созданную программу.

2. Затем можно при желании ввести изменения в программу, исполнив команду E (Edit — редактировать). На экране появится текст программы и мы попадем в редактор системы Турбо-паскаль. С помощью клавиш управления курсором можно переместиться в любое место программы и ввести с клавиатуры какую-либо вставку или удалить часть имеющегося текста. Кроме того, редактор позволяет при помощи специальных команд выполнять более сложные исправления. Все команды редактора вводятся при нажатой клавише УПР (на машинах с латинизированной клавиатурой на этой клавише написано Ctrl). Для ввода команды редактора нужно нажать клавишу УПР и, не отпуская ее, нажать одну или две клавиши, обозначающие нужную команду. Например, команда УПР+Y удаляет из текста строку. Для выхода из редактора в основное меню используется команда УПР+KD.

В системе Турбо-паскаль не требуется, чтобы программы обязательно создавались с помощью ее собственного редактора, так

что, если вы уже работали на ПЭВМ, но до сих пор не пользовались Турбо-паскалем, вы можете готовить программы с помощью любого привычного вам редактора, а в системе Турбо-паскаль только выполнять их.

3. Следующий этап — выполнение программы, для этого служит команда R (Run — выполнить). Система автоматически оттранслирует программу и, в случае отсутствия ошибок при трансляции начнет ее выполнять. Если при трансляции или выполнении программы будут обнаружены ошибки, система перейдет в режим редактора, указав место найденной ошибки. (При выполнении программы вне системы Турбо-паскаль выдается только адрес ошибки в виде PC=(шестнадцатеричное число). Чтобы найти место в программе, следует использовать команды O, F (Options — необязательные параметры, Find — находить) и затем ввести выданный адрес.)

4. Выход из системы Турбо-паскаль осуществляется по команде Q (Quit — прекратить). При этом система спросит, надо ли сохранять на диске изменения, внесенные в программу:

Save (Y/N)?

Если ответить Y, изменения будут записаны на диск. При ответе N записи на диск не будет, программа, хранящаяся на диске, останется такой же, какой была до входа в Турбо-паскаль. Последний вариант удобно использовать, чтобы поэкспериментировать с какой-либо имеющейся программой: внести в нее изменения, посмотреть, что получится, а затем разом отказаться от всех изменений.

Язык Турбо-паскаль содержит ряд отличий от стандартного паскаля, описанного в книге. В частности:

1. В именах нельзя использовать русские буквы.
2. Отсутствуют стандартные процедуры PUT и GET. Для ввода-вывода следует использовать процедуры READ и WRITE.
3. Не допускается передача процедур и функций в качестве параметров.
4. В заголовке программы не указываются внешние файлы, заголовок имеет вид:

```
PROGRAM (имя программы);
```

5. Имеется большой набор стандартных процедур и функций, облегчающих составление программ. Упомянем лишь две процедуры работы с файлами. Процедура ASSIGN позволяет связать паскаль-файл с файлом на диске. Первый параметр процедуры — паскаль-файл, второй — текст (в кавычках) — имя файла на диске.

Пример.

```
VAR F:TEXT;  
...  
ASSIGN(F, 'TEST.DAT')  
...
```

Процедура CLOSE ((паскаль-имя файла)) осуществляет запись на диск изменений, внесенных в файл при работе паскаль-программы.

Всякая работа с внешними файлами должна начинаться с вызова процедуры ASSIGN, вслед за которой нужно вызвать процедуру RESET или REWRITE; работа заканчивается вызовом процедуры CLOSE.

Дополнительные возможности системы Турбо-паскаль. Пожалуй, наиболее простым способом расширения языка программирования является добавление новых стандартных процедур. Именно по такому пути пошли авторы системы Турбо-паскаль, чтобы сделать доступными для программистов, пишущих на паскале, аппаратные возможности современных персональных ЭВМ. Ниже следует беглое описание соответствующих стандартных процедур и функций применительно к версии системы Турбо на IBM/PC и совместимых ПЭВМ. Для краткости опускается слово *procedure*, описания типов формальных параметров и результатов функций. X и Y обозначают координаты на экране, C — цвет.

1. Работа с текстовым экраном.

CrtInit — инициализация работы с текстовым экраном; на терминал посылается строка, определенная при генерации системы.

CrtExit — окончание работы с текстовым экраном; на терминал посылается строка, определенная при генерации системы.

ClrScr — очистка экрана; курсор перемещается в левый верхний угол.

ClrEol — удаление конца строки от позиции курсора.

DelLine — удаление строки, на которой стоит курсор, и сдвиг всех нижележащих строк на одну вверх.

InsLine — вставка пустой строки в том месте, где стоит курсор.

GotoXY (X, Y) — позиционирование курсора в столбец с номером X и строку с номером Y.

TextMode — перевод экрана в текстовый режим.

TextColor (C) — установка цвета литер.

TextBackground (C) — установка цвета фона.

function WhereX — опрос X-координаты курсора.

function WhereY — опрос Y-координаты курсора.

2. Работа с графическим экраном.

HiResColor (C) — установка режима высокого разрешения с цветом изображения C.

GraphColorMode — установка цветного графического режима нормального разрешения.

Palette (N: 0..3) — установка текущей палитры.

GraphBackground (C) — установка цвета фона.

3. Работа с окнами.

Window (X1, Y1, X2, Y2) — установка активного текстового окна.

GraphWindow (X1, Y1, X2, Y2) — установка активного графического окна.

4. Графические примитивы.

Plot (X, Y, C) — рисование точки.

Draw (X1, Y1, X2, Y2, C) — рисование отрезка.

5. Дополнительные графические возможности. Дополнительные графические подпрограммы содержатся в библиотеке GRAPH.BIN., а их интерфейс описан в файле GRAPH.P. Чтобы воспользоваться дополнительными графическими возможностями, в программу следует вставить строку

```
{$I GRAPH.P}
```

Предоставляемые процедуры:

ColorTable (C0, C1, C2, C3) — установка таблицы перекодировки цветов. Если меняется цвет некоторой точки экрана, и раньше это был цвет I: 0..3, то новый цвет будет C1. Таблица используется процедурой PutPic (см. ниже). Другие рисующие процедуры используют таблицу, если при обращении к ним задан цвет —1.

Arc (X, Y, A, R, C) — рисование дуги радиуса R с градусной мерой A начиная с точки (X, Y).

Circle (X, Y, R, C) — рисование окружности.

GetPic (B, X1, Y1, X2, Y2) — копирование содержимого прямоугольной области экрана в переменную B . Для цветного режима нормального разрешения переменная B должна занимать не менее

$((abs(X2-X1)+1+3) \div 4) * (abs(Y2-Y1)+1) * 2+6$ байт.

PutPic (B, X, Y) — вывод содержимого переменной B в прямоугольную область экрана, где (X, Y) — координаты ее левого нижнего угла. При выводе содержимое переменной преобразуется в соответствии с таблицей перекодировки цветов.

function GetDotColor (X, Y) — опрос цвета точки. Если точка вне активного окна, результат равен -1.

FillScreen (C) — заполнение активного окна заданным цветом. Если задан цвет -1, а таблица перекодировки цветов имеет вид 3, 2, 1, 0, выполнится инвертирование изображения.

FillShape (X, Y, CF, CB) — заполнение области, граница которой имеет цвет CB , цветом CF начиная с точки (X, Y). Таблица перекодировки игнорируется.

FillPattern (X1, Y1, X2, Y2, C) — заполнение прямоугольной области шаблонами, заданными процедурой Pattern (P).

Pattern (P) — определение текущего шаблона — битового образа квадрата 8 на 8 точек. Определим шаблон, изображающий наклонную стрелку:

const

P: array [0..7] of Byte=

(\$F8, \$F0, \$F8, \$FC, \$BE, \$8F, \$0E, \$04)

В этом примере использованы возможности системы Турбо-паскаль по заданию структурных и шестнадцатеричных констант. Тип Byte — стандартный в Турбо-паскале и определяется как

type Byte = 0..255

6. Несколько полезных процедур и функций.

Randomize — установка случайного начального значения датчика случайных чисел.

function Random — получение очередного случайного числа в диапазоне от 0 до 1.

function KeyPressed — возвращает значение истина, если на клавиатуре нажата какая-либо клавиша.

Delay (T) — задержка на T миллисекунд (T — целое).

Sound (H) — генерация звука частоты H герц (H — целое). Звук длится до выполнения процедуры NoSound.

NoSound — прекращение звучания.

ТЕРМИНОЛОГИЧЕСКИЙ СЛОВАРЬ

1. *Данные* — представление сведений в виде, пригодном для восприятия и обработки человеком или автоматом. Одни и те же сведения можно представить разными способами — более удобными для человека (тексты, рисунки) или для ЭВМ (перфокарты, электрические сигналы).

2. *Информация* — смысл того, что человек выражает в данных или воспринимает из данных. Часто слова «информация» и «данные» употребляют как синонимы.

3. *Информатика* — наука о законах и методах получения, обработки и использования информации с помощью ЭВМ.

4. *Алгоритм* — точное предписание, которое задает процесс преобразования исходных данных в результаты. Близкими по смыслу являются слова «метод», «рецепт». Примером алгоритма может служить алгоритм Евклида для нахождения наибольшего общего делителя двух натуральных чисел.

5. *Электронная вычислительная машина (ЭВМ)*, или *компьютер*, — автомат, способный выполнять сложные преобразования информации, включая разнообразные арифметические и логические действия, ввод и вывод информации без вмешательства человека (в отличие от калькулятора, предназначенного в первую очередь для арифметических расчетов и требующего вмешательства человека-оператора на каждой стадии вычислений). ЭВМ действует по программе, которая хранится в ее памяти. За счет возможности смены программ достигается универсальность ЭВМ.

6. *Персональная ЭВМ (ПЭВМ)* — ЭВМ индивидуального пользования. ПЭВМ применяются в быту и профессиональной деятельности, для обучения, планирования семейного бюджета, для игр и т. д. Отличаются от других ЭВМ не только малыми размерами, но и особой, «дружественной» манерой общения с человеком. Персональная ЭВМ, подключенная к сети ЭВМ, становится «умным» терминалом, открывающим доступ к информационным системам, другим персональным ЭВМ и большим ЭВМ. Персональная ЭВМ — массовое средство приобщения людей к информатике.

7. *Вычислительный центр (ВЦ)* — организация, в которой сосредоточены несколько ЭВМ, обеспечивающая их нормальную работу и предоставляющая возможность пользователям решать разнообразные задачи.

8. *Сеть ЭВМ* — совокупность ЭВМ, соединенных каналами связи. Позволяет наиболее полно удовлетворить потребности пользователей в выполнении информационно-вычислительных работ.

9. *Центральный процессор (ЦП)*, часто просто *процессор*, — устройство, выполняющее арифметические и логические операции,

заданные программой. Управляет вычислительным процессом, координирует работу устройств ввода/вывода.

10. *Память* (запоминающее устройство, ЗУ) — устройство, предназначенное для хранения данных. Различают оперативную и вспомогательную память. Оперативная память (оперативное запоминающее устройство, ОЗУ) содержит программы, а также данные, на которые могут указывать адреса в машинных командах. Вспомогательная память (внешнее запоминающее устройство, ВЗУ) имеет большую, чем ОЗУ, емкость, но и большее время доступа к данным.

11. *Устройство ввода/вывода* (УВВ) — устройство, обеспечивающее обмен данными между ЭВМ и внешним миром. К устройствам ввода/вывода относится терминал, алфавитно-цифровое печатающее устройство (АЦПУ), устройство речевого ввода/вывода и т. д.

12. *Терминал* — устройство ввода/вывода, имеющее клавиатуру для ввода данных и экран (дисплей) для вывода данных в виде текстов, графиков и рисунков. В персональных ЭВМ в качестве дисплея часто используется бытовой телевизор.

13. *Программа* — описание алгоритма решения задачи, заданное в понятной компьютеру форме. Состоит из команд (инструкций). Процесс составления программ называется программированием. Специалистов по составлению программ называют программистами.

14. *Язык программирования* — искусственный язык, специально предназначенный для записи программ. Различают машинный язык и языки более высокого уровня. Машинный язык (система команд ЭВМ) интерпретируется аппаратурой ЭВМ. Для выполнения программ, написанных на языках высокого уровня, они переводятся на машинный язык с помощью специальной программы — транслятора. К настоящему времени создано несколько сотен языков высокого уровня. Наиболее известные из них — фортран, кобол, лисп, симбол, паскаль, алгол, ада, бейсик.

15. *Команда* (инструкция) — указание, определяющее конкретное действие из набора возможных действий. Машинные команды задают арифметические операции (сложение, умножение и т. п.), логические операции (И, ИЛИ и т. п.) или управляющие действия процессора (например, продолжить выполнение программы с определенного адреса, если истинно некоторое условие). Инструкции языков высокого уровня (например, обращение к подпрограмме), как правило, переводятся транслятором в несколько машинных команд.

16. *Цикл* — последовательность инструкций, которые выполняются несколько раз. Циклы позволяют в полной мере воспользоваться быстродействием ЭВМ, поскольку дают возможность много раз выполнять однажды записанные инструкции. Для организации циклов в языках программирования предусмотрены специальные виды инструкций.

17. *Подпрограмма* — часть программы, реализующая определенный алгоритм и допускающая обращение к ней из различных мест общей программы. Использование библиотек подпрограмм существенно облегчает решение сложных задач. С расширением библиотеки подпрограмм ЭВМ становится умнее.

18. *Файл* — набор взаимосвязанных данных, например текст программы, список подписчиков журнала, перечень деталей, библиотечный каталог и т. п. При ручной обработке данных аналогом машинного файла является картотека.

19. *Операционная система (ОС)* — совокупность программ, облегчающих общение человека и ЭВМ, обеспечивающих эффективное использование машинных ресурсов — времени, оперативной и вспомогательной памяти и т. п. Операционная система избавляет человека от прямого общения со сложной аппаратурой ЭВМ. Операционная система фактически предоставляет человеку более удобную и умную машину.

20. *База данных (БД)* — интегрированный набор данных, необходимых для решения задач из какой-либо области, например географическая, конструкторская, технологическая, медицинская база данных.

21. *Информационно-поисковая система (ИПС)* — программа, обеспечивающая поиск нужной информации в базе данных.

22. *Математическая модель* — описание явления или предмета реального мира математическим языком (например, системой уравнений). Строятся модели физических, биологических, экономических и других процессов и систем. Исследование математической модели с помощью ЭВМ (вычислительный эксперимент) позволяет предсказывать свойства и поведение моделируемых объектов.

23. *Обработка текстов* — создание, хранение, редактирование и печать текстовых документов с помощью ЭВМ. Текстовый документ — это любая комбинация слов: письмо, справка, отчет, книга, программа. Одно из главных средств автоматизации конторского труда.

24. *Машинная графика* — методы, служащие для преобразования данных в графическую форму. Графическое представление данных наиболее удобно для человека.

25. *Системы автоматизации проектирования (САПР)* — программные системы, которые используются для конструирования разнообразных изделий, от деталей машины до деталей интерьера.

26. *Автоматизированная система управления (АСУ)* — совокупность аппаратных и программных средств, обеспечивающих управление сложным объектом (отраслью, предприятием, технологическим процессом и т. д.).

27. *Искусственный интеллект (ИИ)* — научное направление, занимающееся созданием компьютерных систем, работа которых сходна с интеллектуальной деятельностью человека. К таким системам можно отнести: роботов, воспринимающих и анализирующих окружающую обстановку; программы, переводящие с одного естественного языка на другой; программы, играющие в шахматы; экспертные системы, обладающие знаниями в какой-либо области, например в медицине, и т. д.

28. *Робот* — машина с человекоподобным действием; предназначается для замены человека в трудных, неудобных для него условиях. Роботы широко используются в гибких (перестраиваемых) производственных системах (ГПС) и выполняют такие функции, как сварка, окраска, сборка и т. д. К одному из простейших видов роботов можно отнести станки с числовым программным управлением (ЧПУ).

29. *Пользователь* — человек, использующий ЭВМ в своей деятельности.

30. *Компьютерная грамотность* — наличие общего представления о вычислительных машинах и способах их использования, владение средствами информационно-поисковых систем, обработки текстов, проведения числовых расчетов.

СПИСОК ЛИТЕРАТУРЫ

1. Агеев М. И., Алиев В. П., Марков Ю. И. Справочное пособие.— М.: Сов. радио.— Вып. 2: Библиотека алгоритмов 516—1006, 1976.— С. 84—118; Вып. 3: Библиотека алгоритмов 1016—1506, 1978.— С. 92—99.
2. Алгоритм // Математическая энциклопедия. Т. 1.— М.: Сов. энциклопедия, 1977.
3. Брудно А. Л., Каплан Л. И. Олимпиады по программированию для школьников.— М.: Наука, 1985.
4. В мире науки.— 1984.— № 11.
5. Вирт Н. Систематическое программирование. Введение.— М.: Мир, 1977.
6. Вирт Н. Алгоритмы + структуры данных = программы.— М.: Мир, 1985.
7. Грис Д. Конструирование компиляторов для цифровых вычислительных машин.— М.: Мир, 1975.
8. Грогано П. Программирование на языке Паскаль.— М.: Мир, 1982.
9. Дал У., Дейкстра Э., Хоор К. Структурное программирование.— М.: Мир, 1975.
10. Йенсен К., Вирт Н. Паскаль.— М.: Финансы и статистика, 1982.
11. Квант.— 1981.— № 10—12; 1982.— № 1—4, 10—12; 1983.— № 1.— (Заочная школа программирования.)
12. Кнут Д. Искусство программирования для ЭВМ.— М.: Мир.— Т. 2: Получисленные алгоритмы, 1977; Т. 3: Сортировка и поиск, 1978.
13. Корн Г. А., Корн Т. М. Справочник по математике для научных работников и инженеров.— М.: Наука, 1984.— Разд. 20.7.
14. Мазуров О. Задача M764 // Квант.— 1982.— № 9.— С. 33.
15. Успенский В. А. Машина Поста.— М.: Наука, 1979.
16. Уэзерелл Ч. Этюды для программистов.— М.: Мир, 1982.
17. Хофштадтер Д. Р. Вокруг кубика Рубика: сферы, пирамиды, додекаэдр и бог знает, что еще // В мире науки.— 1983.— № 2.— С. 88.

Некоторые библиографические комментарии
к рекомендуемой литературе

Вирт Н. Систематическое программирование. Введение.— М.: Мир, 1977.

Вирт Н. Алгоритмы + структуры данных = программы.— М.: Мир, 1985.

Эти две книги представляют собой по существу две части единого учебника программирования, предназначенного для начинающих. Основное внимание уделяется не самому языку паскаль, а современным методам разработки программ и проверки их правильности. Способ изложения программирования в нашей книге близок к подходу Н. Вирта.

Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка.— М.: Финансы и статистика, 1982.

Книга состоит из двух частей. В первой части — руководство для пользователя — рассмотрено программирование на стандартном паскале и особенности версии Паскаль-6000, версия 3.4. Вторая часть — сообщение — содержит формальное описание языка паскаль. Книга полезна как справочник по паскалю.

Форсайт Р. Паскаль для всех.— М.: Машиностроение, 1986.

Довольно простой учебник программирования на паскале. Содержит большое число примеров программ, в том числе игровых.

Грехэм Р. Практический курс языка паскаль для микро-ЭВМ.— М.: Радио и связь, 1986.

Излагается язык паскаль и его использование на микро-ЭВМ Commodore и Apple. Книга рассчитана на подготовленных программистов.

Прайс Д. Программирование на языке паскаль: практическое руководство.— М.: Мир, 1987.

Описывается версия паскаля UCSD-Pascal (язык паскаль Калифорнийского университета). Помимо собственно языка рассматривается ряд вопросов программирования, например структурное программирование.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Алгоритм 7
 - , анализ 63
 - Евклида 7
 - , свойства 8, 9
- Алгоритмически неразрешимые проблемы 15
- База данных 173
- Буферная переменная файла 118
- Ветвь дерева 146
- Волшебный квадрат 55
- Вызов процедуры (функции) 50
- Вычислительный эксперимент 172
- Глобальное имя 51
- Дерево 106
 - бинарное 146, 184
- Динамическая переменная 136
- Динамическое программирование 107
- Дискриминант (записи) 126
- Запись 122
 - с вариантами 126, 144
- Имя 20
- Инвариант цикла 38
- Инструкция 19
 - ввода-вывода 23
 - выбора 113
 - перехода 105
 - присваивания 21
 - условная 27
 - цикла 30, 31, 45
 - WITH 125
- Ключевое слово 20
- Команда 9, 19
- Комментарий 29
- Компоненты записи 122, 123
- Корень дерева 106, 146
- Лексикографический порядок 89, 103
- Локальное имя 51
- Массив 43, 81, 91, 116
- Математическое моделирование 172
- Машинная графика 171
- Метка 105
- Микропроцессор 170
- Множество 114
- Начальная установка 33
- Обработка последовательностей 32
 - рекуррентных последовательностей 72
 - текстол 79, 115, 173
- Обход дерева 146, 151, 152, 159
- Оператор 19
- Описание меток 105
 - переменных 21
- Определение констант 41
- типов 42, 43, 111, 114, 115, 117, 122
- Оптимизация программы по времени выполнения (упрощение циклов) 77
- — по памяти (понижение размерности массивов) 81
- Отладка программ 97
- Параметр-переменная 52
- Параметр фактический 50
 - формальный 49
- Паскаль-машина 19
- Паскаль-программа 19, 20
- Перебор полный 100

Перебора сокращение 107
Переменная 19
Персональная ЭВМ 170, 173
Погрешность машинных вычислений 129
Поиск путем сравнений 78, 150
— с вставкой 150
Поста машина 9
Поста предложение 13
Пошаговая разработка программ 56
Представление чисел в ЭВМ 129
Принцип оптимальности Беллмана 107
Процедура 48
— как параметр 159
Результат функции 49
Рекуррентная последовательность 72, 75
Рекурсивные процедуры 152
Сеть ЭВМ 170
Синтаксическая диаграмма 175
Системы автоматизации проектирования 171
—числения 69
Слияние упорядоченных массивов 91
— упорядоченных файлов 120
Совершенное число 55
Сортировка 146
Список кольцевой 138, 141

Список линейный 138—142
Стандартные функции 22
Супер-ЭВМ 170

Тестирование 94
Тип вещественный (REAL) 19, 21, 129
—литерный (CHAR) 79
—логический (BOOLEAN) 46
—целый (INTEGER) 37
Типы, заданные перечислением 111
—отрезочные 42
Треугольник Паскаля 82

Узел дерева 146
Указатель 135
Унакованная структура данных 115
Условие 28, 46

Файл 117
Функция 48

Ханойская башня 155

Цикл 30
—внутренний 65

Числа Фибоначчи 72, 74

Экспертная система 172
Электронный бланк 173

*Вьюкова Надежда Ивановна
Галатенко Владимир Антонович
Ходулев Андрей Борисович*

**СИСТЕМАТИЧЕСКИЙ ПОДХОД
К ПРОГРАММИРОВАНИЮ**

Серия «Библиотечка программиста», вып. 53

Редакторы *Л. Г. Полякова, Д. С. Фурманов*

Художественный редактор *Г. М. Коровина*

Технический редактор *И. Ш. Аксельрод*

Корректор *М. Н. Дронова*

ИБ № 32550

Сдано в набор 19.10.87. Подписано к печати 20.04.88. Т-09574. Формат 84×108 1/32. Бумага тип. № 2. Гарнитура обыкновенная. Печать высокая. Усл. печ. л. 10,92. Усл. кр.-отт. 11,13. Уч.-изд. л. 13,99. Тираж 100 000 экз. Заказ № 8—334. Цена 85 коп.

Ордена Трудового Красного Знамени издательство «Наука»
Главная редакция физико-математической литературы. 117071 Москва В-71.
Ленинский проспект, 15

Набор и матрицы изготовлены в ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО «Первая образцовая типография» имени А. А. Жданова Союзполиграфпрома при Госкомиздате СССР. 113054 Москва, ул. Валуевская, 28.

Отпечатано на полиграфкомбинате ЦК ЛКСМ Украины «Молодь» ордена Трудового Красного Знамени ИПО ЦК ВЛКСМ «Молодая гвардия». 252119, Киев, ул. Пархоменко, 38—44.

Vjukova N. I., Galatenko V. A., Hodulev A. B.
SYSTEMATIC APPROACH TO PROGRAMMING/
Y. M. Bayakovsky, ed.

Moscow, Nauka, Main Editorial Board for Physical and
Mathematical Literature, 1988, 208 pp.

This book is an introductory course of programming. It is intended for those wishing for professional knowledge of programming. The main goal of the book is to teach systematic methods of program development. Attention is paid to mathematical analysis of programs, stepwise refinement, program correctness proof, and algorithm analysis which are the basic components of systematic approach. We used PASCAL as a programming language throughout the book for its wide appreciation in both teaching practice and programming activity.

The book is concluded with a quick reference on some PASCAL compilers and specific features of Turbo PASCAL system for IBM/PC and compatibles. There are also a commented list of recommended books on PASCAL and a short thesaurus of the thirty most usable computer science terms.