С. А. АБРАМОВ, Е. В. ЗИМА

НАЧАЛА ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПАСКАЛЬ

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1987

ББК 22.18
А16
УДК 519.6

Рецензент
кандидат физико-математических наук *Н. П. Трифонов*

Абрамов С. А., Зима Е. В.

А 16 Начала программирования на языке паскаль.— М.: Наука. Гл. ред. физ.-мат. лит., 1987. — 112 с.

Предлагается сокращенный вариант языка программирования паскаль. Всякая программа, правильно написанная на сокращенном паскале, будет правильной в смысле полного паскаля.

Понятия языка и приемы программирования на нем излагаются таким образом, чтобы с первых же параграфов привлечь читателя к самостоятельному составлению законченных программ. Разбирается большое число примеров и предлагаются задачи для самостоятельного решения.

Для учащихся средних школ, ПТУ, техникумов и для начинающих программистов.

Табл. 2. Ил. 36.

А $\frac{1702070000-038}{053\,(02)-87}$ 37-87

ББК 22.18

© Издательство «Наука». Главная редакция физико-математической литературы, 1987

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В этой книге предлагается сокращенный вариант языка паскаль. Всякая программа, правильно написанная на сокращенном паскале, будет правильной в смысле полного паскаля. Сокращенным вариантом можно овладеть быстрее и легче, чем всем языком; возможностей же предлагаемого сокращенного варианта вполне достаточно для решения не слишком сложных задач обработки информации.

Самое первое знакомство с программированием на паскале может даже быть сведено к овладению материалом, который содержится в главах I, II и в двух первых параграфах главы III книги. Этот материал доступен учащимся школ и ПТУ. Учащиеся школ с углубленным изучением математики и студенты техникумов смогут проработать до конца главу III, а также главу IV. Глава V может рассматриваться как дополнительная.

Всюду в тексте слово «паскаль» употребляется как название сокращенного варианта языка, слово «программа» означает программу на паскале.

ВВЕДЕНИЕ

Последнее время в разговорной речи и в журнальном лексиконе всё более укореняется слово *алгоритм*, под которым в большинстве случаев понимается рецепт достижения некоторой цели. Говорят, например, об алгоритме перехода улицы. В математике и в информатике в термин «алгоритм» вкладывается более конкретное содержание. Так, под числовым алгоритмом понимается детально описанный способ получения одних чисел (результатов), исходя из других (исходных данных), с помощью математических операций. Можно говорить, в этом смысле, об алгоритме нахождения корней квадратного уравнения, заданного своими коэффициентами, или об алгоритме разложения натурального числа на простые множители с помощью основных арифметических операций. В общем случае исходные данные и результаты должны принадлежать некоторому множеству, над элементами которого можно выполнять определенные операции. Например, довольно часто в роли исходных данных и результатов выступают не числа, а последовательности символов — тексты, формулы и т. д., в роли операций — не операции сложения, умножения и подобные им, а операции табличной замены символов на другие символы, приписывания одной последовательности к другой и т. д. Примером может служить алгоритм преобразования текста в его код Морзе. Чуть позже мы упомянем пример значительно более сложного нечислового алгоритма.

Поиски различных алгоритмов входили в круг важных научных задач во всё время существования науки. Уже в древнейшие времена были получены методы нахождения площадей и объемов геометрических фигур и тел. Одним из достижений античной науки было изобретение Евклидом (III век до н. э.) необычайно остроумного способа быстрого нахождения наибольшего общего делителя двух натуральных чисел. Математики Древнего

Востока изобрели десятичную систему и дали правила вычислений в этой системе. Сам термин «алгоритм» тоже имеет древнее происхождение, являясь латинизированной транскрипцией имени великого среднеазиатского ученого IX века Мухаммеда аль-Хорезми (буквально: Мухаммеда из Хорезма или Мухаммеда Хорезмского; Хорезм — название древнего государства на территории Узбекистана). В математическом трактате Мухаммеда аль-Хорезми формулировались, в частности, правила разнообразных вычислений.

В настоящее время интерес к алгоритмам особенно велик благодаря возможности использования в научных исследованиях, в технике, экономике и т. д. вычислительных машин, выполняющих построение некоторых величин в точном соответствии с указанным алгоритмом. Эта возможность привлекательна по той причине, что явления и процессы, которые изучаются в рамках упомянутых исследований, часто удается описать с помощью понятий математики — функций, систем уравнений, неравенств и т. д. — и для получения конкретных сведений об изучаемых явлениях и процессах надо произвести некоторые действия над математическими объектами. Человеку достаточно описать алгоритм необходимых преобразований и вычислений, а сами действия (как правило, настолько обширные и громоздкие, что их невозможно выполнить вручную) выполнит вычислительная машина.

Не следует думать, что вычислительные машины выполняют только числовые алгоритмы: эти машины могут преобразовывать и получать последовательности символов — алгебраические формулы, тексты и т. д. (числа, заданные наборами цифр, — частный случай последовательностей символов).

Правила записи алгоритма для выполнения его вычислительной машиной оказываются очень жесткими — автомат не может ничего додумывать за человека. Совокупность средств и правил представления алгоритма в виде, пригодном для выполнения вычислительной машиной, называется *языком программирования*, а каждый алгоритм, записанный на некотором языке программирования, называется *программой*. Для удобства составления программ предлагаются различные языки программирования; это ставит задачу разработки алгоритмов перевода программ на этих языках в такие программы, которые понятны вычислительной машине в силу ее кон-

струкции. Алгоритм перевода записывается в виде программы, после чего перевод может выполнить сама вычислительная машина. Алгоритм перевода с одного языка программирования на другой — пример очень сложного нечислового алгоритма. Методы создания таких алгоритмов рассматриваются специальной отраслью информатики — системным программированием.

Из распространенных языков программирования в последнее десятилетие выделился своей популярностью язык паскаль, созданный в 70-х годах швейцарским ученым Н. Виртом. В паскале сконцентрированы многие лучшие черты языков-предшественников. Сравнение этого языка с более простыми языками вроде бейсика (с которым теперь знакомят школьников в курсе основ информатики и вычислительной техники) показывает, что текст алгоритма в виде программы на бейсике содержит больше подробностей о действиях вычислительной машины, а текст на паскале яснее выражает математическую сущность алгоритма. Паскаль в сравнении с бейсиком, как принято сейчас говорить, является языком более высокого уровня.

Можно надеяться, что читатель оценит достоинства паскаля уже после нескольких первых шагов программирования на нем.

Глава I

ОСНОВНЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ

§ 1. О записи программы. Выражения

Программа записывается в виде последовательности символов, к числу которых относятся латинские и русские буквы, арабские цифры, знаки препинания, знаки операций.

Для обозначения исходных данных и результатов вычислений употребляются *переменные*, которыми могут быть не только любые буквы — *a*, *b*, *X*, *Y*, *ш*, *щ* и т. д., но и, например, последовательности символов вида *x*1, *x*2, *time*, *Сила*, *alfa*100, *a*1*a*2 и т. д., которые состоят из букв и цифр и начинаются с буквы. Соответствующее исходное данное или результат вычисления называется *значением переменной*. Пока мы будем иметь дело с переменными, значениями которых являются числа. Числа в программе записываются в десятичной системе, вместо запятой пишется точка: 0, —17, 0.26, 3.1415, +1.567, —0.18 и т. д. Количество цифр в числе не может быть слишком большим; граница для этого количества в паскале не оговаривается — она определяется характеристиками используемой вычислительной машины. Это относится и к количеству букв и цифр в переменной.

Переменные и числа — простейшие частные случаи *выражения*. Более сложные выражения строятся из чисел и переменных с помощью знаков операций сложения, вычитания, умножения и деления. Эти знаки суть +, —, *, /. Кроме того, в выражении могут быть использованы круглые скобки и некоторые функции. Знак операции деления / позволяет записывать в строку выражения, которые традиционно записываются с выходом из строки: в паскале пишут a/b, $c/17$, $(a*x+b)/(c+d)$ и т. д. Знак операции умножения * нельзя опускать или заменять точкой. Допустимое для математического текста выражение $0{,}5(x+7)\cdot(x+2)\cdot(x-3)$ в паскале должно быть записано в виде $0.5*(x+7)*(x+2)*(x-3)$. Знак — (минус) может употребляться и для изображе-

ния величины, противоположной данной: $-x$, $-(a*b+y)$ и т. д. Нельзя размещать два знака операций рядом; последовательности символов $3*-2$, $x1/-x2$ — это не выражения, выражениями будут $3*(-2)$, $x1/(-x2)$.

В выражении могут быть использованы следующие функции:

sin(*E*) — синус *E*,
cos(*E*) — косинус *E*,
arctan(*E*) — главное значение арктангенса *E*,
ln(*E*) — натуральный логарифм *E*,
exp(*E*) — показательная функция *E*, т. е. e^E,
abs(*E*) — абсолютная величина (модуль) *E*, т. е. $|E|$,
sqr(*E*) — квадрат (вторая степень) *E*, т. е. E^2,
sqrt(*E*) — квадратный корень из *E*, т. е. $\sqrt{E}$;

выражение, задающее аргумент, всегда заключается в скобки. Так, например, мы пишем *sqrt*(*sqr*(*b*) — 4**a***c*).

В ходе выполнения программы вычисляются значения выражений. При вычислении действуют обычные правила старшинства операций: старшие операции — умножение и деление, следующие по старшинству — сложение и вычитание. Из двух операций одинакового старшинства первой выполняется та, знак которой в выражении встречается раньше. Круглые скобки изменяют этот естественный порядок: значением выражения $(x+y)/2$ будет половина суммы значений переменных x и y, в то же время значением выражения $x+y/2$ будет сумма значения x и половины значения y.

Последовательность букв и цифр, начинающаяся с буквы, называется *идентификатором*. Из рассмотренных примеров видно, что идентификатор не обязательно представляет собой переменную: *sin*, *cos*, *arctan* и т. д. — это не переменные, а *имена функций*. В программах встречаются и другие виды идентификаторов.

ЗАДАЧИ

1. Какие из следующих последовательностей символов являются идентификаторами?

a^3, *a*3, $a \cdot 3$, x_3, 100*d*, *d*100, *max*, α,
alfa, *sin*, *sinx*, *sin*(*x*), *sin*1*sin*.

2. Записать по правилам паскаля выражение

$$0{,}3\left(\left(\frac{\sin^2 x-\cos^2 x}{\sin\frac{x+y}{2}}-e^{|\cos x+\sin x|}\right)\ln x-\sqrt{x-1}\right).$$

3. Записать по традиционным правилам выражение

$(-b - sqrt(sqr(b) - 4 * a * c))/(2 * a)$.

4. Вычислить значение выражения

$(sin(sqr(x) - 1) + 2 * abs(y))/cos(2 + y)$

при следующих значениях переменных: $x = 1$, $y = -2$.

5. В выражении

$a/b * c/d * e/f * h$

расставить скобки так, чтобы выражению со скобками соответствовала дробь

$$\frac{a}{b \cdot \dfrac{c}{d \cdot \dfrac{e}{f \cdot h}}}.$$

§ 2. Операторы присваивания, ввода и вывода

Отдельные инструкции, входящие в программу, в паскале принято называть *операторами.*

В результате выполнения *оператора присваивания* переменной присваивается значение некоторого выражения. Примеры операторов присваивания:

$a := 0$

$b := c$

$x1 := (-b + sqrt(sqr(b) - 4 * a * c))/(2 * a)$

$x := x + 1$

Во всех случаях вначале вычисляется значение выражения, расположенного справа от комбинации символов :=, а затем вычисленное значение присваивается переменной, расположенной слева. Для того чтобы оператор присваивания мог быть выполнен, необходимо, чтобы все переменные, которые входят в выражение, имели некоторые значения. Последний из приведенных выше операторов присваивания предписывает увеличение значения x на единицу. Если до выполнения оператора $x := x + 1$ переменная x имела значение 1.3, то после выполнения этого оператора значение x станет равным 2.3. В результате выполнения таких операторов переменные в ходе выполнения программы могут изменять свои значения.

Для ввода данных и вывода результатов используются *операторы ввода* и *вывода*. Вот как, например, они могут выглядеть:

read (*x*)

read (*x*1, *x*2, *y*)

write (*x*)

write (*x*, *y*, *z*, *p*, *q*, *r*)

Оператор ввода состоит из идентификатора *read* (read по-английски означает читать) и следующего за ним в круглых скобках списка переменных. Число переменных в списке может быть любым; если переменных больше одной, то они разделяются запятыми. Оператор вывода состоит из идентификатора *write* (write — означает писать) и следующего за ним в круглых скобках списка переменных.

При выполнении оператора ввода переменным присваиваются значения исходных данных. Программу, содержащую операторы ввода, необходимо снабдить приложением — конечной последовательностью чисел. Когда выполняется оператор ввода, переменным присваиваются очередные значения. Пусть приложение состоит из чисел 3.5, 8.2 и —1.1. Тогда после выполнения оператора *read*(*a*) переменная *a* получит значение 3.5. Если же через некоторое время будет выполнен оператор *read*(*b*, *c*), то *b* получит значение 8.2, *c* — значение —1.1.

Привлечение оператора ввода открывает возможность многократного использования одной и той же программы для вычислений с различными исходными данными. Надо будет менять только упомянутое приложение к программе — конечную последовательность чисел.

С помощью оператора вывода строится последовательность значений, которая является результатом выполнения программы. Пусть был выполнен оператор *write*(*x*), и значение переменной *x* в момент выполнения равнялось 0.1. Пусть через некоторое время был выполнен оператор *write* (*z*, *t*), и в момент выполнения этого оператора значение *z* было равно 3.5, а значение *t* было равно —1.08. Если больше никаких операторов вывода не выполнялось, то результатом выполнения всей программы будет последовательность, состоящая из чисел 0.1, 3.5 и —1.08. Эта последовательность будет напечатана вычислительной машиной на бумаге или высвечена на экране.

Оператор вывода, таким образом, позволяет выделить из всего набора вычисленных значений те, которые служат ответом к решавшейся при выполнении программы задаче.

Приведем последовательность операторов, предназначенную для ввода трех чисел и вывода их суммы; друг от друга операторы отделяются точкой с запятой:

$read(a, b, c)$; $x := a + b + c$; $write(x)$

В результате выполнения оператора $read(a, b, c)$ переменным a, b и c присваиваются значения, равные соответственно первому, второму и третьему из данных чисел. После выполнения оператора $x := a + b + c$ переменная x получает значение, равное сумме значений переменных a, b и c. Выполнение оператора $write(x)$ приводит к выводу вычисленного значения переменной x.

При решении рассмотренной задачи можно было обойтись меньшим количеством переменных:

$read(a, b)$; $a := a + b$; $read(b)$;

$a := a + b$; $write(a)$

Здесь после выполнения первых двух операторов значение переменной a будет равняться сумме первых двух данных чисел. После выполнения оператора $read(b)$ значение переменной b станет равным третьему из данных чисел. В результате выполнения четвертого из приведенных операторов переменной a присвоится значение, равное сумме ее предыдущего значения и значения переменной b.

ЗАДАЧИ

1. Какие значения будут выведены в результате выполнения последовательности операторов?

$x := (sin(sqr(1) - 1) + 2 * abs(-2))/cos(2 - 2)$;

$y := x * (sqr(2))$; $write(x, y)$

2. Какое значение будет выведено в результате выполнения последовательности операторов

$read(x, y)$; $z := sqr(sqr(x)) * sqrt(y)$;

$write(z)$

если последовательность исходных данных была составлена из двух чисел: 0.5 и 0.16?

3. Какие значения будут иметь переменные x и y после выполнения последовательности операторов

$$read(x, y);\quad t := x;\quad x := y;\quad y := t$$

если последовательность исходных данных была составлена из двух чисел: 5.2 и 18.7?

4. Какие значения будут выведены в результате выполнения последовательности операторов

$$read(x, y);\quad x := x + y;\quad y := x - y;$$
$$x := x - y;\quad write(x, y)$$

если последовательность исходных данных была составлена из двух чисел:

а) 3.5 и 2.4;

б) 6.7 и —10.1?

5. Написать последовательность операторов, предназначенную для ввода двух данных чисел и последующего вычисления и вывода их среднего арифметического и среднего геометрического (предполагается, что данные числа неотрицательны).

§ 3. Простейшая программа

Во всех рассмотренных ранее примерах переменные принимают значения в множестве действительных чисел, такие переменные называются переменными типа *real* (real по-английски — действительный).

Кроме операторов, программа содержит описания переменных. Примеры описаний переменных:

$$x : real$$
$$y, z, t : real$$

После списка переменных идет двоеточие и идентификатор *real*. Каждая переменная должна быть описана, т. е. включена в описание. Все описания переменных соединяются вместе: **var** A_1; ...; A_k;, т. е. после **var** идет некоторое количество описаний переменных, отделенных друг от друга точкой с запятой. Пока в наших програм-

мах будет преимущественно по одному описанию. Мы будем писать

var x: *real*;

или

var y, z, t: *real*;

и т. д.

Отметим, что **var** относится к так называемым служебным словам, назначение которых жестко зафиксировано в паскале; эти служебные слова нельзя, например, использовать в качестве переменных. Этим они отличаются от идентификаторов *sin*, *read*, *write* и т. д., которым программист может, в принципе, придавать смысл, отличный от того, которым они изначально наделены в паскале (хотя делать это не рекомендуется). Служебные слова в печатном тексте набираются жирным шрифтом, а в рукописном тексте подчеркиваются; **var** является сокращением английского слова variable — переменная (далее переводы служебных слов и некоторых других «осмысленных» идентификаторов будут даваться в подстрочных примечаниях без указания на то, что слова английские).

Простейшая программа на паскале схематически может быть изображена следующим образом:

program N (...);
var A_1; ...; A_k;
begin P_1; ...; P_n **end.** *)

Здесь N — имя программы (идентификатор), A_1, ..., A_k — описания переменных, P_1, ..., P_n — операторы. Вместо многоточия внутри скобок надо поместить либо идентификатор *input*, либо *output*, либо оба эти идентификатора через запятую: *input*, *output* **). Идентификатор *input* означает, что в программе встретится оператор ввода, *output* — оператор вывода.

Операторы P_1, ..., P_n выполняются в порядке следования друг за другом.

Пример. Программа *корни*1 вычисления корней квадратного уравнения $ax^2 + bx + c = 0$, заданного ко-

*) program — программа, begin — начало, end — конец.
**) input — ввод, output — вывод.

эффициентами a, b и c (предполагается, что $a \neq 0$ и что корни действительные)*):

```
program корни1 (input, output);
 var a, b, c, x1, x2 : real;
 begin read(a, b, c);
   x1 := (- b + sqrt(sqr(b) - 4 * a * c))/(2 * a);
   x2 := (- b - sqrt(sqr(b) - 4 * a * c))/(2 * a);
   write(x1, x2)
 end.
```

При выполнении этой программы придется дважды вычислить значение $sqrt(sqr(b) - 4 * a * c)$ и дважды — значение $2 * a$. Поэтому более разумным вариантом будет программа *корни2*:

```
program корни2 (input, output);
 var a, b, c, x1, x2, d, e : real;
 begin read(a, b, c);
   d := sqrt(sqr(b) - 4 * a * c); e := 2 * a;
   x1 := (- b + d)/e; x2 := (- b - d)/e;
   write(x1, x2)
 end.
```

Если это почему-либо удобно, можно одно длинное описание заменить совокупностью более коротких. В программе *корни2* можно было бы написать, например,

```
var a, b, c : real; x1, x2 : real; d, e : real;
```

Важно лишь, чтобы каждая переменная была описана и входила только в одно описание.

Еще один пример. Программа вычисления площади треугольника по трем сторонам a, b, c:

```
program площадь (input, output);
 var a, b, c, p, s : real;
 begin read(a, b, c);
   p := (a + b + c)/2;
   s := sqrt(p * (p - a) * (p - b) * (p - c));
   write(s)
 end.
```

*) Программа — это последовательность символов. При записи программы мы из соображений удобства разбиваем ее на строки. Это разбиение является условным, никакие знаки переноса не используются.

Видно, что площадь вычисляется по формуле Герона.

При выполнении программы вычисления могут проводиться приближенно: числа в паскале имеют конечное число знаков после десятичной точки, и округления неизбежно возникают, например, при вычислении значений *sqrt*(2), *sin*(1) и т. д.

После того как программа написана и для нее выбраны исходные данные, нужно, чтобы эта программа была выполнена. Эту работу обычно возлагают на вычислительную машину. Возникает следующее взаимодействие четырех основных компонент вычислительной машины — устройства ввода, устройства вывода, памяти и процессора. На клавиатуре устройства ввода, напоминающей клавиатуру пишущей машинки, набирается программа и исходные данные, в результате чего и то и другое попадает в память вычислительной машины. Процессор выполняет программу. Результаты поступают в устройство вывода: дисплей (телевизионный экран, на котором высвечиваются результаты вычислений) или принтер (устройство, печатающее результаты на бумаге).

Между вводом программы в память и выполнением операторов программы имеется подготовительный этап работы процессора, в течение которого, в частности, рассматриваются содержащиеся перед операторами описания переменных и отводится место в памяти для размещения значений каждой из переменных. Всякий раз, когда позднее во время выполнения программы некоторая переменная будет получать значение, это значение будет помещаться в закрепленное за данной переменной место в памяти, а предыдущее значение этой переменной будет стираться. Если же переменная встретится в выражении, стоящем в правой части оператора присваивания, то при вычислении значения выражения произойдет обращение за значением этой переменной в отведенное ей место памяти.

ЗАДАЧИ

1. Написать программу вычисления суммы и произведения двух данных чисел.
2. Написать программу нахождения гипотенузы и площади прямоугольного треугольника по двум данным катетам.

3. Смешано v_1 литров воды температуры t_1 с v_2 литрами воды температуры t_2. Написать программу вычисления объема и температуры образовавшейся смеси.

4. Написать программу вычисления a^{10}, где a — данное число. Если получившаяся программа требует более четырех умножений (использование функции *sqr* тоже считается умножением), то попытаться дать более экономное решение.

5. Написать программу вычисления значений $1 - 2x + 3x^2 - 4x^3$ и $1 + 2x + 3x^2 + 4x^3$, где x — данное число. Позаботиться об экономии операций.

§ 4. Условный и составной операторы

Если мы хотим, чтобы переменной *max* присвоилось наибольшее из значений переменных $x1$ и $x2$, то надо сравнить значения $x1$ и $x2$ и в зависимости от результата сравнения выполнить либо оператор $max := x1$, либо $max := x2$. Действия такого рода задаются *условным оператором*:

if B **then** P_1 **else** P_2 *),

где B — условие, P_1 и P_2 — операторы. Если условие B удовлетворяется, то выполняется P_1, иначе выполняется P_2.

В качестве условий используются отношения. Отношения представляют собой записи равенств и неравенств. Примеры отношений:

$a = b$

$d \neq 0$

$sqr(b) - 4 * a * c > 0$

В общем случае отношение — это два выражения, разделенных одним из знаков $=$, $\neq$, $\geqslant$, $>$, $<$, $\leqslant$ **). Для решения задачи о присваивании переменной *max* наибольшего из значений $x1$ и $x2$ достаточно выполнить условный оператор

if $x1 > x2$ **then** $max := x1$ **else** $max := x2$

*) if — если, then — то, else — иначе.

**) На некоторых вычислительных машинах нет знаков $\neq$, $\geqslant$, $\leqslant$, и вместо этих знаков используются соответственно комбинации $<>$, $>=$, $<=$.

Оператор, расположенный после **else**, может быть любым оператором; оператор, расположенный между **then** и **else**, не может быть условным.

Рассмотрим примеры программ, включающих в себя условные операторы. Пусть задана функция

$$y = \begin{cases} 0, & \text{если } x \leqslant 0; \\ x^3, & \text{если } x > 0. \end{cases}$$

Требуется написать программу F вычисления значения y по значению x. Программа может выглядеть так:

```
program F (input, output);
 var x, y : real;
 begin read (x);
   if x ⩽ 0 then y := 0 else y := x * x * x;
   write (y)
 end.
```

Пример. Пусть значение y зависит от значения x, график зависимости приведен на рис. 1. Программа G вычисления значения y по значению x:

```
program G (input, output);
 var x, y : real;
 begin read (x);
   if x < 2 then y := x else
     if x < 3 then y := 2 else y := - x + 5;
   write (y)
 end.
```

Если удовлетворено условие $x < 2$, то y получит значение, равное значению x, и это значение затем будет выведено. Если условие $x < 2$ не удовлетворяется, то значение y будет определено выполнением условного оператора **if** $x < 3$ **then** $y := 2$ **else** $y := -x + 5$.

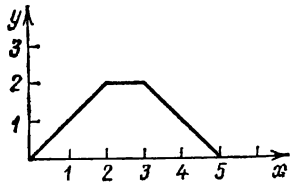


Рис. 1

Допускается сокращенная форма условного оператора:

if B **then** P

где B — условие, P — оператор, не являющийся условным. В случае, если B удовлетворено, должен быть выполнен оператор P; если же B не удовлетворено, его выполнять не нужно.

Пример. Пусть даны два числа. Если первое больше второго по абсолютной величине, то необходимо уменьшить первое в пять раз. Иначе оставить числа без изменения. Программа:

```
program N (input, output);
 var x, y : real;
 begin read (x, y);
   if abs (x) > abs (y) then x := x/5;
   write (x, y)
 end.
```

В условном операторе после **then** и **else** можно помещать по одному оператору. Однако часто необходимо в зависимости от результата проверки выполнить ту или иную группу операторов. Паскаль предоставляет возможность сделать из группы операторов один составной оператор. Структура *составного оператора*:

begin P_1; P_2; ...; P_K **end**

где P_1, P_2, ..., P_K — любые операторы. Как частный случай составного оператора имеем **begin** P **end**, где P — любой оператор. Существенно, что оператор **begin** P **end** при любом операторе P не является условным и может быть размещен после **then**.

Пусть квадратное уравнение $ax^2 + bx + c = 0$ задано коэффициентами a, b и c ($a \neq 0$).

Ниже следует программа, при выполнении которой исследуется дискриминант уравнения, выводится число 0, если дискриминант отрицателен, и выводится пара корней, если дискриминант неотрицателен:

```
program корни4 (input, output);
 var a, b, c, x1, x2, d : real;
 begin read (a, b, c);
   d := sqr (b) - 4 * a * c;
   if d < 0 then begin
                   d := 0; write (d)
                 end
```

```
    else begin
           d := sqrt(d);
           a := 2 * a;
           x1 := (- b + d)/a;
           x2 := (- b - d)/a;
           write(x1, x2)
         end
end.
```

В этой программе в условном операторе после **then** и **else** размещены составные операторы.

Если корней нет, то выполнение программы *корни4* приводит к выводу числа 0. Есть, однако, возможность выдать сообщение об отсутствии корней в словесной форме: для этого достаточно вместо составного оператора, расположенного после **then**, в программе *корни4* поместить оператор **write** (*'корней нет'*).

В результате выполнения этого оператора на бумаге или на экране появится следующий текст: *корней нет.*

Вообще, в операторе вывода могут указываться не только переменные, но и произвольные выражения (будет вычислено и выведено значение каждого выражения), а также любые тексты, каждый из которых ограничен с обеих сторон знаком ' (штрих) (будет выведен сам текст без штрихов). Так, если второй оператор вывода в программе *корни4* заменить на *write*('x1 =',x1,', x2 =',x2), то выполнение программы при наличии корней будет приводить к печати на бумаге или к высвечиванию на экране легко читаемой информации

$x1 = \ldots, x2 = \ldots$

(вместо многоточий появятся, разумеется, настоящие значения корней). Возможность использования выражений в операторе вывода часто позволяет исключать из программы некоторые переменные:

```
program корни5(input, output);
 var a, b, c, d : real;
 begin read(a, b, c); d := b * b - 4 * a * c;
   if d < 0 then write('корней нет')
```

```
        else begin
                d := sqrt(d);
                a := 2 * a;
                write('x1=', (-b + d)/a,
                      ',x2=', (-b - d)/a)
             end
end.
```

В паскаль дополнительно включен оператор вывода *writeln*, который выполняется так же, как *write*, с той разницей, что после его выполнения происходит переход на новую строку листа бумаги или экрана.

Пробел — это тоже, фактически, печатный знак (символ). В тексте *корней нет* присутствует пробел. На клавиатуре, с которой программа передается вычислительной машине, есть клавиша пробела. Для того чтобы явственно обозначить пробел в рукописном тексте программы, часто используют специальный знак ␣. Специальный знак ␣ особенно удобен, если надо указать несколько пробелов, идущих подряд:

write('x1=', x1, ', ␣␣␣ x2=', x2)

Переход на другую строку, в соответствии со сказанным, может быть предписан оператором *writeln*(' ') или, более четко, *writeln*('␣') — напечатать пробел, т. е. ничего не напечатать, и перейти на другую строку.

Пропуски строк и выводы групп пробелов широко используются при оформлении выводимой информации в виде таблиц и графиков.

Служебные слова, встречающиеся в программе, при вводе в вычислительную машину отделяются пробелом от соседних с ними идентификаторов (в частности, от других служебных слов) и чисел: *if* ␣ *d* < 0 ␣ *then*␣ *write*('*корней* ␣ *нет*') и т. д. В рукописном или печатном тексте знаки этих пробелов не ставят, так как служебные слова там выделены подчеркиванием или жирным шрифтом и не сливаются с соседними символами.

ЗАДАЧИ

1. Написать программу вычисления значения функции

$$y = \begin{cases} x^2, & \text{если } -2 \leqslant x \leqslant 2; \\ 4 & \text{в остальных случаях.} \end{cases}$$

2. Написать программу выбора наибольшего из трех чисел.

3. Условный оператор

if $sqr(x) > 2$ **then**

begin if $x > 2$ **then** $y := x * x * x$ **else** $y := 8$ **end**

else $y := 8 * sqr(x)$

устанавливает зависимость значения y от значения x. Построить график этой зависимости.

4. Написать программу полного исследования совокупности корней биквадратного уравнения $ax^4 + bx^2 + c = 0$. Если корней нет, то должно быть выведено сообщение об этом, иначе должны быть выведены два или четыре корня.

5. Как надо изменить последний оператор вывода в программе *корни5*, чтобы результаты печатались или высвечивались в виде

$x_1 = \ldots$

$x_2 = \ldots$

(вместо многоточий должны быть настоящие значения корней)?

§ 5. Оператор цикла

Многократно повторяемые действия могут быть заданы *оператором цикла*

while B **do** P *),

где B — условие (отношение), P — оператор (называемый телом цикла). Выполняется выписанный оператор цикла так: проверяется условие B, и если оно удовлетворено, то выполняется P, а затем вновь проверяется условие B и т. д. Как только на очередном шаге окажется, что условие B не удовлетворяется, то выполнение оператора цикла прекратится.

Если значение x положительно, то выполнение оператора цикла

while $x \leqslant 0$ **do** $x := x + 1$

*) while — пока, do — выполнять.

прекратится после первой же проверки условия $x \leqslant 0$, и значение переменной x не изменится. Если же значение x не положительно, то к этому значению будет добавляться по единице до тех пор, пока значение не станет положительным.

Пусть даны числа a, b $(a > 1)$ и надо получить все члены бесконечной последовательности a, a^2, a^3, ..., меньшие числа b. Программа:

```
program x(input, output);
 var a, b, c : real;
 begin read(a, b); c:=a;
   while c < b do
     begin
       writeln(c); c:=c*a
     end
 end.
```

При выполнении этой программы переменная c последовательно принимает значения a, a^2, a^3, ... Изменение значения c происходит до тех пор, пока оно не станет больше или равно значению b. Если $a \geqslant b$, то не будет выведено ни одного члена последовательности a, a^2, a^3, ...

Пример. Программа приближенного вычисления суммы

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots$$

При суммировании учитываются только слагаемые, большие по абсолютной величине данного положительного числа ε. Программа вычисления этой суммы, на первый взгляд, должна предписывать поочередное получение слагаемых возведением x в соответствующую степень, вычислением факториала и делением одного значения на другое. Но надо стараться так применять оператор цикла, чтобы каждый следующий шаг максимально использовал сделанное на предыдущих шагах. Если уже получено $x^{i-1}/(i-1)!$, то для вычисления $x^i/i!$ достаточно домножить предыдущий результат на x/i.

Естественным будет следующее решение:

```
program Sum (input, output);
 var x, eps, i, y, s : real;
 begin read (x, eps);
   s := 0; y := 1; i := 0;
   while abs (y) > eps do
    begin
      s := s + y; i := i + 1;
      y := y * x/i
    end;
   write (s)
 end.
```

При выполнении этой программы переменная i последовательно принимает значения 0, 1, 2, ..., переменная y принимает значения $1, \frac{x}{1!}, \frac{x^2}{2!}, \ldots$, переменная s в свою очередь принимает значения $0, 1, 1+\frac{x}{1!}, 1+\frac{x}{1!}+\frac{x^2}{2!}, \ldots$

Один из наиболее трудных моментов при составлении циклических программ — это присваивание переменным нужных начальных значений перед выполнением оператора цикла. Эти значения требуют тщательного подбора.

ЗАДАЧИ

1. Написать программу нахождения среди чисел $1, 1+\frac{1}{2!}, 1+\frac{1}{2}+\frac{1}{3}, \ldots$ первого, большего данного числа a.

2. Дано положительное число ε. Последовательность $a_1, a_2, a_3, \ldots$ образована по следующему закону:

$$a_i = \left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\cdot\ldots\cdot\left(1-\frac{1}{i+1}\right).$$

Надо найти первый член a_n последовательности, для которого выполнено условие

$$|a_n - a_{n-1}| < \varepsilon.$$

Написать программу выполнения этого задания.

3. Пусть $0 \leq x \leq 1$. Написать программу вычисления суммы

$$x^{(1^2)} + x^{(2^2)} + x^{(3^2)} + \ldots;$$

учитываются только слагаемые, большие данного положительного числа ε.

Указание. Значение $x^{(i^2)}$ получается из $x^{((i-1)^2)}$ домножением на x^{2i-1}. Значение x^{2i-1} получается из $x^{2(i-1)-1}$ домножением на x^2.

4. Написать программу выполнения следующего задания. Найти сумму чисел, предшествующих первому отрицательному в последовательности данных чисел.

Указание. Должен многократно выполняться оператор ввода *read*(x).

5. Так же как в предыдущей задаче, рассмотрим все числа, предшествующие первому отрицательному в последовательности данных чисел. Написать программу, которая позволяет получить те из них, которые лежат в интервале от 1.5 до 2.

§ 6. Тип *integer*

В § 2 было приведено схематическое изображение программы. Тогда нам было известно всего лишь три вида операторов. В дальнейшем понятие оператора существенно расширилось, теперь же у нас появится новый вид описания переменных, в котором вместо *real* пишется *integer* *), например:

i, j : integer

В данном случае мы имеем дело с описанием переменных целого типа, или переменных типа *integer*. Применявшиеся ранее описания были описаниями переменных действительного типа, или переменных типа *real*.

Переменные типа *integer* могут принимать лишь целые значения, а переменные типа *real* — как целые, так и нецелые. Числа в паскале тоже разделяются на целые, или имеющие тип *integer*, и действительные, или имеющие тип *real*: если в записи числа использована точка, то оно действительное, в противном случае — целое.

Примеры целых чисел:

0, —3, 17, +193, —1000000, 5

*) integer — целый.

Примеры действительных чисел:
3.14, —0.0001, 0.6, 17.0, —19.19191919, 5.0

Объясним причину, по которой выделяются переменные и числа типа *integer*. Уже говорилось, что значения выражений, в которые входят переменные, числа типа *real* и некоторые функции, могут вычисляться приближенно, т. е. с небольшой погрешностью. Значение выражения $0.1 * 10$ может вычислиться, например, как 0.999999999999 или 1.000000000001. (Поэтому не имеет смысла проверять выполнение равенства между действительными значениями выражений.)

Арифметические операции сложения, вычитания и умножения над целыми числами производятся абсолютно точно, и результатами этих операций снова являются целые числа. Целочисленное значение выражения может быть сохранено в точном виде для дальнейших вычислений присваиванием этого значения переменной типа *integer*.

Программа вычисления $n!$:

```
program fact1 (input, output);
 var n, i, p : integer;
 begin read (n); p := 1; i := 0;
   while i < n do
      begin i := i + 1; p := p * i end;
   write ('n!=', p)
 end.
```

Здесь вычисление проводится в n шагов. Переменная p последовательно принимает значения $1, 1\cdot2, 1\cdot2\cdot3, \ldots, 1\cdot2\cdot\ldots\cdot n$. Если бы переменные n, i, p были описаны как переменные типа *real*, то после того как в произведение вошло n сомножителей, переменная i могла бы иметь значение, чуть меньшее n, и в итоге имели бы приближенное значение $(n+1)!$.

Еще один вариант программы вычисления факториала:

```
program fact2 (input, output);
 var n, p : integer;
 begin read (n); p := 1;
   while n > 0 do
      begin p := p * n; n := n — 1 end;
   write ('n!=', p)
 end.
```

Кроме названных ранее арифметических операций, в паскале есть еще две операции, знаки которых суть div и mod*). Эти операции имеют по два целых операнда (аргумента); если значения a и b неотрицательны и $b \neq 0$, то a **div** b и a **mod** b — это частное и остаток, возникающие при делении a на b с остатком. Например, 17 **div** 3 = 5, 17 **mod** 3 = 2, 8 **div** 2 = 4, 8 **mod** 2 = 0, 1 **div** 5 = 0, 1 **mod** 5 = 1 и т. д.

В общем случае для целых a и b, $b \neq 0$, определение операций таково:

$$a \textbf{ div } b = \begin{cases} \left[\frac{a}{b}\right], & \text{если } \frac{a}{b} \geqslant 0; \\ -\left[\left|\frac{a}{b}\right|\right], & \text{если } \frac{a}{b} < 0, \end{cases}$$

[...] означает целую часть числа,

$$a \textbf{ mod } b = a - ((a \textbf{ div } b) * b).$$

Эти операции одного старшинства с умножением и делением, что важно иметь в виду при вычислении значений выражений. Операцию **mod** можно использовать, чтобы узнать, кратно ли целое a целому b. А именно, a кратно b тогда и только тогда, когда a **mod** $b = 0$.

Пример программы, использующей введенные операции: алгоритм Евклида нахождения наибольшего общего делителя одновременно не равных нулю целых чисел a, b, $a \geqslant b \geqslant 0$.

```
program E(input, output);
 var a, b, c : integer;
 begin read(a, b);
   while b > 0 do
     begin c := a mod b; a := b; b := c end;
   write(a)
 end.
```

Пояснение. Алгоритм Евклида нахождения наибольшего общего делителя (НОД) неотрицательных целых чисел основан на следующих свойствах этой величины. Пусть a и b — одновременно не равные нулю целые числа, и пусть $a \geqslant b$. Тогда, если $b = 0$, то НОД$(a, b) = a$, а если $b \neq 0$, то для чисел a, b и c, где

*) div — сокращение слова division — деление, mod — сокращение латинского слова modulus — мера.

c — остаток от деления a на b, выполнено равенство НОД(a, b) = НОД(b, c). Например, НОД(15, 6) = = НОД(6, 3) = НОД(3, 0) = 3.

Рассмотрим пример программы, в которой встречаются переменные разных типов; вычисление суммы $1+\frac{1}{2}+\ldots+\frac{1}{n}$:

```
program S(input, output);
 var i, n : integer; s : real;
 begin read(n); s:=0; i:=0
   while i<n do
      begin i:=i+1; s:=s+1/i end;
   write(s)
 end.
```

Описания переменных разных типов могут произвольно чередоваться друг с другом. В последней программе мы могли бы написать **var** *s* : *real*; *i*, *n* : *integer*;

В паскале имеются две функции округления: *trunc*(x), *round*(x)*). Первая из них — это x без «дробных цифр»: *trunc*(3.14) = 3, *trunc*(—3.14) = —3, *trunc*(3) = 3 и т. д. Функция *round*(x) дает округление x до ближайшего целого:

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

round(3.14) = 3, *round*(3.7) = 4, *round*(—3.14) = —3, *round*(7) = 7 и т. д. Значениями функций *trunc* и *round* всегда являются числа типа *integer*.

Надо иметь в виду, что если слева от знака := помещается переменная типа *integer*, то справа от этого знака может помещаться лишь такое выражение, которое либо вообще не содержит переменных типа *real*, чисел типа *real*, знака операции деления /, функций *sin*, *cos*, *arctan*, *ln*, *exp*, *sqrt*, либо содержит их только под знаками функций *trunc* и *round*. В силу принятого в паскале способа выполнения арифметических операций и способа вычисления значений функций, значения этих и только этих выражений будут иметь тип *integer*. Такие выражения мы назовем выражениями со значениями типа *integer*. Спра-

*) trunc — сокращение от truncate — усекать, отрезать; round — круглый.

ва от знака := может размещаться любое выражение, коль скоро слева от этого знака помещена переменная типа *real*. Таким образом, если переменные i, j имеют тип *integer*, а переменные r, s — тип *real*, то недопустимы операторы присваивания:

$i := 3.14$, $i := r$, $j := 4/2$, $i := sqrt(abs(j)) + 2$ и т. д.;

допустимы операторы присваивания:

$r := 3.14$, $r := i$, $r := 4/2$, $r := sqrt(s)$, $i := round(4/2)$,

$i := abs(j)$, $i := i * j$, $i :=$ **i div** j,

$j := 1 + trunc(1.5 + sin(2.7 + r))$.

ЗАДАЧИ

1. Верно ли, что $b * a$ **div** $b = a$ тогда и только тогда, когда a кратно b?

2. Дано натуральное n. Написать программу вычисления суммы всех чисел вида $i^3 - 3in^2 + n$, $i = 1, 2, \ldots, n$, которые являются удвоенными нечетными числами.

3. Последовательность $u_0, u_1, u_2, \ldots$ определяется правилами

$$u_0 = 0, \quad u_1 = 1, \quad u_{i+2} = u_{i+1} + u_i, \quad i = 0, 1, \ldots$$

Написать программу вычисления u_n.

4. Доказать, что любую целочисленную (в рублях) денежную сумму больше 7 р. можно выплатить без сдачи трешками и пятерками. Написать программу выработки по данному целому числу $n > 7$ пары целых неотрицательных чисел a и b таких, что $n = 3a + 5b$.

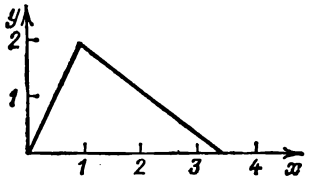


Рис. 2

5. Показать, что если периодическая функция $f(x)$ с периодом t задана на отрезке $[0, t]$, то для любого x значение

$$a = \begin{cases} x - trunc(x/t) * t & \text{при } x \geqslant 0, \\ x - (trunc(x/t) - 1) * t & \text{при } x < 0 \end{cases}$$

принадлежит отрезку $[0, t]$, и значение функции $f(a)$, вследствие периодичности, должно быть равно $f(x)$. Пользуясь этим, написать программу вычисления значения $f(x)$, период которой равен 3,4, а часть графика изображена на рис. 2.

Глава II

РЯД ДОПОЛНИТЕЛЬНЫХ ВОЗМОЖНОСТЕЙ

§ 1. Оператор цикла с параметром

Привлечение переменных типа *integer* позволяет использовать еще один оператор цикла, который называется *оператором цикла с параметром* *):

for $i := A$ **to** B **do** S **),

здесь i — некоторая переменная типа *integer*, которая называется параметром цикла, A и B — выражения со значением типа *integer*, S — оператор (тело цикла). Предполагается, что в выражения A и B не входит переменная i и что выполнение оператора S не приводит к изменению значений параметра цикла i и тех переменных, которые входят в A и B. Такой оператор цикла заставляет переменную i последовательно принимать значения $A, A+1, A+2, \ldots, B$. Для каждого из этих значений выполняется оператор S. Если значение A больше значения B, то оператор S не будет выполнен ни разу.

Пусть перед выполнением оператора цикла

for $i := 1$ **to** n **do** $s := s + i * i * i$

переменная s имела значение 0. Тогда после выполнения переменная s примет значение $1^3 + 2^3 + \ldots + n^3$.

Программа вычисления $n!$:

```
program fact3(input, output);
 var n, i, p : integer;
 begin read(n); p:=1;
   for i:=1 to n do p:=p*i;
   write('n!=', p)
 end.
```

*) Разбиравшийся в предыдущей главе оператор цикла будем называть *оператором цикла с условием*.

**) for — для, to — к.

В § 5 гл. I уже рассматривались два варианта программы вычисления факториала *fact*1, *fact*2. Обе эти программы были написаны с использованием операторов цикла с условием. Вариант *fact*3 несколько проще, так как в нем не пришлось выписывать операторы, описывающие изменение значений *i*.

Оператор цикла с условием необходимо применять в тех случаях, когда число шагов не устанавливается заранее непосредственным рассмотрением значений переменных. Например, он очень удобен для описания алгоритма Евклида (программа *E* из § 6 гл. I) и написания целого ряда программ, аналогичных программам *S* и *Sum* из § 4 гл. I. Но в тех случаях, когда число шагов легко определяется по исходным данным (например, видно, что это число равно значению переменной *n*), оператор цикла с параметром выглядит, как правило, проще.

Имеется еще один вариант оператора цикла с параметром:

for *i* := *A* **downto** *B* **do** *S* *).

Здесь *i* принимает последовательно значения *A*, *A* — 1, ..., ..., *B*, и для каждого из них выполняется оператор *S*, если же значение *A* меньше значения *B*, то оператор *S* не выполняется ни разу.

Еще один вариант программы вычисления факториала:

```
program fact4 (input, output);
 var n, i, p : integer;
 begin read (n); p := 1;
   for i := n downto 2 do p := p * i;
   write ('n!=', p)
 end.
```

Рассмотрим еще примеры программ с операторами цикла с параметром.

---

*) down to — вниз к.

Программа выбора наименьшего из 1000 данных чисел:

```
program min(input, output);
 var i : integer; x, y : real;
 begin read(y);
   for i:=2 to 1000 do
     begin read(x);
       if x<y then y:=x
     end;
   write('min=', y)
 end.
```

Значение наименьшего элемента вычисляется за 1000 шагов, первый из которых состоит в присваивании с помощью оператора ввода переменной y значения первого элемента; остальные 999 шагов выполняются с помощью оператора цикла. После выполнения i-го шага значение переменной y равно наименьшему из первых i чисел.

В программе *min* строку

```
for i:=2 to 1000 do
```

можно заменить строкой

```
for i:=1 to 999 do
```

или строкой

```
for i:=1000 downto 2 do
```

— от этого ничего не изменится, так как сами значения параметра цикла при выполнении программы не используются, и существенным является только число шагов цикла.

Интересна конструкция, в которой тело цикла само является оператором цикла или содержит в себе оператор цикла. Напишем программу получения всех совершенных чисел из диапазона от 1 до n (натуральное число называется совершенным, если оно равно сумме всех своих делителей, исключая себя самого; например, $6=1+2+3$):

```
program совершенные (input, output);
 var n, i, j, m : integer;
 begin read (n);
   for i:=2 to n do
     begin m:=0;
       for j:=1 to i-1 do
         if i mod j=0 then m:=m+j;
       if m=i then writeln (i)
     end
 end.
```

Пояснение: единица — это не совершенное число; каждое число i от 2 до n проверяется путем сложения всех его делителей в диапазоне от 1 до $i-1$ и сравнения суммы с самим i.

В заключение этого параграфа приведем правило, принятое в паскале: *по окончании выполнения оператора цикла с параметром значение параметра цикла считается неопределенным.*

ЗАДАЧИ

1. Написать программу вычисления a^n, n — натуральное число.

2. Написать программы вычисления значений:

а) $\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+\ \ldots\ +\frac{1}{(n-1)\cdot n}$, $n\geqslant 2$;

б) $1-\frac{1}{2}+\ \ldots\ +\frac{(-1)^n}{n}$;

в) $\left(1+\frac{1}{1^2}\right)\left(1+\frac{1}{2^2}\right)\cdot\ \ldots\ \cdot\left(1+\frac{1}{n^2}\right)$.

3. Написать программу получения в порядке убывания всех делителей данного числа.

4. Написать программу вычисления суммы положительных и количества отрицательных чисел, содержащихся среди данных 250 чисел.

5. Обозначим

$$f_i=\frac{1}{i^2+1}+\frac{1}{i^2+2}+\ \ldots\ +\frac{1}{i^2+i+1},\qquad i=0,\ 1,\ \ldots$$

Написать программу вычисления произведения $f_0\cdot f_1\cdot\ \ldots\ \cdot f_n$.

§ 2. Оператор перехода. Пустой оператор

Каждый оператор может быть помечен *меткой* — целым числом без знака, содержащим не более 4 цифр. Метка располагается перед оператором и отделяется от него двоеточием. Оператор *write*(x), помеченный меткой 21, запишется так:

21 : *write*(x)

Метка не влияет на выполнение оператора.

В рассмотренных программах операторы выполнялись в том порядке, в каком были написаны. Изменить этот порядок можно с помощью *оператора перехода.* Оператор перехода состоит из специального слова **goto** *), за которым следует метка.

goto 17

goto 0

Оператор перехода прерывает естественную последовательность выполнения операторов: следом за ним выполняется оператор, помеченный указанной меткой.

Пусть программа содержит последовательность операторов:

$x := 2$; $a := b$; **goto** 99;

14 : $a := 0$; $x := b$; 99 : $y := x$; *write*(x)

В этом случае сначала выполняются операторы $x := 2$ и $a := b$, затем следует переход к оператору, помеченному меткой 99, т. е. к оператору $y := x$. После оператора $y := x$ будет выполнен оператор *write*(x).

О *пустом операторе.* Пустой оператор не предписывает никаких действий. По определению он представляет собой пустую совокупность символов. Как и все операторы, пустой оператор может быть помечен меткой.

Рассмотрим конец некоторой программы или составного оператора:

... 10 : **end.** или ... 10 : **end**

Здесь перед **end** расположен помеченный меткой 10 пустой оператор. Основное назначение пустого оператора — дать возможность выхода из середины программы или составного оператора.

*) go to — перейти к.

Все метки должны быть описаны. Описание меток состоит из служебного слова **label** *) и следующего за ним списка меток. Например,

```
label 17, 0, 14;
```

Описания меток располагаются до совокупности всех описаний переменных. Описанной меткой должен быть помечен ровно один оператор программы.

В результате выполнения следующей программы выясняется, имеются или не имеются среди чисел $cos(i^3)\,sin(in)$, $i = 1, \ldots, n$, меньшие 0.0001. Если имеются, то выводится «есть», если нет — «нет».

```
program П1 (input, output);
 label 1;
 var i, n : integer;
 begin read (n);
   for i := 1 to n do
     if cos(i * i * i) * sin(i * n) < 0.0001 then
       begin
         write('есть');
         goto 1
       end;
   write ('нет');
1 : end.
```

Если оказывается, что некоторое число меньше 0.0001, то следующие числа уже не рассматриваются. Программу можно было бы написать с двумя операторами перехода:

```
program П2 (input, output);
 label 1, 2;
 var i, n : integer;
 begin read (n);
   for i := 1 to n do
     if cos(i * i * i) * sin(i * n) < 0.0001 then goto 1;
   write ('нет'); goto 2;
 1 : write ('есть');
2 : end.
```

*) label — метка.

С помощью оператора перехода, расположенного вне условного оператора или оператора цикла, нельзя перейти внутрь этого условного оператора или оператора цикла.

Заметим, что благодаря наличию описаний меток последние две программы уже не укладываются в ту схему простейшей программы, которая приводилась в § 2 гл. I.

ЗАДАЧИ

1. Написать программу, в результате выполнения которой выясняется, есть ли среди чисел $i^3 - 17in^2 + n^3$ $(i = 1, \ldots, n)$ хотя бы одно число, которое кратно a и не кратно b.

2. Рассмотрим все числа, предшествующие первому отрицательному числу в последовательности данных чисел. Образуют ли они возрастающую последовательность? Написать программу этого исследования.

3. Написать программу, в результате выполнения которой выясняется, есть или нет среди 70 данных целых чисел точные квадраты. Если есть, то должно быть выведено одно из таких чисел, если нет, то должен быть выведен текст «квадратов нет».

4. Написать программу получения первого простого числа, большего данного числа n.

5. Можно ли указать программу, выполнение которой завершается и конец которой имеет вид

а) ;**goto** 1 **end**.
б) **goto** 1 **end**.
в) **goto** 1; 1 : **end**.
г) **goto** 1; 2 : **end**.

§ 3. Логические операции

В условных операторах

if B **then** P **else** Q

if B **then** P

и в операторе цикла

while B **do** P

в качестве условия B мы до сих пор использовали только отношения равенства и неравенства. Отношение — это высказывание определенного вида о значениях переменных. Отношение $s \geq t$ — это высказывание, утверждаю-

щее, что значение переменной s больше, чем значение переменной t. Отношение $s \neq t + 12$ — это высказывание, утверждающее, что значение переменной s не равно сумме значения переменной t и двенадцати.

Высказывания о значениях переменных могут быть истинными или ложными в зависимости от самих значений переменных. Так, если $s = 5$, $t = 6$, то высказывание $s > t$ — ложное, а высказывание $s \neq t + 12$ — истинное. Если же $s = 14$, $t = 2$, то высказывание $s > t$ — истинное, а $s \neq t + 12$ — ложное. Высказывания, которые не содержат переменных, например $2 * 2 = 4$, $7 < 1$, как и высказывания вида $3 * abs(x) > x$, $s + 100 < s$, либо истинны для всех значений переменных, либо ложны для всех значений переменных.

Из простых высказываний в паскале разрешается строить более сложные. При этом считается, что все отношения, которые используются при построении, заключены в скобки. Пусть A и B — некоторые высказывания, тогда A **and** B *) — это новое высказывание, утверждающее истинность обоих высказываний A и B; A **or** B **) — это новое высказывание, утверждающее истинность хотя бы одного из высказываний A и B. Если C — высказывание, то **not** C ***) — это новое высказывание, утверждающее, что C — ложное высказывание. Следующие таблицы уточняют словесные объяснения.

A	*B*	*A* and *B*	*A* or *B*
истинно	истинно	истинно	истинно
истинно	ложно	ложно	истинно
ложно	истинно	ложно	истинно
ложно	ложно	ложно	ложно

C	not *C*
истинно	ложно
ложно	истинно

Операции над высказываниями (логические операции) **and**, **or**, **not** называются соответственно *конъюнкцией*, *дизъюнкцией* и *отрицанием*.

Если переменные x и y имеют числовые значения, то можно построить высказывания

$(abs(x) + abs(y) \leqslant 1)$ **or** $(x \geqslant 0)$ и

$(abs(x) + abs(y) \leqslant 1)$ **and** $(x \geqslant 0)$.

*) and — и.
**) or — или.
***) not — не.

Первое из них истинно, если, например, $x = y = 0$, и ложно, если, например, $x = -2$, $y = 3$. Второе из них истинно, если, например, $x = 0.5$, $y = 0.25$, и ложно, если, например, $x = 3$, $y = 3$. На рис. 3, 4 в заштрихованные области попали все точки, значения координат x, y которых таковы, что истинны высказывания $(abs(x) + abs(y) \leqslant 1)$ **or** $(x \geqslant 0)$ и $(abs(x) + abs(y) \leqslant 1)$ **and** $(x \geqslant 0)$ соответственно. На рис. 5 в заштрихованную область

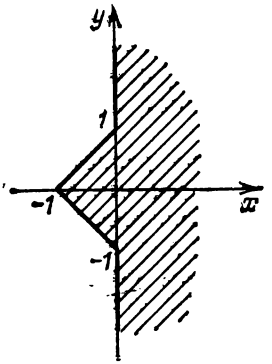


Рис. 3

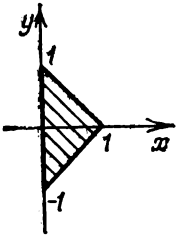


Рис. 4

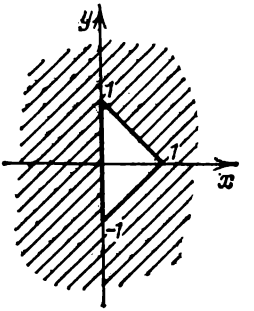


Рис. 5

попали все точки, значения координат x, y которых таковы, что истинно высказывание **not** $((abs(x) + abs(y) \leqslant 1)$ **and** $(x \geqslant 0))$, граница в этом последнем случае не включена в область.

При определении истинности высказывания, построенного из отношений с помощью знаков логических операций и круглых скобок, действуют следующие правила старшинства операций: самая старшая операция — отрицание, следующая — конъюнкция, потом — дизъюнкция.

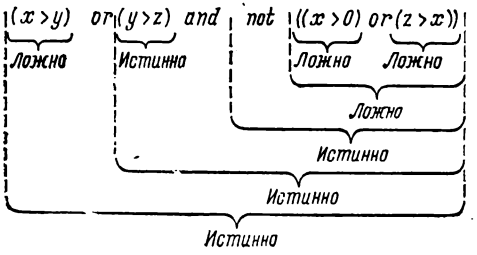
Первой из двух операций одного старшинства выполняется та, знак которой в выражении встречается раньше. Круглые скобки изменяют этот естественный порядок. Для высказывания

$$(x > y)\ \textbf{or}\ (y > z)\ \textbf{and not}\ ((x > 0)\ \textbf{or}\ (z > x))$$

устанавливается следующий порядок логических операций:

$$\overset{4}{(x > y)}\ \textbf{or}\ \overset{3}{(y > z)}\ \textbf{and not}\ \overset{2}{((x > 0)}\ \textbf{or}\ \overset{1}{(z > x))}.$$

При $x = -1$, $z = -2$, $y = 1$ имеем



Такого рода высказывания используются в условных операторах и операторах цикла в качестве условий. Рассмотрим, например, программу, в результате выполнения которой выясняется, принадлежит ли число x отрезку $[a, b]$. Первый вариант:

```
program отрезок1 (input, output);
 var x, a, b : real;
 begin read (x, a, b);
   if (x ⩾ a) and (x ⩽ b) then write ('принадлежит')
                          else write ('не принадлежит')
 end.
```

Второй вариант:

```
program отрезок2 (input, output);
 var x, a, b : real;
 begin read (x, a, b);
   if (x < a) or (x > b) then write ('не ');
   write ('принадлежит')
 end.
```

В этих программах логические операции позволили избежать громоздких вложений одного условного оператора в другой.

Другой пример. Пусть $x_0 = y_0 = 10$ и $x_{i+1} = 0.1y_i$, $y_{i+1} = 0.1x_i - 0.12y_i$ для $i = 1, 2, \ldots$ Пусть $\varepsilon > 0$. Требуется найти наименьшее i такое, что $|x_i| \leq \varepsilon$, $|y_i| < \varepsilon$.

Программа:

```
program I (input, output);
 var x, y, xx : real; i : integer; eps : real;
 begin read (eps); x := 10; y := 10; i := 0;
   while (abs (x) ⩾ eps) or (abs (y) ⩾ eps) do
     begin
       xx := 0.1 * y; y := 0.1 * x - 0.12 * y;
       x := xx; i := i + 1
     end;
   write (i)
 end.
```

ЗАДАЧИ

1. Написать программы, в результате выполнения которых определяется, принадлежит ли точка с координатами x, y

а) квадрату, изображенному на рис. 6;

б) фигуре, изображенной на рис. 7.

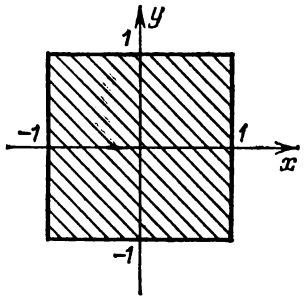


Рис. 6

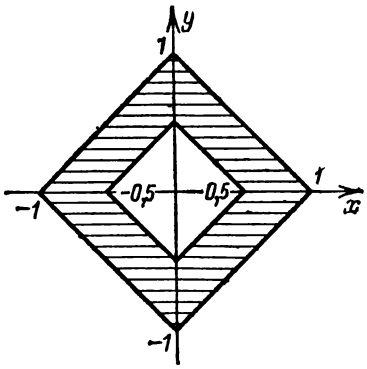


Рис. 7

2. Написать программу вычисления значения функции

$$y = \begin{cases} x, & \text{если } x \geqslant 1 \text{ или } x < -2, \\ x^2 + 2x - 2 & \text{в остальных случаях.} \end{cases}$$

3. Написать программу, в результате выполнения которой определяется количество удвоенных нечетных чисел среди данных 150 целых чисел.

4. Написать программу нахождения первого члена последовательности $a_n = (-1)^n \left(1 + \frac{1}{2} + \ldots + \frac{1}{n}\right)$, $(n = 1, 2, \ldots)$, который не принадлежит отрезку $[a, b]$, где a и b — данные числа.

5. Пусть A и B — некоторые высказывания. Доказать, что высказывание **not** (A **and** B) истинно тогда и только тогда, когда истинно высказывание **not** A **or not** B, а высказывание **not** (A **or** B) истинно тогда и только тогда, когда истинно высказывание **not** A **and not** B.

§ 4. Тип *char*

До сих пор рассматривались исключительно программы, предназначенные для обработки числовых данных. Обработка символьных (иначе: знаковых, литерных) данных становится возможной благодаря привлечению значений и переменных типа *char* *). Значениями типа *char* служат отдельные символы — все те символы, которые могут быть восприняты используемой вычислительной машиной и напечатаны ею на бумаге или высвечены на экране. К этим символам обязательно должны относиться те, которые используются в текстах программ на паскале: буквы, цифры, знаки операций, скобки и т. д.

Если в программе имеется описание

u, v : char

то возможны, например, операторы присваивания $u := 'a'$, $u := v$, $v := '*'$ и т. д. Штрих $'$ — принятая в паскале форма кавычки — употребляется всякий раз, когда значение типа *char* явно указывается в программе. При выполнении оператора вывода штрихи — кавычки не печатаются и не высвечиваются. Выполнение операторов

$u := 'b'$; *write* (u)

приводит к печати или высвечиванию символа b.

Пусть, по-прежнему, u и v — переменные типа *char*. В качестве условий после служебных слов **if** и **while** могут употребляться отношения $'a' = u$, $u = v$, $v \neq 'b'$

*) char — сокращение от character — символ.

и т. д., а также построенные на их основе более сложные логические выражения.

В следующих двух примерах будут приведены программы, которые предназначены для исследования и преобразования символов, предшествующих первому символу / в последовательности данных символов.

Программа замены всех восклицательных знаков точками:

```
program точки(input, output);
 var c : char;
 begin read(c);
   while c ≠ '/' do
   begin if c = '!' then write('.')
                    else write(c);
         read(c)
   end
 end.
```

Программа выбрасывания групп символов, расположенных между скобками (,). Сами скобки тоже выбрасываются. Предполагается, что внутри каждой пары скобок нет других скобок.

```
program скобки(input, output);
 var c : char; i : integer;
 begin i:=0; read(c);
   while c ≠ '/' do
     begin
       if c = '(' then i:=1 else
         if c = ')' then i:=0 else
           if i = 0 then write(c);
       read(c)
     end
 end.
```

Пояснение: $i = 1$ означает, что ранее была прочитана левая скобка, которой пока еще не нашлось пар-

ной правой; в этой ситуации прочитываемые символы не выводятся.

Кроме отношений со знаками $=$ и $\neq$, после служебных слов **if** и **while** могут использоваться аналогичные отношения (связывающие переменные и значения типа *char*), в которых использованы знаки $>$, $\geqslant$, $<$, $\leqslant$, так как все множество символов считается упорядоченным. Предполагается, что эта упорядоченность получается так: из всех символов составлен список, и из двух символов меньше тот, который встречается в списке раньше. Оговорим, что малые латинские буквы идут друг за другом, не перемешиваясь с другими символами, в алфавитном порядке, т. е. список содержит фрагмент

$$\ldots a, b, \ldots, z \ldots,$$

и арабские цифры идут друг за другом, как обычно:

$$\ldots 0, 1, \ldots, 9 \ldots$$

Поэтому, например, $'a' < 'c'$, $'y' \geqslant 'x'$, $'3' > '1'$. В остальном вопрос порядка решается по-разному на разных вычислительных машинах.

Упорядоченность открывает возможность использования в программах операторов цикла с параметром. Параметр цикла, имеющий тип *char*, пробегает ряд символов в указанных границах. Выполнение оператора цикла

for $c := 'a'$ **to** $'z'$ **do** *write*(c)

где c — переменная типа *char*, приведет к печати или высвечиванию всех малых букв латинского алфавита:

abcdefghijklmnopqrstuvwxyz

Выполнение оператора цикла

for $c := 'z'$ **downto** $'a'$ **do** *write*(c)

приведет к печати или высвечиванию этих же букв в обратном порядке:

zyxwvutsrqponmlkjihgfedcba

Пример. Программа вывода последовательностей букв:

```
a
ab
abc
. . . . . .
abc ... xyz
program aab(output);
 var c, d : char;
 begin
   for c:='a' to 'z' do
    begin
      for d:='a' to c do write(d);
      writeln('_')
    end
 end.
```

Программа не содержит оператора ввода, идентификатор *input* опущен.

Рассмотрим еще пример, в котором использована упорядоченность значений типа *char*. Программа, в результате выполнения которой выясняется, имеется ли хотя бы одна малая латинская буква среди символов, предшествующих первому символу / в последовательности данных символов:

```
program буква(input, output);
 label 1;
 var c : char;
 begin read(c);
   while c<>'/' do
     if ('a'<=c) and (c<='z') then
       begin write('есть'); goto 1 end
                          else read(c);
   write('нет');
1 : end.
```

ЗАДАЧИ

В задачах 1—3 рассматриваются символы, предшествующие символу / в последовательности данных символов. Требуется написать программы выполнения указанных заданий.

1. Заменить каждую из групп стоящих рядом точек одной точкой.

2. Заменить каждую точку многоточием, т. е. тремя точками.

3. Выяснить, имеется ли пара соседствующих символов, — (запятая, тире).

4. Среди данных 500 символов найти и вывести все имеющиеся пары стоящих рядом одинаковых символов.

5. Вывести последовательности

а) $abbccc \ldots zzz \ldots z$,

б) $zyyxxx \ldots aaa \ldots a$,

в) $abc \ldots zbc \ldots zc \ldots z \ldots z$.

Глава III

НЕСТАНДАРТНЫЕ ТИПЫ

§ 1. Массивы. Регулярные типы

Тип в программировании — это множество, для которого оговорен некоторый набор операций над элементами. Сами элементы множества называются *объектами* (или значениями) данного типа.

Типы *real* и *integer* — это числовые множества. Вместе с ними рассматривают соответствующие арифметические операции и операции сравнения. Тип *char* — это множество символов. Вместе с ним рассматривают операции сравнения.

Эти три типа — *стандартные* типы паскаля. В паскале имеются средства, позволяющие определять, исходя из имеющихся типов, новые, нестандартные типы. Мы остановимся сейчас на одном из этих средств.

В математике наряду с некоторым множеством часто рассматривают множества упорядоченных пар элементов данного множества, троек и т. д. *). Упорядоченные пары, тройки и т. д. в паскале удобно задавать в виде массивов длины 2, 3 и т. д. *Массив* — это набор объектов одного типа, у каждого из которых есть индекс (номер). Элементы массива длины 20 могут иметь, например, индексы 1, 2, ..., 20 или 0, 1, ..., 19. Способ индексации, тип элементов, длина массива фиксируются в определении того типа, к которому принадлежит массив. Рассмотрим конкретный пример. Определение, имеющее вид

t = **array** [1 .. 20] **of** *real* **),

это определение типа, имя которого t. Объектами типа t будут упорядоченные наборы по 20 элементов, имеющих

*) Например, координаты точек плоскости образуют упорядоченные пары действительных чисел, координаты точек пространства — упорядоченные тройки.

**) array — массив, порядок, строй; of — из (точнее, здесь of — предлог, служащий для образования родительного падежа — массив действительных чисел).

тип *real*; диапазон изменения значения индекса — от 1 до 20. Это определение типа, предваренное служебным словом type *), помещается в программу перед совокупностью описаний переменных. Пусть переменная a описана в программе как переменная типа t:

var a : t; ...

Тогда при выполнении программы значениями переменной a будут массивы длины 20, элементы которых имеют тип *real*. Для того чтобы рассматривать эти элементы по отдельности, для них применяются обозначения $a[1]$, $a[2]$, ..., $a[20]$.

Переменная a — это переменная типа t, переменные $a[1]$, $a[2]$, ..., $a[20]$ — это переменные типа *real*. С переменными $a[1]$, $a[2]$, ..., $a[20]$ можно обращаться как с обычными переменными типа *real* (т. е. как с переменными — идентификаторами x, y, z и т. д.). Заключенный в квадратные скобки индекс — это не обязательно целое число, им может быть произвольное выражение со значением типа *integer*. Например, переменные $a[i]$, $a[2*i]$, $a[2*i-1]$ удобно использовать для поочередного рассмотрения в цикле всех элементов массива, элементов, стоящих на четных и нечетных местах; здесь проявляется преимущество обозначений $a[1]$, $a[2]$, ..., ..., $a[20]$ перед обозначениями $a1$, $a2$, ..., $a20$. Значение индекса обязано лежать в указанном в определении типа диапазоне, в данном случае — в диапазоне от 1 до 20.

Операции над объектами типа t — это доступ к отдельным элементам массивов через индексы и изменение отдельных элементов массивов с помощью операций, связанных с типом *real*.

Пример. Пусть $a[1]$, ..., $a[20]$ — количество осадков в миллиметрах, выпадавшее в Москве в течение первых 20 лет нашего столетия. Надо вычислить среднее количество осадков и отклонение от среднего для каждого года. Программа:

```
program осадки(input, output);
 type t = array [1 .. 20] of real;
 var a : t; i : integer; s : real;
 begin s := 0;
```

*) type — тип.

```
    for i:=1 to 20 do
      begin read(a[i]); s:=s+a[i] end;
    s:=s/20; writeln(s);
    for i:=1 to 20 do writeln(s-a[i])
  end.
```

В определении типа *t* мы указали диапазон изменения индекса от 1 до 20. Можно было бы взять и диапазон от 0 до 19. Вообще, можно было в качестве границ взять любые целые числа, разность между которыми равна 19. Естественным был бы диапазон от 1901 до 1920 — определение типа *t* выглядело бы тогда так:

t = **array** [1901 .. 1920] **of** *real*

Соответствующие изменения потребовалось бы внести и в операторы цикла: **for** *i* := 1901 **to** 1920 **do** ...

Резюмируем и дополним сказанное. Если тип *u* определен в программе с помощью конструкции **array**...**of**..., то он называется *регулярным* типом. Общий вид определения регулярного типа *u* есть

u = **array** [n_1 .. n_2] **of** *r*

где n_1 и n_2 — некоторые конкретные целые числа такие, что $n_2 \geqslant n_1$, а *r* — это или один из стандартных типов *real*, *integer*, *char*, или имя ранее определенного типа. Тип *r* называется *базовым* по отношению к типу *u*. Объекты регулярного типа называются массивами. Совокупность всех определений типов оформляется в виде

type T_1; T_2; ...; T_m;

где T_1, T_2, ..., T_m — определения отдельных типов. Эта совокупность помещается в программе до совокупности описаний переменных и после описания меток (если описание меток есть в программе). Если тип *u* определен так, как написано выше, а переменная *a* — это переменная типа *u*, то $a[n_1]$, ..., $a[n_2]$ — это переменные типа *r*. В квадратных скобках в качестве индекса можно помещать выражение со значением типа *integer*, например, можно писать $a[2*i+1]$. Такие конструкции тоже являются переменными типа *r*. Значение индекса не должно выходить из диапазона от n_1 до n_2. Операции над значениями типа *u* — это доступ к отдельным элементам массивов посредством указания индексов, а также изменение отдельных элементов с помощью операций, связанных с базовым типом *r*.

Для переменных a и b одного и того же типа можно использовать оператор присваивания $a := b$.

Пример. Дано 200 целых чисел $x_1, x_2, \ldots, x_{200}$. Необходимо получить последовательность чисел

$x_{200}, x_{100}, x_{199}, x_{99}, \ldots, x_{101}, x_1$.

Программа:

```
program смесь (input, output);
 type t = array [1 .. 200] of integer;
 var x : t; i : integer;
 begin
   for i := 1 to 200 do read (x [i]);
   for i := 0 to 99 do write (x [200 — i], '_',
   x [100 — i])
 end.
```

Пример. Дана последовательность символов (объектов типа *char*) $s_1, s_2, \ldots, s_{300}$. Требуется определить, совпадает ли начальная часть $s_1, \ldots, s_{150}$ последовательности с ее концевой частью $s_{151}, \ldots, s_{300}$.

Программа:

```
program дуплекс (input, output);
 label 1;
 type t = array [1 .. 150] of char;
 var x : t; y : char; i : integer;
 begin
   for i := 1 to 150 do read (x [i]);
   for i := 1 to 150 do
     begin read (y);
       if x [i] ≠ y then
         begin write ('не_');
               goto 1
         end
     end;
1 : write ('совпадают')
 end.
```

Программа не требует одновременного удерживания в памяти обеих частей данной последовательности символов.

Пример. Программа построения последовательности $a_1, \ldots, a_{30}$, образованной по следующему закону:

$$a_1 = 1,\ a_i = a_{\left[\frac{i}{2}\right]} + a_{i-1}\ (i = 2, \ldots, 30):$$

```
program числа(output);
 type t = array [1 .. 30] of integer;
 var a : t; i : integer;
 begin a[1] := 1; writeln(a[1]);
   for i := 2 to 30 do
     begin
       a[i] := a[i div 2] + a[i - 1];
       writeln(a[i])
     end
 end.
```

ЗАДАЧИ

1. Написать программу, которая позволяет получить элементы данной последовательности символов $s_1, \ldots, s_{75}$ в обратном порядке: $s_{75}, \ldots, s_1$.

2. Написать программы, которые по данной последовательности действительных чисел $a_1, \ldots, a_{200}$ позволяют вычислить:

а) $a_1a_{101} + a_2a_{102} + \ldots + a_{100}a_{200}$,

б) $a_1a_{200} + a_2a_{199} + \ldots + a_{100}a_{101}$.

3. Написать программы, которые по данной последовательности символов $s_1, \ldots, s_{99}$ позволяют выяснить:

а) имеются ли в последовательности повторяющиеся символы,

б) является ли последовательность палиндромом (перевертышем), т. е. верно ли, что $s_1 = s_{99}$, $s_2 = s_{98}$, ...

4. На плоскости даны 700 точек, эти точки попарно соединены отрезками. Написать программу вычисления длины наибольшего из отрезков. Считать, что координаты i-й точки суть x_i, y_i и что задана последовательность действительных чисел $x_1, y_1, x_2, y_2, \ldots, x_{700}, y_{700}$.

5. Написать программу вычисления коэффициентов многочлена $c_0 + c_1x + \ldots + c_{20}x^{20}$, являющегося произведением многочленов $a_0 + a_1x + \ldots + a_5x^5$ и $b_0 + b_1x + \ldots + b_{15}x^{15}$. Считать, что задана последовательность действительных чисел $a_0, \ldots, a_5, b_0, \ldots, b_{15}$.

§ 2. Массивы массивов. Матрицы

В определении регулярного типа u

$u =$ **array** $[n_1 .. n_2]$ **of** r

в качестве базового типа r может выступать *real*, *integer*, *char* или ранее определенный тип. Пусть r, в свою очередь, определен как регулярный тип

$r =$ **array** $[m_1 .. m_2]$ **of** s

и пусть переменная a — это переменная типа u. Тогда $a[i]$ — это переменная типа r, а $a[i][j]$ — переменная типа s.

Переменные с двумя индексами удобны для работы с таблицами. Таблица круга футбольного чемпионата после отбрасывания названий команд и после заполнения диагональных клеток нулями представляет собой квадратную числовую таблицу, обладающую тем свойством, что, во-первых, каждое число — это 0, 1 или 2 и, во-вторых, сумма числа очков, набранных i-й командой в игре с j-й, и числа очков, набранных j-й командой в игре с i-й, при $i \neq j$ равна 2 (рис. 8). В программе удобно эти числа очков обозначить $a[i][j]$ и $a[j][i]$. К этому обозначению можно прийти так: a — это таблица (массив строк), $a[i]$ — i-я строка таблицы (массив чисел), $a[i][j]$ — j-й элемент i-й строки (число).

0	1	0	1
1	0	2	2
2	0	0	1
1	0	1	0

Рис. 8

Напишем программу проверки равенства двум каждой из сумм вида $a[i][j] + a[j][i]$ при $i \neq j$. Предполагается, что таблица имеет размер 15×15 и что данная последовательность целых чисел $c_1, \ldots, c_{225}$ — это выписанные одна за другой строки таблицы: $c_1, \ldots, c_{15}$ —

первая строка, c_{16}, ..., c_{30} — вторая строка и т. д.

```
program круг(input, output);
 label 1;
 type строка = array [1 .. 15] of integer;
      таблица = array [1 .. 15] of строка;
 var a : таблица; i, j : integer;
 begin
   for i := 1 to 15 do
     for j := 1 to 15 do read(a[i][j]);
   for i := 1 to 15 do
     for j := i + 1 to 15 do
       if a[i][j] + a[j][i] ≠ 2
         then begin
                write('есть ошибка');
                goto 1
              end;
   write('ошибок нет')
1 : end.
```

Пояснение: после того как таблица введена, с помощью двойного цикла организуется перебор всех i, j таких, что $1 \leqslant i < j \leqslant 15$, т. е. перебор всех элементов, находящихся в правой верхней части таблицы. Эти элементы складываются с соответствующими им элементами левой нижней части.

Квадратные и прямоугольные таблицы часто называют матрицами. Пусть квадратная матрица 15×15, состоящая из действительных чисел, задана, как и в предыдущем примере, построчно. Надо поэлементно вычесть последний столбец из всех столбцов, кроме последнего (столбец с номером j — это элементы $a[1][j]$, $a[2][j]$, ...). Программа:

```
program столбцы(input, output);
 type строка = array [1 .. 15] of real;
   матрица = array [1 .. 15] of строка;
 var a : матрица; i, j : integer;
 begin
```

```
for i:=1 to 15 do
  for j:=1 to 15 do read(a[i][j]);
for j:=1 to 14 do
  for i:=1 to 15 do a[i][j]:=a[i][j]-a[i][15];
for i:=1 to 15 do
  begin for j:=1 to 15 do write(a[i][j], '_');
        writeln('_')
  end
end.
```

В программах можно использовать переменные, содержащие три индекса: например, $a[i][j][k]$, и еще большее число индексов.

Следует помнить, что исходные данные для программы — это не таблицы и не массивы, а последовательность значений стандартных типов, которая может быть прочитана элемент за элементом с помощью оператора ввода *read*. Каким именно переменным (с индексами, без индексов и т. д.) будут присвоены эти значения, целиком зависит от того, как составлена программа.

ЗАДАЧИ

1. Таблица круга футбольного чемпионата, в котором участвовало 25 команд, задана своей верхней правой частью; первые 24 числа данной последовательности относятся к первой строке таблицы, следующие 23 числа — ко второй и т. д. Написать программу построения всей таблицы целиком.

2. Написать программу решения системы линейных уравнений

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1\,10}x_{10} &= b_1 \\ a_{22}x_2 + \ldots + a_{2\,10}x_{10} &= b_2 \\ \cdots\cdots\cdots\cdots & \\ a_{99}x_9 + a_{9\,10}x_{10} &= b_9 \\ a_{10\,10}x_{10} &= b_{10} \end{aligned}$$

Предполагается, что $a_{11} \cdot a_{22} \cdot \ldots \cdot a_{10\,10} \neq 0$ и что исходные данные имеют вид последовательности

a_{11}, a_{12}, ..., $a_{1\,10}$, a_{22}, ..., $a_{2\,10}$, ..., a_{99}, $a_{9\,10}$, $a_{10\,10}$, b_1, b_2, ..., b_{10}.

3. Написать программу транспонирования данной матрицы 20×20 (при транспонировании строки и столбцы меняются ролями: i-й столбец становится i-й строкой).

4. Написать программу, которая позволяет получить строку и столбец, содержащие наибольший элемент целочисленной матрицы 20×20.

5. Написать программу, которая позволяет проверить, верно ли, что в данной целочисленной матрице 17×17 суммы элементов во всех строках и всех столбцах равны между собой.

§ 3. Записи. Комбинированные типы

Довольно часто приходится иметь дело с комбинациями разнотипных объектов: наименование класса состоит из числа (года обучения) и буквы, багаж может характеризоваться целым числом (числом вещей) и действительным числом (общим весом), краткие сведения об ученике школы могут состоять из двух последовательностей букв (имени и фамилии) и наименования класса.

В паскале комбинациями объектов разного типа являются *записи*. Составляющие запись объекты называются ее *полями*. В записи каждое поле имеет свое собственное имя (идентификатор). Если выбраны имена полей *годобуч* и *буква* и если x обозначает некоторую запись, то поля этой записи обозначаются через *x.годобуч* и *x.буква*. Количество полей, тип и имя каждого поля фиксируется в определении типа, к которому относится запись. Определение, имеющее вид

класс = **record** *годобуч* : *integer*; *буква* : *char* **end** *),

это определение типа, имя которого *класс*. Объектами типа *класс* будут записи, состоящие из двух полей, одно из которых имеет тип *integer*, а другое — *char*. Пусть переменная x описана в программе как переменная типа *класс*:

var x : *класс*; ...

Тогда при выполнении программы значениями x будут записи с числом полей, равным 2. Переменная x — это переменная типа *класс*, переменные *x.годобуч* и *x.буква* — это переменные типа *integer* и типа *char* соответ-

*) record — запись.

ственно. С переменной *x.годобуч* можно обращаться, как с обычной переменной типа *integer*, а с переменной *x.буква* — как с обычной переменной типа *char*. Операции над объектами типа *класс* — это доступ к отдельным полям через имена полей и изменение отдельных полей с помощью операций, связанных с типами *integer* и *char*.

Пример. Программа, позволяющая определить, являются ли два класса параллельными:

```
program пар(input, output);
  type класс = record годобуч : integer;
  буква : char end;
  var x, y : класс;
  begin read(x.годобуч, x.буква, y.годобуч, y.буква);
    if (x.годобуч = y.годобуч) and (x.буква ≠ y.буква)
      then write('да') else write('нет')
  end.
```

Резюмируем и дополним сказанное. Если тип v определен в программе с помощью конструкции **record** ... **end**, то он называется *комбинированным* типом. Общий вид определения комбинированного типа v есть

v = **record** $l_1 : r_1; l_2 : r_2; \ldots; l_k : r_k$ **end**

где каждое из $r_1, r_2, \ldots, r_k$ — это *real*, *integer*, *char* или имя ранее определенного типа, а каждое из $l_1, \ldots, l_k$ — это либо имя поля (идентификатор), либо несколько имен полей, перечисленных через запятую. Типы $r_1, r_2, \ldots, \ldots, r_k$ называются базовыми по отношению к типу v. Если переменная x — это переменная типа v, и, например, l_1 имеет вид $p, q, \ldots, s$, то $x.p, x.q, \ldots, x.s$ — это переменные типа r_1 и т. д. Операции над объектами типа v — это доступ к отдельным полям записей посредством указания имен полей, а также изменение отдельных полей с помощью операций, связанных с базовыми типами.

Для переменных x, y одного и того же комбинированного типа можно использовать оператор присваивания $x := y$.

Воспользуемся возможностью привлечения типа, который определен в программе, в качестве базового типа. Мы можем определить тип *ученик*. В записях, являющихся объектами этого типа, будут фиксироваться

фамилия, имя ученика (массивы длины 15 с элементами типа *char*) и *класс* (тип *класс* определен так же, как в программе *пар*):

фи = **array** [1 .. 15] **of** *char*;

ученик = **record** *ф*, *и* : *фи*; *кл* : *класс* **end**;

Тогда, если *x* — переменная типа *ученик*, то

x.и, *x.ф.* — переменные типа *фи*,
x.и[*j*], *x.ф*[*j*] — переменные типа *char*,
x.кл — переменная типа *класс*,
x.кл.годобуч — переменная типа *integer*,
x.кл.буква — переменная типа *char*.

Можно рассмотреть массивы с элементами типа *ученик*:

школа = **array** [1 .. 500] **of** *ученик*,

тогда, если *s* — переменная типа *школа*, то

s[*i*] — переменная типа *ученик*,
s[*i*].*и*, s[*i*].*ф* — переменные типа *фи*,
s[*i*].*и*[*j*], *s*[*i*].*ф*[*j*] — переменные типа *char*,
s[*i*].*кл* — переменная типа *класс*,
s[*i*].*кл.годобуч* — переменная типа *integer*,
s[*i*].*кл.буква* — переменная типа *char*.

Напишем программу, позволяющую определить, есть ли в школе в каких-либо параллельных классах однофамильцы. Данные об ученике идут в следующем порядке: имя, фамилия, год обучения, буква. Если фамилия или имя записываются менее чем 15 буквами, то оставшиеся концевые элементы — это пробелы. Данные о разных учениках идут в некоторой очередности, о которой заранее ничего не известно.

```
program оф(input, output);
 label 1,2;
 type класс = record годобуч : integer;
                     буква : char end;
      фи = array [1 .. 15] of char;
      ученик = record ф, и : фи; кл : класс end;
      школа = array [1 .. 500] of ученик;
```

```
var s : школа; i, j, k : integer;
begin
  for i:=1 to 500 do
   begin for j:=1 to 15 do read(s[i].u[j]);
     for j:=1 to 15 do read(s[i].ф[j]);
     read(s[i].кл. годобуч, s[i].кл. буква)
   end;
  for i:=1 to 500 do
        for j:=i+1 to 500 do
begin if (s[i].кл. годобуч=s[j].кл. годобуч)
    and (s[i].кл. буква ≠ s[j].кл. буква) then
    begin for k:=1 to 15 do
        if s[i].ф[k]≠s[j].ф[k] then goto 2;
        write ('есть'); goto 1 end;
2 : end;  write('нет');
1 : end.
```

Пояснение: вначале выполняется ввод данных, и 500 элементов массива *s* получают свои значения; при поиске однофамильцев *s*[*i*] сопоставляется с *s*[*i* + 1], ..., ..., *s*[500].

На примерах последних параграфов видно, что выбор и определение типов — это важный этап составления программы на паскале.

ЗАДАЧИ

В задачах 1—3 требуется переделать программу *оф* так, чтобы получились программы для выполнения следующих заданий.

1. Поиск однофамильцев, обучающихся в одном каком-нибудь классе.

2. Поиск двух учащихся школы, у которых совпадают имя и фамилия.

3. Вывод фамилий и первых букв имен всех учеников 9а.

4. Приняв способ изображения рационального числа в виде записи с двумя полями *числ*, *знам*: *integer*, написать программу, позволяющую, во-первых, определить,

есть ли среди 50 рациональных чисел равные и, во-вторых, вычислить наибольшее из данных рациональных чисел (числа не обязательно имеют несократимую форму).

5. Определить в программе тип *багаж* (см. начало этого параграфа). Написать всю программу так, чтобы она позволяла выяснить, имеется ли среди 80 записей типа *багаж* такая, поля которой превосходят соответствующие поля всех остальных записей.

§ 4. Файлы. Файловые типы

Большие совокупности данных, как, например, сведения обо всех учениках школы или, тем более, адреса всех жителей города, удобно иметь записанными в виде последовательности сигналов на магнитной ленте или магнитном диске. Лента или диск могут вместить очень большой объем информации — значительно больший, чем тот, который помещается в памяти вычислительной машины. Ленты и диски устанавливаются на магнитофоны вычислительной машины перед запуском программы обработки данных. Результаты выполнения программы тоже могут быть записаны на ленту или диск и впоследствии использованы другими программами.

В паскале предусмотрены специальные объекты — *файлы*, операции над которыми сводятся к работе с лентами и дисками. *Файл* — это последовательность *компонент*, являющихся объектами одного и того же типа. Количество компонент в файле заранее не оговаривается, компоненты файла не имеют индексов. До некоторой компоненты можно добраться, только перебрав по очереди все предыдущие компоненты.

Пример. Определение, имеющее вид

v = **file of** *integer* *)

это определение типа, имя которого *v*. Объектами типа *v* будут файлы с целочисленными компонентами.

Объясняя принципы работы с файлами, будем для простоты и наглядности считать, что каждый файл записан на своей собственной ленте, при этом с самого начала ленты записано имя файла (идентификатор), затем идут компоненты, а после самой последней компоненты записан *признак конца файла*. На рис. 9, а прямоуголь-

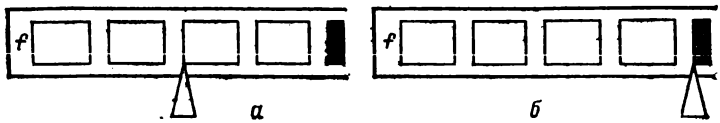
*) file — подшивка, картотека.

ники изображают компоненты, вертикальный отрезок — признак конца. В качестве имени файла здесь и в следующих рисунках использован идентификатор *f*. На рис. 9, *б* изображен файл, не имеющий ни одной компоненты (такая возможность допускается), — *пустой файл*.

а *б*

Рис. 9

В те моменты, когда на магнитофоне не происходит ни чтения, ни записи, головка чтения — записи совмещена либо с началом некоторой компоненты, либо с признаком конца. На рис. 10, *а* и *б* представлены эти возможности, головка изображена треугольником.



а *б*

Рис. 10

Теперь опишем операции над файлами. Пусть *f* — имя рассматриваемого файла, и пусть *a* — переменная того типа, объектами которого являются компоненты файла. Отметим предварительно, что файл может быть *открыт для записи* и при этом *закрыт для чтения*, может быть *открыт для чтения* и при этом *закрыт для записи*, а может быть закрыт и для записи и для чтения. Подробности будут сообщены ниже.

Запись в файл. Эта операция возможна, только когда файл открыт для записи и головка совмещена с признаком конца. При соблюдении этих условий выполнение оператора *write*(*f*, *a*) приведет к тому, что в файл будет записана еще одна компонента, равная значению переменной *a* *). Признак конца будет записан после новой компоненты. Головка будет совмещена с признаком

*) Перед выполнением оператора *write*(*f*, *a*) проверяется, описан ли идентификатор *f* как переменная, значением которой должен быть файл. Если да, то происходит запись в файл, иначе оператор *write*(*f*, *a*) выполняется как обычный оператор вывода. Это же касается оператора *read*(*f*, *a*), о котором речь пойдет ниже.

конца. Фазы выполнения этой операции изображены на рис. 11, *а* и *б*.

Подготовка к записи с начала файла. В результате выполнения оператора *rewrite*(*f*)*) все компоненты

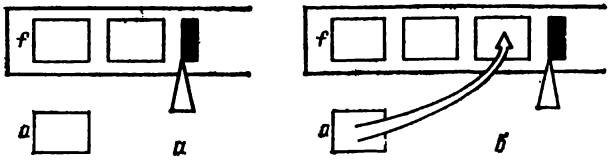


Рис. 11

файла пропадают, признак конца помещается в самое начало, головка совмещается с признаком конца. Файл становится открытым для записи и закрытым для чтения. Фазы выполнения этой операции изображены на рис. 12, *а* и *б*.

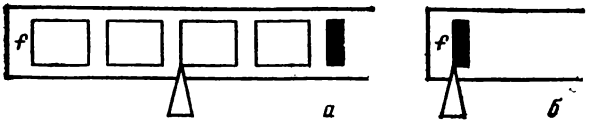


Рис. 12

Пример. Пусть *f* — файл, компонентами которого могут быть целые числа. Приведем фрагмент программы, обеспечивающий запись в *f* квадратов ста первых натуральных чисел:

rewrite(*f*); **for** *i*:= 1 **to** 100 **do**
begin *j* = *sqr*(*i*); *write*(*f*, *j*) **end**

переменные *i* и *j* должны иметь тип *integer*.

Чтение из файла. Эта операция возможна, только когда файл открыт для чтения и головка совмещена с началом некоторой компоненты. При соблюдении этих условий после выполнения оператора *read*(*f*, *a*) переменная *a* будет иметь значение, равное той компоненте файла, с началом которой была совмещена головка. После выполнения оператора головка будет совмещена с началом следующей компоненты, если же следующей компоненты нет, то она будет совмещена с признаком

*) rewrite — перезапись.

конца. Фазы выполнения этой операций изображены на рис. 13, *а* и *б*.

Подготовка к чтению с начала файла. Если файл не пуст, то после выполнения оператора *reset*(*f*)*) головка будет совмещена с началом первой компоненты файла.

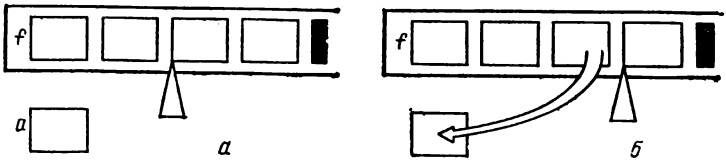


Рис. 13

Две фазы показаны на рис. 14, *а* и *б*. Если файл пуст, то головка указывает на признак конца, расположенный после имени файла (рис. 12, *б*).

Распознавание конца файла. Условие *eof*(*f*)**) можно использовать в условном операторе после **if** и в опера-

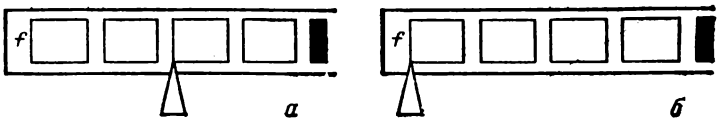


Рис. 14

торе цикла после **while**. Это условие оказывается удовлетворенным тогда и только тогда, когда головка совмещена с признаком конца файла *f*.

Пример. Выполнение оператора

if *eof*(*f*) **then goto** 1

повлечет переход к оператору, помеченному меткой **1**, тогда и только тогда, когда головка совмещена с признаком конца.

Часто бывает нужно после **if** или после **while** поместить не условие конца файла, а условие того, что файл не закончен. Это делается таким образом:

if not *eof*(*f*) **then** *read*(*f*, *a*)

---

*) reset — переустановка.

**) eof — сокращение от end of file — конец файла.

Пример. Фрагмент программы, обеспечивающий чтение из файла *f*, компонентами которого служат целые числа, всех его компонент и вычисление суммы их квадратов:

```
reset(f); s:=0;
while not eof(f) do
  begin
    read(f, i); s:=sqr(i)+s
  end
```

переменные *s* и *i* должны иметь тип *integer*.

Резюмируем и дополним сказанное. Если тип *v* определен в программе с помощью конструкции **file of** ..., то он называется *файловым типом*. Общий вид определения файлового типа есть

v = **file of** *r*

r — это *real*, *integer*, *char* или название ранее определенного типа. Тип *r* называется базовым по отношению к типу *v*. Файловый тип не может быть базовым по отношению к другому типу (регулярному, комбинированному, файловому). Операции над значениями типа *v* — это последовательный доступ к компонентам файлов и запись компонент типа *r* с начала файла. Сравнение значений файлового типа с помощью операций $=$, $\neq$ производить нельзя. Не разрешены и присваивания $a := b$. Имя каждого файла, который будет использован в данной программе, должно быть указано в скобках после названия программы. Например,

program *P* (*input*, *output*, *f*)

или

program *P* (*f*1, *output*, *f*2)

и т. д. Программа должна содержать описания тех файловых типов, к которым принадлежат файлы, упомянутые после имени программы. Идентификаторы, служащие именами этих файлов, должны быть описаны в программе как переменные соответствующих типов *).

*) Таким образом, имя файла — это и переменная, значением которой является файл, и, одновременно, обозначение ленты, на которой записан файл.

Пример. Программа, в результате выполнения которой выводятся все четные числа из данного файла *int* с целочисленными компонентами

```
program rem(int, output);
 type v = file of integer;
 var int : v; i : integer;
 begin reset(int);
   while not eof(int) do
     begin
       read(int, i);
       if i mod 2 = 0 then writeln(i)
     end
 end.
```

Пример. Пусть *school* — это файл, компоненты которого имеют тип *ученик* (см. предыдущий параграф). Приведем программу вывода фамилий и имен учеников класса 9а.

```
program фи9а(school, output);
 type класс = record годобуч : integer;
                     буква : char end;
      фи = array [1 .. 15] of char;
      ученик = record ф, и : фи; кл : класс end;
      v = file of ученик;
 var school : v; i : integer; x : ученик;
 begin reset(school);
   while not eof(school) do
     begin read(school, x);
     if (x.кл.годобуч = 9) and (x.кл.буква='а') then
       begin
         for i := 1 to 15 do write(x.ф[i]);
         write('_');
         for i := 1 to 15 do write(x.и[i]);
         writeln('_')
       end
     end
 end.
```

Если к компонентам файла *f1* надо добавить еще какие-то компоненты, то для этого приходится переносить компоненты файла *f1* и новые компоненты в дополнительный файл, а потом, если нужно, из дополнительного файла переносить все компоненты в файл *f1*. Программа добавления к целочисленному файлу числа 100:

```
program p100(f1, f2);
 type t=file of integer;
 var f1, f2 : t; a : integer;
 begin reset(f1); rewrite(f2);
   while not eof(f1) do
     begin read(f1, a); write(f2, a) end;
   a:=100; write(f2, a)
 end.
```

Если все компоненты следует собрать в файле *f1*, то в конце программы надо поместить еще операторы

```
rewrite(f1); reset(f2);
while not eof(f2) do
  begin read(f2, a); write(f1, a) end.
```

Появление имен файлов рядом с идентификаторами *input* и *output* объясняется тем, что *input* и *output* — это имена специальных файлов (совершенно особых по способу хранения и обращения с ними), эти файлы называются *входным* и *выходным*.

К входному файлу можно применять *eof*, т. е. в программе можно, например, использовать оператор цикла

```
while not eof(input) do
  begin
    read(a); s:=s+a
  end
```

условный оператор

```
if eof(input) then goto 1
```

и т. д.

ЗАДАЧИ

1. Написать программу, в ходе выполнения которой файл *int* заполняется теми целыми числами из файла *num*, которые являются полными квадратами.

2. Дан файл, компонентами которого являются массивы по 100 целых чисел; элементы массивов имеют индексы 0, ..., 99. Написать программы, в результате выполнения которых в каждом массиве определяется наименьший элемент и

а) выводится один из массивов, обладающий наибольшим из всех наименьших элементов,

б) в файле *maxmin* собираются все такие массивы.

3. Написать программу, в ходе выполнения которой компоненты файла *f*1 (действительные числа) переписываются в файл *f*2 в обратном порядке.

4. Компоненты файла *ассортимент* являются объектами типа *игрушка*:

```
type название = array [1 .. 15] of char;
     диапазон = record мл, ст : integer end;
     игрушка = record н : название; ц : integer;
                      д : диапазон end;
```

Предполагается, что поле *н* — это название игрушки (кукла, конструктор, кубики и т. д., незанятый буквами конец массива заполнен пробелами), поле *ц* — цена в копейках (например, 25,480 и т. д.), поле *д* — возрастные границы. Написать программы, в результате выполнения которых выдаются следующие сведения:

а) название игрушек, стоимость которых не превышает 4 р. и которые подходят детям 5 лет,

б) стоимость самого дорогого конструктора,

в) название игрушек, которые подходят как детям 4 лет, так и детям 10 лет.

5. Исходя из условия предыдущей задачи, написать программу, в результате выполнения которой проверяется, есть ли в файле компонента со следующей информацией: мяч ценой в 2 р. 50 к. подходит детям от трехлетнего до восьмилетнего возраста. Если нет, то такая компонента должна быть внесена в файл *ассортимент*.

Полезно вернуться к уже рассмотренным задачам из первой части книги: 4, 5 из § 5 гл. I; 2 из § 2 гл. II; 1—3 из § 4 гл. II и решить их без предположения о том, что последовательность исходных данных заканчивается специальным сигнальным числом или символом. В этом предположении отпадает необходимость ввиду того, что можно воспользоваться условием *eof*(*input*).

Глава IV

ПРОЦЕДУРЫ И ФУНКЦИИ

§ 1. Процедуры без параметров. Параметры — переменные

При составлении программы иногда получается так, что, по сути дела, одну и ту же последовательность операторов надо выписать несколько раз. Рассмотрим пример: пусть требуется составить программу вычисления площади выпуклого четырехугольника, заданного длинами четырех сторон и диагонали (рис. 15).

Диагональ делит выпуклый четырехугольник на два треугольника, к которым применима формула Герона

$$s_{\triangle} = \sqrt{p(p-a)(p-b)(p-c)},$$

где a, b, c — длины сторон треугольника, $p=(a+b+c)/2$. Простейшее решение — дважды выписать в программе операторы, задающие вычисления по этой формуле. Однако можно избежать этого повторения, если воспользоваться предоставляемой паскалем возможностью введения имени для составного оператора (а из любой группы операторов всегда можно сделать один составной оператор). После того как имя введено, в нужных местах программы помещается не сам оператор, а его имя.

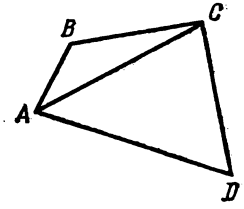


Рис. 15

Оператор, для которого введено имя, представляет собой *процедуру*. Использование в программе имени процедуры в качестве оператора называется *обращением к процедуре* или *оператором процедуры*.

Для того чтобы ввести имя (идентификатор) P для составного оператора S, достаточно включить в программу *описание процедуры*, имеющее вид

procedure P; S; *)

*) procedure — процедура.

В программе может содержаться несколько описаний различных процедур, все они, одно за другим, располагаются после совокупности описаний переменных.

Напишем первый вариант программы вычисления площади четырехугольника:

```
program F1 (input, output);
 var AB, BC, CD, DA, AC, s1, s, a, b, c, p : real;
 procedure str1;
   begin p:=(a + b + c)/2;
     s := sqrt (p * (p — a) * (p — b) * (p — c))
   end;
 begin read (AB, BC, CD, DA, AC);
   a := AB; b := BC; c := AC; str1; s1 := s;
   a := DA; b := AC; c := CD; str1; s1 := s1 + s;
   write (s1)
 end.
```

В программе — два обращения к процедуре *str*1. Непосредственно перед каждым обращением идет группа операторов присваивания, задающих значения переменных *a*, *b* и *c*; каждое обращение влечет выполнение процедуры; после обращения к процедуре переменная *s* имеет значение площади соответствующего треугольника.

Итак, связь процедуры *str*1 с остальными операторами программы осуществляется через переменные *a*, *b*, *c*, *s*; в процедуре имеется еще переменная *p*, которая является вспомогательной внутри процедуры. Можно избежать случайного влияния изменения значений вспомогательных переменных процедуры на выполнение других операторов. Паскалем разрешается включать в описание процедуры совокупность описаний переменных:

```
procedure str2;
 var p : real;
 begin
   p := (a + b + c)/2;
   s := sqrt (p * (p — a) * (p — b) * (p — c))
 end;
```

Переменная, описанная в процедуре, называется *локальной* по отношению к процедуре. Слово «локальная» означает местная, имеющая местное назначение. Если переменная p — локальная по отношению к некоторой процедуре, то при выполнении этой процедуры работа с переменной p никак не будет влиять на значение переменной p, описанной в программе вне процедуры (если такая переменная имеется). Как только процедура будет выполнена, значение локальной переменной p забудется. Аналогично, в процедуре могут описываться и использоваться локальные метки. Типы и процедуры тоже могут быть локальными по отношению к процедуре (мы этой возможностью пользоваться не будем). Введение локальных меток, переменных и т. д. делает оформление процедуры похожим на оформление программы.

Если в программе *F*1 использовать вместо процедуры *str*1 процедуру *str*2, то можно из описания переменных программы удалить переменную p. Но можно и оставить описание прежним.

Продемонстрированный способ использования процедур не удобен из-за большого числа операторов присваивания, которые до обращения к процедуре определяют значения переменных a, b и c. Есть другой способ — описать процедуру с параметрами (аргументами), что позволит обращаться к процедуре, например, так:

*str*3(*AB*, *BC*, *AC*, *s*1)

Это обращение обеспечит вычисление площади треугольника со сторонами *AB*, *BC*, *AC* и присваивание значения площади переменной *s*1. Описание может выглядеть следующим образом:

```
procedure str3(var a, b, c, s : real);
  var p : real;
  begin p:=(a + b + c)/2;
    s:=sqrt(p*(p - a)*(p - b)*(p - c))
  end;
```

указанные в скобках после имени процедуры параметры a, b, c, s — это *формальные параметры*, при выполнении программы в момент обращения к процедуре они заменяются *фактическими параметрами*: первый формальный параметр заменяется первым фактическим параметром, второй формальный параметр — вторым фактическим и т. д.; фактическим параметром при таком описании процедур может быть переменная того типа,

который указан при соответствующем ему формальном параметре. В данном примере фактические параметры должны быть переменными типа *real*. После подстановки фактических параметров на место формальных параметров процедура выполняется.

Формальные параметры не описываются.

Дадим еще один вариант программы вычисления площади четырехугольника:

```
program F2(input, output);
 var AB, BC, CD, DA, AC, s1, s2 : real;
 procedure str3(var a, b, c, s : real);
   var p : real;
   begin p := (a + b + c)/2;
     s := sqrt(p * (p - a) * (p - b) * (p - c))
   end;
 begin read(AB, BC, CD, DA, AC);
   str3(AB, BC, AC, s1);
   str3(CD, DA, AC, s2);
   write(s1 + s2)
 end.
```

Первое обращение к процедуре приведет к выполнению оператора

```
begin p := (AB + BC + AC)/2;
  s1 := sqrt(p * (p - AB) * (p - BC) * (p - AC))
end
```

второе обращение — к выполнению оператора

```
begin p := (CD + DA + AC)/2;
  s2 := sqrt(p * (p - CD) * (p - DA) * (p - AC))
end
```

Резюмируем и дополним сказанное. Процедура — это часть программы (составной оператор) S, получающая некоторое имя (идентификатор) p с помощью описания процедуры. Описание процедуры начинается с *заголовка*

```
procedure p;
```

или

```
procedure p(l1; l2; ...; lm);
```

здесь $l_1, l_2, \ldots, l_m$ — группы формальных параметров. В группе формальных параметров **var** *a, b, ..., x : t* идентификаторы *a, b, ..., x* и есть, собственно, формальные параметры, а *t* должно быть именем типа. Первый заголовок — это заголовок *процедуры без параметров*, второй — заголовок *процедуры с параметрами*, точнее — заголовок процедуры с *параметрами — переменными*. Пример заголовка процедуры с параметрами — переменными

procedure *q* (**var** *a* : *t*; **var** *b, c* : *char*; **var** *d* : *t*;
var *e* : *real*);

Вслед за заголовком могут идти описания локальных меток и локальных переменных. Но эти описания могут и отсутствовать. Далее идет процедура, т. е. оператор, который получает имя. Описание процедуры заканчивается точкой с запятой.

Все описания процедур идут одно за другим после совокупности описаний переменных. В процедуре могут содержаться обращения к ранее описанным процедурам.

Обращение в программе к процедуре без параметров влечет непосредственное выполнение процедуры. Обращение к процедуре с параметрами — переменными влечет подстановку в процедуру переменных, указанных в качестве фактических параметров, на место формальных параметров, а затем — выполнение процедуры. Тип фактического параметра должен совпадать с типом, сопоставленным в заголовке формальному параметру. Обращение к процедуре *q* (см. выше пример заголовка) может иметь вид

q(*v*[*j* + 1], *b*1, *b*2, *u*, *r*[*j*].*st*)

при этом переменные *v*[*j* + 1], *u* обязаны иметь тип *t*, переменные *b*1, *b*2 — тип *char*, переменная *r*[*j*].*st* — тип *real*.

Если в обращении к процедуре с параметрами — переменными фигурируют переменные, включающие индексы, как, например, *v*[*j* + 1], *r*[*j*].*st*, то перед подстановкой этих фактических параметров на место формальных параметров все индексы заменяются их числовыми значениями. При *j* = 3 будут подставлены *v*[4] и *r*[3].*st*.

Локальные переменные и метки доступны только внутри процедуры. Значения локальных переменных после окончания выполнения процедуры забываются.

Приведем пример программы с процедурой, которая предназначена для проверки гипотезы Гольдбаха для данного четного n. Эта гипотеза (по сегодняшний день не опровергнутая и полностью не доказанная) заключается в том, что каждое четное число, большее двух, представляется в виде суммы двух простых чисел. Проверка может быть проведена так: для каждого $i = 2, 3, \ldots, n/2$ выясняем, является ли оно простым, и если да, то дополнительно выясняем, является ли простым $n - i$. Напишем отдельно группу операторов, выполнение которой приводит к присваиванию переменной p значения 1 (коль скоро значение $k \geqslant 2$ является простым) и к присваиванию переменной p значения 0 в противном случае. Воспользуемся тем, что каждому большему или равному $\sqrt{k}$ делителю числа k соответствует меньший или равный $\sqrt{k}$ делитель числа k:

```
l := round (sqrt (k)); p := 1;
for i := 2 to l do
  if k mod i = 0 then begin p := 0; goto 1 end; 1 :
```

(последний оператор пустой, помеченный меткой 1).

Эту последовательность операторов оформим в программе проверки гипотезы Гольдбаха как процедуру *pr*:

```
program Goldbach (input, output);
 label 0, 10;
 var n, s, i, m : integer;
 procedure pr (var k, p : integer);
   label 1;
   var l, i : integer;
   begin l := round (sqrt (k)); p := 1;
     for i := 2 to l do
       if k mod i = 0 then
         begin p := 0; goto 1 end;
1 : end;
 begin read (n);
   for i := 2 to n div 2 do
     begin pr (i, s);
       if s = 0 then goto 0;
```

```
        m := n - i; pr(m, s);
        if s = 0 then goto 0;
        write(i, '_', m); goto 10;
    0 : end;
    write(n)
10 : end.
```

Выполнение этой программы приводит к выводу самого числа *n*, если пары простых слагаемых не найдено. Если же простые слагаемые удалось найти, то будут выведены они. Отметим некоторые важные моменты, связанные с использованием процедуры *pr*:

1) локальная, по отношению к процедуре *pr* переменная *i* не смешивается с переменной *i*, описанной вне процедуры;

2) первое обращение, имеющее вид *pr*(*i*, *s*), несколько рискованно — нужно обязательно предварительно убедиться, что такое обращение не повлечет изменения значения переменной *i*, являющейся параметром цикла в основной программе. Но в данном случае опасности нет, так как формальный параметр *k* не встречается в процедуре слева от знака :=;

3) перед вторым обращением пришлось поместить оператор присваивания *m* := *n* — *i*, так как формальному параметру — переменной может соответствовать в качестве фактического параметра только переменная, и мы не можем написать *pr*(*n* — *i*, *s*).

В следующем параграфе будет указан способ преодоления неудобств, отмеченных в двух последних пунктах.

ЗАДАЧИ

1. Вернемся к программе *фи9а* из последнего параграфа предыдущей главы; потребуем теперь, чтобы выполнение программы приводило к печати фамилий и имен столбиком, каждая строка которого выглядит так: фамилия, 1 пробел, имя, 1 пробел, запятая (после последней строки — точка). Описать процедуру *печать* с одним параметром, значением которого является массив с элементами типа *char*, имеющими индексы от 1 до 15. При обращении к процедуре должны печататься элементы массива до появления пробела.

2. Переделать программу *Goldbach* и процедуру *pr* так, чтобы выполнение процедуры приводило, если число составное, к переходу к метке 0 (не локальной по отношению к процедуре). Если же число простое, то должен происходить естественный переход к следующему в программе оператору. Оператор цикла основной программы можно будет записать так:

```
for i:=2 to n div 2 do
  begin pr1(i); m:=n-i; pr1(m);
    write(i, '_', m); goto 10;
0 : end
```

3. Написать программу проверки существования «близнецов», т. е. простых чисел, разность между которыми равна двум, среди чисел n, $n+1$, ..., $2n$, где n — данное число (по сегодняшний день неизвестно, бесконечно ли множество пар близнецов).

4. а) Написать программу, в ходе выполнения которой компоненты файла *f1* переписываются в файл *f2*, а компоненты файла *f2* — в файл *f1*. Использовать файл *h* как вспомогательный. Компоненты всех файлов имеют тип *real*. В паскале не разрешены операторы присваивания вида $f:=g$, где f и g — названия файлов, поэтому следует описать процедуру присваивания *присв*(f, g).

б) С помощью процедуры *присв*(f, g) (см. задачу а) написать программу, в ходе выполнения которой файлы *f1*, *f2*, *f3*, *f4*, *f5* обмениваются компонентами в соответствии со следующей схемой:

f1	*f2*	*f3*	*f4*	*f5*
f3	*f4*	*f5*	*f2*	*f1*

т. е. компоненты файла *f1* переписываются в файл *f3*, компоненты файла *f2* переписываются в файл *f4* и т. д. Разрешается использовать только один дополнительный файл.

5. Написать программу вычисления по схеме Горнера значения многочлена с рациональными коэффициентами для данного рационального значения переменной. Считать, что коэффициенты многочлена записаны по порядку в данном файле *coef*: первая компонента — старший коэффициент, ..., последняя компонента — свободный член. Рациональное число — это запись с двумя целочисленными полями *числ*, *знам*. Описать процедуры

полного сокращения рационального числа, сложения двух рациональных чисел, умножения двух рациональных чисел. Последние две процедуры содержат обращения к первой.

§ 2. Параметры — значения

Заголовок процедуры может быть устроен так, что некоторые группы формальных параметров не содержат слова **var**. Например,

procedure *str4* (*a*, *b*, *c* : *real*; **var** *s* : *real*);

procedure *prim* (*k* : *integer*; **var** *p* : *integer*);

и т. д. Формальные параметры, которые входят в группы, не содержащие слова **var**, называются *формальными параметрами — значениями*. В первом примере *a*, *b*, *c* — это формальные параметры — значения, а *s* — это формальный параметр — переменная. Во втором примере *k* — это формальный параметр — значение, а *p* — это формальный параметр — переменная.

Фактическим параметром, соответствующим такому формальному параметру — значению, при котором указан тип *real*, *integer* или *char*, может быть не только переменная. Так, если при формальном параметре — значении указан тип *real* (в первом примере такими формальными параметрами будут *a*, *b*, *c*), то соответствующим фактическим параметром может быть любое выражение, т. е. переменная типа *real* или *integer*, число или же более сложная конструкция, возможно, содержащая знаки арифметических операций и функций. Если указан тип *integer* (во втором примере таким формальным параметром будет *k*), то соответствующим фактическим параметром может быть любое выражение со значением типа *integer*. К процедуре *str4* можно обратиться, например, так:

str4 (3.14, *x*, *sqrt* (10 — *sqr* (*x*)), *y*)

к процедуре *prim*, в свою очередь, — так:

prim (*n* — *i*, *m*)

Фактическим параметром, соответствующим такому формальному параметру — значению, при котором указан тип *char*, может быть не только переменная типа *char*, но и конкретный символ, взятый в кавычки, например *'a'*, *'*'* и т. д.

Пусть в некоторый момент выполнения программы происходит обращение к процедуре, имеющей параметры — значения. Тогда в начало составного оператора, следующего в описании процедуры вслед за заголовком и, возможно, за описаниями локальных меток и локальных переменных, для каждого формального параметра — значения вставляется оператор присваивания; слева от знака := помещается формальный параметр — значение, справа — соответствующий ему фактический параметр. В получившийся составной оператор на место формальных параметров — переменных подставляются переменные, являющиеся фактическими параметрами. После этого оператор выполняется.

Если описанием процедуры *str*4 служит

```
procedure str4(a, b, c : real; var s : real);
  var p : real;
  begin p:=(a+b+c)/2;
    s:=sqrt(p*(p-a)*(p-b)*(p-c))
  end;
```

то обращение *str*4(3.14, *x*, *sqrt*(10 — *sqr*(*x*)), *y*) повлечет выполнение составного оператора

```
begin a:=3.14; b:=x; c:=sqrt(10-sqr(x));
  p:=(a+b+c)/2;
  y:=sqrt(p*(p-a)*(p-b)*(p-c))
end
```

Вернемся теперь к программе *Goldbach*, рассмотренной в предыдущем параграфе. В новом варианте этой программы опишем процедуру *pr*1, использовав соображение, которое сформулировано в задаче 2 предыдущего параграфа. Это будет процедура с параметром — значением:

```
program Goldbach1(input, output);
  label 0, 10;
  var n, i : integer;
  procedure pr1(k : integer);
    var l, i : integer;
    begin l:=round(sqrt(k));
```

```
      for i:=2 to l do
        if k mod i=0 then goto 0
    end;
  begin read(n);
    for i:=2 to n div 2 do
      begin pr1(i); pr1(n-i);
        write(i, '_', n-i);
        goto 10;
    0 : end;
      write(n);
10 : end.
```

В программе — два обращения к процедуре *pr*1. Первое обращение приводит к выполнению оператора

```
begin k:=i; l:=round(sqrt(k));
  for i:=2 to l do
    if k mod i=0 then goto 0
end
```

второе — к выполнению оператора

```
begin k:=n-i; l:=round(sqrt(k));
  for i:=2 to l do
    if k mod i=0 then goto 0
end
```

Из общих правил о параметрах — переменных и о параметрах — значениях, а также из приведенных примеров видно, что если некоторый формальный параметр изображает результат выполнения процедуры (как, например, параметр *s* в рассмотренных вариантах процедуры вычисления площади треугольника), то этот формальный параметр должен быть формальным параметром — переменной. Формальные параметры, при которых указан файловый тип, запрещается объявлять формальными параметрами — значениями, они обязаны быть формальными параметрами — переменными.

Все формальные параметры, кроме тех, которые изображают результаты выполнения процедуры, и тех, при

которых указан файловый тип, на первых этапах занятий программированием рекомендуется объявлять в программах формальными параметрами — значениями.

В заключение этого параграфа отметим, что если формальный параметр — значение обозначен тем же идентификатором, что и некоторая переменная программы, то при выполнении программы для этого формального параметра выбирается другой идентификатор, и никаких недоразумений не возникает.

Например, формальный параметр процедуры *pr*1, которая содержится в программе *Goldbach*1, мог бы быть обозначен через *n* — это не изменило бы результата выполнения программы. Однако формальный параметр этой процедуры нельзя обозначить через *i*, так как в процедуре есть локальная переменная *i*.

ЗАДАЧИ

1. Написать программу вычисления площади многоугольника, изображенного на рис. 16. Длины известных отрезков написаны рядом с этими отрезками. Предложить два варианта программы:

а) описав процедуру *str*4, рассмотренную в этом параграфе,

б) описав процедуру вычисления площади треугольника *str*5, имеющую три параметра — длины сторон треугольника. В результате обращения к этой процедуре вычисленная площадь должна прибавляться к значению переменной *s*, которая не является параметром и не является локальной переменной по отношению к процедуре.

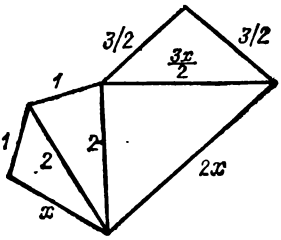


Рис. 16

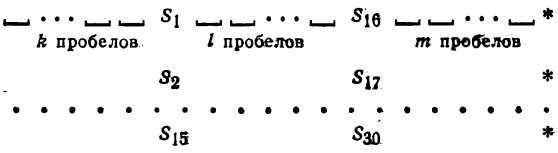
2. Написать программу исследования трех файлов *f*1, *f*2, *f*3 с компонентами типа *ученик* (см. § 3, гл. III). Эти файлы содержат сведения об учащихся трех разных школ. В результате выполнения программы должно выясниться, верно ли, что среди учащихся восьмых классов первой школы есть Иванов, среди учащихся девятых классов второй школы — Петров, среди учащихся десятых классов третьей школы — Сидоров.

3. Написать программу исследования файла *f* с компонентами типа *игрушка* (см. задачу 4 из § 4 гл. III). В результате выполнения этой программы должны быть

выведены цены всех игрушек, сведения о которых имеются в данном файле. Цены должны указываться в рублях и копейках: 3 р. 15 к., 0 р. 07 к., 10 р. 00 к. (число копеек записывается всегда двумя цифрами). В программе полезно описать процедуру печати цены; цена первоначально задана в виде целого числа копеек; например, 315, 7, 1000.

4. Написать программу исследования существования целочисленного корня уравнения с целыми коэффициентами $f_{10}x^{10} + f_9x^9 + \ldots + f_1x + f_0 = 0$. Если $f_0 = 0$, то имеется корень 0, если же $f_0 \neq 0$, то целочисленный корень, если он существует, принадлежит конечному множеству положительных и отрицательных делителей числа f_0. В программе полезно описать процедуру вычисления по схеме Горнера значения многочлена, а также процедуру $d(k, m, l)$, обращение к которой при $m > k \geqslant 0$ приводит к присваиванию переменной l значения наименьшего делителя числа m, содержащегося среди чисел $k+1, k+2, \ldots, m$.

5. Даны целые числа k, l, m и последовательность символов $s_1, \ldots, s_{30}$. Надо написать программу, которая позволяет вывести данные символы в следующем виде:



Полезно описать процедуру *print*(s, n), обращение к которой дает вывод символа s после n пробелов.

§ 3. Функции

В паскале, помимо процедур, разрешены похожие на них конструкции, которые называются *функциями*. Обращение к функции приводит к вычислению ее *значения* — объекта типа *real*, *integer* или *char*. Тип значения фиксируется в описании функции.

Если значение некоторой функции (например, функции $f(a, b)$) имеет тип *real* или *integer*, то способ употребления этой функции аналогичен способу употребления функций *sin*, *cos*, *round* и т. д. Так, в программе могут встретиться, например, операторы присваивания $x := x + f(x, y)$, $y := sin(f(x, x)/2)$ или оператор цикла

while $f(x, y) < f(y, x)$ **do** $x := f(x, x)$

Если же значение некоторой функции (например, функции $h(a)$) имеет тип *char*, то эта функция может быть использована в программе наравне с конкретными символами, взятыми в кавычки. В программе может встретиться, например, оператор присваивания $s := h(i)$, условный оператор

if $s = h(l)$ **then** $s := 'a'$ **else** $s := h(i)$

и т. д.

Имеются две отличительные особенности описания функции в сравнении с описанием процедуры. Первая особенность связана с заголовком: он должен начинаться со служебного слова **function** *) и заканчиваться названием того типа, которому принадлежит значение функции. Например:

function $f(a$: *real*; **var** b : *t*$)$: *real*;

function g(**var** a, b : *integer*) : *integer*;

function $h(a$: *integer*$)$: *char*;

и т. д. Вторая особенность: составной оператор, который располагается в описании функции после заголовка и, возможно, после описания локальных меток и переменных, должен обязательно содержать внутри себя оператор присваивания, в котором слева от знака := помещено имя функции, например:

$f := 3.14$

$g := a + 2 * b$

if a **mod** $2 = 0$ **then** $h := '*'$ **else** $h := 'I'$

и т. д. Таких операторов присваивания может быть несколько, но при каждом конкретном обращении к функции «сработать» должен только один из них. Это присваивание и определит значение функции.

Относительно параметров функций имеется полная аналогия с параметрами процедур. Описания функций, как и описания процедур, располагаются в программе после совокупности описаний переменных.

Программу вычисления площади четырехугольника, варианты которой мы рассматривали в § 1, можно написать и так:

*) function — функция.

```
program F3(input, output);
 var AB, BC, CD, DA, AC : real;
 function triangl(a, b, c : real) : real;
   var p : real;
   begin p := (a + b + c)/2;
     triangl := sqrt(p * (p - a) * (p - b) * (p - c))
   end;
 begin
   read(AB, BC, CD, DA, AC);
   write(triangl(AB, BC, AC) + triangl(CD, DA, AC))
 end.
```

При составлении программы проверки гипотезы Гольдбаха можно было бы воспользоваться функцией *primer*(a), описав ее таким образом, что

$$primer(a) = \begin{cases} 1, & \text{если } a \text{ — простое число,} \\ 0, & \text{если } a \text{ — составное число.} \end{cases}$$

Другой пример. Пусть $a_1, \ldots, a_{90}$ — такая последовательность целых положительных чисел, что $a_1 < \ldots < a_{90}$. Пусть f — файл с компонентами типа *char*. Напишем программу исследования компонент файла f, стоящих на местах с номерами $a_1, \ldots, a_{90}$. В результате выполнения этой программы выводится первая из этих компонент, а также те из последующих компонент, для которых нет равных среди уже выведенных.

В ходе выполнения программы выведенные символы будут поочередно присваиваться переменным $d[1]$, $d[2]$, ... типа *char*. Включим в программу процедуру $w(i, s)$, где значением переменной i является целое неотрицательное число, а значением s — некоторый символ. При выполнении процедуры символ s сравнивается со значениями переменных $d[1], \ldots, d[i]$. Если символ s не совпадает ни с одним из этих значений, то этот символ выводится и присваивается в качестве значения переменной $d[i+1]$, значение переменной i увеличивается на 1. Иначе ничего не выводится, переменной $d[i+1]$ не присваивается никакого значения, и значение переменной i не изменяется. Дополнительно включим в программу функцию $c(n)$, которая может применяться в тех случаях, когда файл f открыт для чтения,

Значением функции будет n-я компонента файла f при отсчете от той компоненты, с началом которой совмещена головка чтения — записи.

```
program af(input, output, f);
 type t1 = array [1 .. 90] of char;
      t2 = file of char;
 var d : t1; f : t2; a, b, i, l : integer;
 function c(n : integer) : char;
   var i : integer; a : char;
   begin for i := 1 to n do read(f, a);
     c := a
   end;
 procedure w(var i : integer; s : char);
   label 1;
   var k : integer;
   begin for k := 1 to i do
     if d[k] = s then goto 1;
     i := i + 1; d[i] := s; write(s, '_');
1 : end;
 begin reset(f); b := 0; i := 0;
   for l := 1 to 90 do
     begin read(a); w(i, c(a - b));
       b := a
     end
 end.
```

Пояснение. Последовательность $a_1, \ldots, a_{90}$ не вводится сразу вся целиком: в поле зрения находятся a и b — два очередных элемента этой последовательности (первоначально полагаем $b = 0$, $a = a_1$).

Обращение $w(i, c(a - b))$ содержит $c(a - b)$ в качестве фактического параметра, соответствующего такому формальному параметру — значению, которому сопоставлен тип *char*. Такое обращение к процедуре w допустимо, так как выше было отмечено, что функция со значением типа *char* может быть использована в про-

грамме наравне с конкретными символами, взятыми в кавычки.

Вычисление значения функции *c* приводит к изменению положения головки чтения — записи относительно файла *f*. Возможны случаи, когда вычисление значения функции приводит к изменению значений некоторых переменных, которые являются фактическими параметрами, соответствующими формальным параметрам — переменным, и переменных, которые не являются локальными переменными по отношению к функции.

ЗАДАЧИ

1. Написать программу, позволяющую выбрать из четырех данных файлов *f*1, *f*2, *f*3, *f*4, компоненты которых имеют тип *real*, тот, который имеет наибольшее число компонент (если таких файлов более одного, то выбрать один из них). Включить в программу функцию $l(f)$, значение которой равно количеству компонент файла.

2. Написать программу, в результате выполнения которой компоненты типа *char* файла *f*1 переписываются в файл *f*2 в обратном порядке. Воспользоваться функцией *c*, которая была включена в программу *af* этого параграфа.

3. Написать программу проверки существования близнецов (см. задачу 3 из § 1) среди целых чисел n, $n+1$, ..., $2n$, включив в эту программу функцию $spr(k)$, значением которой является первое простое число, большее данного неотрицательного целого числа k.

4. Написать программу поиска среди целых чисел n, $n+1$, ..., $2n$ таких, которые являются суммой двух квадратов. Включить в программу функцию *squ* такую, что

$$squ(k) = \begin{cases} 1, & \text{если } k \text{ — квадрат,} \\ 0 & \text{в противном случае.} \end{cases}$$

5. Составить программу приближенного решения n уравнений: $0.5(a_i x)^4 - \cos x = 0$, $i = 1, 2, \ldots, n$. Каждое из уравнений должно быть решено методом деления отрезка пополам. Корни разыскиваются на отрезке $[0, \pi/2]$, точность решения i-го уравнения равна $1/(i+3)$. В программе описать функцию $f(t, x) = 0.5(tx)^4 - \cos x$ и функцию $root(t, eps)$ вычисления приближенного значения корня уравнения $f(t, x) = 0$ при фиксированном t на отрезке $[0, \pi/2]$ с точностью *eps* методом деления пополам.

Глава V

ССЫЛКИ, СПИСКИ, ДЕРЕВЬЯ

§ 1. Ссылки. Ссылочные типы

Ранее отмечалось, что все переменные, встречающиеся в программе, должны быть описаны. Перед началом выполнения программы каждой переменной для размещения ее значений выделяется место в памяти вычислительной машины. Размер выделяемого места зависит от типа переменной. Например, очевидно, что для размещения значений переменной регулярного типа **array** [1 .. 10] **of** *integer* требуется места в 10 раз больше, чем для размещения значений переменной типа *integer*. Обращение в программе к объекту, размещенному в некотором месте памяти, происходит с помощью той переменной, которой это место сопоставлено. Соответствие между переменной и сопоставленным ей местом в памяти сохраняется для описанных в программе переменных на всем протяжении выполнения программы.

В паскале имеются средства, позволяющие заниматься отведением и освобождением памяти для размещения объектов того или иного типа непосредственно по ходу выполнения программы. В программе можно определить *ссылочный* тип, например, можно включить в программу следующее определение типа:

$p = \uparrow t$

Объектами типа p являются *ссылки* на места в памяти, выделенные для объектов заданного типа t. Ссылка, по существу, представляет собой адрес начала, т. е. первой ячейки, некоторого места в памяти, выделенного для объекта типа t. Кроме того, объектом ссылочного типа p является и специальная ссылка **nil** *). Это так называемая пустая ссылка, т. е. ссылка, не указывающая ни на какое место в памяти. Пусть в программе описана

*) nil — ничего, ноль.

переменная v ссылочного типа p (ссылочная переменная). Для того чтобы было выделено новое место в памяти для объекта типа t, нужно воспользоваться оператором $new(v)$*). Адрес начала выделенного места присваивается переменной v. Обозначение $v\uparrow$ после этого становится переменной типа t. В ходе выполнения программы значения переменной $v\uparrow$ будут помещаться в то место памяти, ссылка на которое является значением переменной v.

В программе, содержащей определения типов и описания

type *pint* = ↑*integer*; *w* = **array** [1 .. 20] **of** *real*;

p = ↑*w*;

var *n* : *pint*; *u* : *p*;

допустимы операторы

read(*n*↑), $n\uparrow := n\uparrow + 1$, $n\uparrow := 0$,

$u\uparrow[1] := 1$, $u\uparrow[2] := u\uparrow[n\uparrow]$ и т. д.

Для того чтобы отказаться от места в памяти, выделенного для объекта типа t, ссылка на которое является значением переменной v, следует воспользоваться оператором $dispose(v)$**). После выполнения этого оператора значение переменной v становится неопределенным, а соответствующее место в памяти считается свободным. Следует различать случаи, когда значение переменной ссылочного типа не определено и когда значением переменной ссылочного типа является **nil**.

Итак, если тип p определен в программе с помощью конструкции ↑..., то он называется ссылочным типом. Общий вид определения ссылочного типа p есть $p = \uparrow t$, где t — имя типа. При этом говорят, что ссылочный тип p связан с типом t. В роли t может выступать любой тип, кроме файлового. Одним из объектов любого ссылочного типа всегда является пустая ссылка — **nil**. Если значение переменной a типа p определено и отличается от **nil**, то $a\uparrow$ — это переменная типа t и с ней можно выполнять те же действия, что и с обычной переменной типа t. Значения переменных одного и того же ссылочного типа можно сравнивать с помощью операций $=$ и $\neq$. Для переменных a и b одного и того же ссылочного

*) new — новый, новая.

**) От dispose of — избавиться, ликвидировать.

типа можно использовать оператор присваивания $a := b$. В программировании часто приходится в рамках одной программы решать несколько задач с данными разных типов. Употребление в программе переменных ссылочного типа в этом случае — это один из путей экономного использования памяти вычислительной машины; под объект отводится место в памяти только тогда, когда в этом возникает необходимость, и это место освобождается, когда необходимость в использовании соответствующего объекта исчезает.

Пример. Дано 10 000 действительных чисел a_1, $a_2, \ldots, a_{10\,000}$ и 8000 натуральных чисел $b_1, b_2, \ldots, b_{8000}$ таких, что $b_1 \leqslant 8000$, $b_2 \leqslant 8000$, ..., $b_{8000} \leqslant 8000$. Вывести числа a_1, a_2, ..., $a_{10\,000}$ в обратном порядке. Затем вывести то число b_k, номер которого k равен минимальному из чисел b_1, b_2, ..., b_{8000}. Программа:

```
program M(input, output);
 type ra = array [1 .. 10 000] of real;
      ia = array [1 .. 8000] of integer;
      pr = ↑ra; pi = ↑ia;
 var k, i : integer;
     f : pr; g : pi;
 begin new(f);
   for i := 1 to 10 000 do read(f↑[i]);
   for i := 10 000 downto 1 do writeln(f↑[i]);
   dispose(f); new(g);
   read(g↑[1]); k := g↑[1];
   for i := 2 to 8000 do
     begin
       read(g↑[i]);
       if g↑[i] < k then k := g↑[i]
     end;
   write(g↑[k])
 end.
```

В этой программе вначале отводится место для объекта регулярного типа *ra*. Переменные $f\uparrow[1], \ldots, f\uparrow[10\,000]$

получают значения, равные данным действительным числам. После решения первой части задачи — вывода этих чисел в обратном порядке, место в памяти, выделенное для объекта типа *ra*, освобождается с помощью *dispose*(*f*). После этого отводится место в памяти для объекта регулярного типа *ia*. Переменные $g\uparrow[1], \ldots, g\uparrow[8000]$ получают значения, равные данным натуральным числам, и решается вторая часть задачи. Видно, что память для объектов типа *ra* и *ia* выделяется поочередно, так что эти объекты могут располагаться в пересекающихся областях памяти вычислительной машины.

ЗАДАЧИ

1. Во входном файле расположено целое число n, за которым, если оно нечетно, следуют 100 действительных чисел $a_1, a_2, \ldots, a_{100}$, если оно четно — 200 символов $s_1, s_2, \ldots, s_{200}$. Написать программу, в результате выполнения которой выводится $a_1a_{51} + a_2a_{52} + \ldots + a_{50}a_{100}$ в случае нечетного n, или последовательность символов, начиная с последнего вхождения буквы a в строку $s_1s_2 \ldots s_{200}$ до конца строки, в случае четного n.

2. Во входном файле расположен символ, за которым следуют 20 различных целых чисел, если этот символ есть *i*, или 30 различных действительных чисел, если этот символ есть *r*. Написать программу, в результате выполнения которой выводится часть данной последовательности чисел, начиная с первого по порядку и заканчивая минимальным из данных.

3. Компоненты файла *ассортимент* являются объектами типа *игрушка* (см. задачу 4 § 4 гл. III), а компоненты файла *грузы* — объектами типа *багаж* (см. задачу 5 § 3 гл. III). Во входном файле расположена последовательность $s_1a_1s_2a_2 \ldots s_{50}a_{50}$, где каждое s_i — это символ *и* или *б*, а каждое a_i — это натуральное число. Написать программу, в результате выполнения которой для каждого $i = 1, 2, \ldots, 50$ выводится: если s_i есть символ *и* — название и стоимость игрушки, информация о которой содержится в a_i-й компоненте файла *ассортимент*; если s_i есть символ *б* — количество мест и вес багажа, информация о котором содержится в a_i-й компоненте файла *грузы*.

Программу составить так, чтобы в каждый момент ее выполнения было выделено место в памяти не более

чем для одного объекта комбинированного типа *игрушка* или *багаж*.

4. Какое число будет выведено в результате выполнения программы?

```
program E(output);
 type pint = ↑integer;
 var p, q, s, t : pint;
 begin new(p); new(q); new(s);
   p↑ := 1; q↑ := 2; s↑ := p↑ * 2 + q↑;
   t := p; p := q; q := s; s := t;
   t↑ := t↑ * p↑; write(s↑)
 end.
```

5. Какие числа будут выведены в результате выполнения программы

```
program T(input, output);
 type arr = array [1 .. 4] of real;
      pa = ↑arr;
 var p, q, s : pa; i, j : integer;
 begin new(p); new(q);
   for i := 1 to 4 do read(p↑[i]);
   for i := 4 downto 1 do read(q↑[i]);
   for i := 1 to 2 do
     begin
       for j := 1 to 4 do write(p↑[j]/q↑[j]);
       writeln('_'); s := p; p := q; q := s
     end
 end.
```

если во входном файле расположены действительные числа 1, 2, 3, 4, 5, 6, 8, 8?

§ 2. Списки

Главная возможность, которую предоставляет наличие ссылочных типов и ссылочных переменных в паскале, — это возможность построения с их помощью объектов со сложной, меняющейся структурой. Примером

таких объектов могут служить списки, о которых и пойдет речь в этом параграфе. Вначале оговорим графический способ изображения объектов, который будет использоваться в иллюстрациях.

Место в памяти, выделенное для объекта некоторого типа, будем изображать прямоугольником, в котором помещается сам объект. Иногда рядом с прямоугольником мы будем указывать ту переменную, которой сопоставлено это место в памяти. Отсутствие в прямоугольнике объекта будет означать, что значение соответствующей переменной не определено. Ссылку будем обозначать стрелкой, которая начинается в прямоугольнике, соответствующем переменной ссылочного типа, и указывает на прямоугольник, обозначающий выделенное с помощью *new* место в памяти. Если некоторая ссылочная переменная имеет значение **nil**, то будем записывать **nil** непосредственно в соответствующем этой переменной прямоугольнике. Прямоугольник, который обозначает место в памяти, выделенное для объекта комбинированного типа, будем разбивать на меньшие прямоугольники по числу полей записи.

Пример. Рассмотрим программу

```
program EG(output);
 type pint = ↑integer;
 var i : integer; p, q : pint;
 begin new(p); new(q); i := 5;
   p↑ := 7; q↑ := p↑ − i; i := i + 1;
   p↑ := q↑ + i; q := p; write(q↑)
 end.
```

Ситуация перед выполнением первого оператора этой программы изображена на рис. 17, *а*. На рис. 17, *б* показана ситуация, возникающая после выполнения операторов *new*(*p*); *new*(*q*); $i := 5$ — переменные *i*, *p*, *q* получили значения. В результате выполнения операторов $p\uparrow := 7$; $q\uparrow := p\uparrow - i$; $i := i + 1$ (рис. 17, *в*) получили некоторые значения и переменные $p\uparrow$, $q\uparrow$, а также изменилось значение переменной *i*. Выполнение операторов $p\uparrow := q\uparrow + i$; $q := p$ приводит к изменению значения переменной $p\uparrow$ и к изменению значения ссылочной переменной *q* (рис. 17, *г*). Теперь значения переменных *p* и

q совпадают (выполнено отношение $p = q$), т. е. значениями обеих этих переменных является ссылка на одно и то же место в памяти. Поэтому в результате выполнения *write*$(q\uparrow)$ будет выведено целое число 8. Заметим, что место в памяти, ссылка на которое прежде являлась значением переменной q, теперь стало недоступным: нельзя, например, воспользоваться записанным там значением, нельзя освободить это место с помощью *dispose*, поскольку на него нет ссылки. Это место в памяти

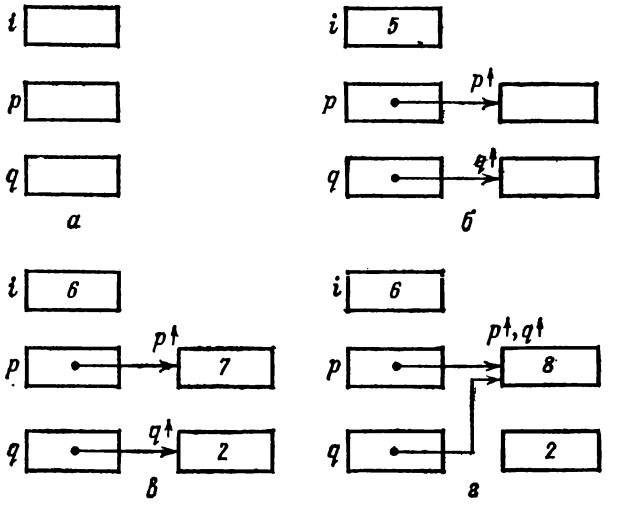


Рис. 17

представляет собой так называемый «мусор». От него следовало бы избавиться до изменения значения переменной q. Заменим три последних оператора в программе *EG* на следующую группу операторов: $p\uparrow := q\uparrow + i$; *dispose*$(q)$; $q := p$; *write*$(q\uparrow)$. Изображение ситуации, возникающей после выполнения *dispose*(q) и после выполнения $q := p$, представлено на рисунках 18, *а* и *б* соответственно.

Предостережение. Если в ситуации, изображенной на рис. 18, *б*, выполнить *dispose*(p), то место в памяти, ссылка на которое была значением переменных p и q, освободится. После этого не только значение переменной p, но и значение переменной q будет неопределенным.

Рассмотрим определения типов

```
type d = ↑t;
     t = record
           . . .
           c : d;
           . . .
         end;
```

В этом определении мы впервые сталкиваемся с ситуацией, когда в правой части определения некоторого типа

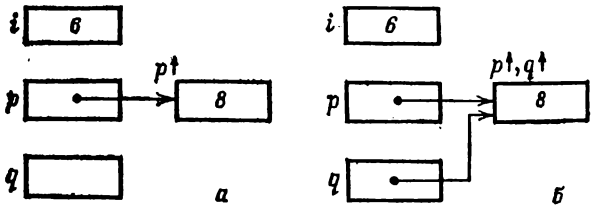


Рис. 18

встречается имя еще не определенного типа. Даже если поменять местами определения типов *d* и *t*, эта ситуация не изменится: в определении комбинированного типа *t* имеется поле *c* типа *d*. В паскале разрешается такое рекурсивное определение типов, если один из типов является ссылочным. Объектами типа *t* являются записи, одно из полей которых есть или **nil**, или ссылка на конкретное место в памяти, отведенное для объекта типа *t*.

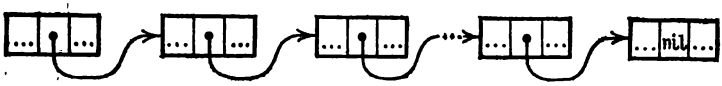


Рис. 19

Совокупность объектов типа *t*, упорядоченных с помощью ссылок так, как показано на рис. 19, называется *списком*. Объекты типа *t* из этой совокупности называются *элементами списка*.

Будем употреблять слова «ссылка на элемент списка» вместо слов «ссылка на место в памяти, отведенное для элемента списка». Каждый элемент списка, кроме последнего, содержит ссылку на следующий за ним эле-

мент. Признаком того, что данный элемент является последним в списке, служит то, что поле типа *d* этого элемента равно **nil**. На каждый элемент списка, кроме первого, имеется ссылка только от одного элемента — предшествующего. Вместе с каждым списком рассматривается переменная, значением которой является ссылка на первый элемент списка. Если список не имеет ни одного элемента (такой список называется *пустым*), значение этой переменной должно равняться **nil**.

Совокупность тех полей элемента списка, которые имеют тип, отличный от *d*, будем называть *информационной частью* элемента списка. Рассмотрим методы работы со списками, информационная часть элементов которых состоит из одного поля типа *integer*. Такие списки естественно называть списками целых чисел. Для того чтобы иметь возможность строить список, нужно включить в программу определения типов

```
type ссылка = ↑элем;
     элем = record
              инф : integer;
              след : ссылка
            end;
```

и описание **var** *p*, *q* : *ссылка*. Построим список из трех элементов, содержащих числа 5, 12 и —8. Значением переменной *p* в процессе построения всегда будет ссылка на первый элемент уже построенной части списка. Переменная *q* будет использоваться для выделения с помощью *new* места в памяти под размещение новых элементов списка. Выполнение оператора *p* := **nil** приводит к созданию пустого списка. После выполнения операторов

new (*q*); *q*↑. *инф* := — 8; *q*↑. *след* := *p*; *p* := *q*

имеем список, состоящий из одного элемента, содержащего число —8 в информационной части. Ссылка на этот элемент является значением переменных *p* и *q* (см. рис. 20, *а*). Выполнение операторов

new (*q*); *q*↑. *инф* := 12; *q*↑. *след* := *p*; *p* := *q*

приводит к тому, что в начало этого списка добавляется новый элемент, содержащий число 12; значением пере-

менных p и q является ссылка на первый элемент списка (см. рис. 20, *б*). После выполнения операторов

new (q); $q\uparrow$.*инф* := 5; $q\uparrow$.*след* := p; $p := q$

добавляющих в начало списка элемент, содержащий число 5, построение списка завершается (см. рис. 20, *в*). Значением переменных p и q является ссылка на первый

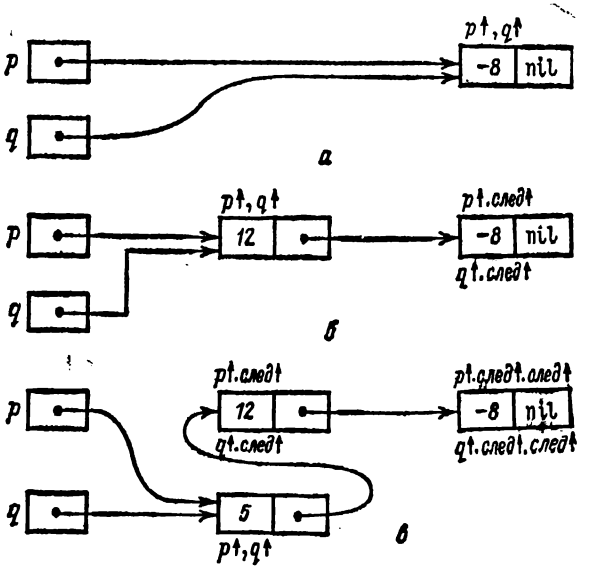


Рис. 20

элемент списка. Значение nil поля *след* элемента, содержащего число —8, является признаком того, что этот элемент — последний в списке. Значением переменной $p\uparrow$.*след*$\uparrow$.*инф* является целое число 12; значением $p\uparrow$.*след*$\uparrow$.*след* — ссылка на третий элемент списка.

Рассмотренный способ построения списка заключается в создании пустого списка и в повторяющемся выполнении ряда однотипных шагов, добавляющих в начало построенного списка новые элементы. Это означает, что в общем случае для построения списка можно использовать оператор цикла. Пусть, кроме переменных p и q типа *ссылка*, описаны переменные i и n типа *integer* и требуется построить список, элементы которого содержат квадраты чисел 1, 2, ..., n. Для этого нужно выполнить последовательность операторов

```
p := nil;
for i := n downto 1 do
  begin
    new(q); q↑.инф := sqr(i);
    q↑.след := p; p := q
  end
```

Отметим, что при $n<1$ тело цикла **for** $i := n$ **downto 1 do**..., не выполняется ни разу. Получающийся в этом случае результат ($p =$ **nil**) является тем не менее вполне осмысленным — построен пустой список.

Пусть построен список целых чисел, и значением переменной p является ссылка на первый элемент этого списка. Для того, чтобы вывести числа, содержащиеся в элементах списка, нужно выполнить последовательность операторов

```
q := p;
while q ≠ nil do
  begin
    write(q↑.инф);
    q := q↑.след
  end
```

По ходу выполнения цикла **while** $q \neq$ **nil do**... значением переменной q является поочередно ссылка на первый, второй, третий и т. д. элементы списка и, наконец, **nil**. Если список был пустым (было выполнено отношение $p =$ **nil**), то тело цикла не выполнится ни разу и не будет выведено ни одного числа.

Пример. Во входном файле находится конечная последовательность, содержащая нечетное количество целых чисел. Напишем программу, в результате выполнения которой выводится число, занимающее в данной последовательности центральную позицию. Поскольку заранее длина данной последовательности неизвестна, использовать для хранения элементов последовательности массив мы не можем. Использование файлов существенно увеличивает время выполнения программы, так как магнитофоны работают медленнее, чем вычислительная машина. Поэтому мы остановимся на реше-

нии, при котором в ходе выполнения программы будет строиться список целых чисел. Программа:

```
program центр (input, output);
 type ссылка = ↑элем;
        элем = record
                 инф : integer;
                 след : ссылка
               end;
 var p, q : ссылка; i, j : integer;
 begin p := nil; j := 0;
   while not eof (input) do
     begin new (q); read (q↑.инф);
       q↑.след := p; p := q; j := j + 1
     end;
   for i := 1 to j div 2 do q := q↑.след;
   write (q↑.инф)
 end.
```

Пояснение. Вначале выполняется построение списка, элементы которого содержат числа из данной последовательности, и, одновременно, подсчет количества этих чисел. Значением переменной q после выполнения цикла **for** $i := 1$ **to** j **div** 2 **do** ... является ссылка на то место в памяти, где расположен элемент, занимающий центральную позицию в списке. Заметим, что в построенном списке данные числа следуют в обратном порядке.

ЗАДАЧИ

1. Во входном файле расположена последовательность целых чисел. Определить, имеется ли среди этих чисел два совпадающих.

2. Написать программу вычисления коэффициентов многочлена $c_0 + c_1x + \ldots + c_{n+m}x^{n+m}$, являющегося произведением многочленов $a_0 + a_1x + \ldots + a_nx^n$ и $b_0 + b_1x + \ldots + b_mx^m$. Считать, что во входном файле расположена последовательность целых чисел n, a_0, a_1, ..., a_n, m, b_0, b_1, ..., b_m.

3. Во входном файле расположены два натуральных числа n и m $(n \leq m)$. Найти непериодическую и перио-

дическую части представления рационального числа n/m в виде бесконечной десятичной дроби.

4. Во входном файле расположена последовательность, содержащая четное число символов. Написать программы, позволяющие выяснить:

а) совпадают ли первая и вторая половины последовательности,

б) является ли последовательность палиндромом (см. задачу 3б § 1 гл. III).

Указание. Задачу решать с помощью построения списка, информационная часть элементов которого содержит одно поле типа *char*.

5. Во входном файле расположена последовательность целых чисел, содержащая по крайней мере два отрицательных числа. Написать программу, в ходе выполнения которой выводятся числа этой последовательности, начиная с первого отрицательного и кончая последним отрицательным.

Указание. Для решения задачи требуется построить список, в который данные числа входят в том же порядке, в каком они расположены во входном файле. Разобранный ранее метод построения списка для этой цели не подходит. Можно воспользоваться следующим методом. Начать построение с создания списка, состоящего из одного элемента, а новые элементы пристраивать не в начало уже построенной части списка, а в хвост. При таком способе построения потребуется дополнительная ссылочная переменная, которая всегда должна содержать ссылку на последний элемент построенной части списка.

§ 3. Операции над списками

Основными операциями над списками являются:

— переход от элемента к следующему элементу;

— включение нового элемента в список;

— исключение элемента из списка.

Пусть значением переменной q типа *ссылка* является ссылка на некоторый элемент списка целых чисел. Тогда после присваивания $q := q\uparrow.след$ ее значением будет или ссылка на следующий элемент этого списка (если такой элемент имеется) или **nil**. Пользуясь таким способом перехода от одного элемента к другому, можно просматривать весь список или его часть.

Пусть теперь имеются переменные p, q, r типа *ссылка* и значением переменной q является ссылка на некоторый элемент списка целых чисел, а значением p — ссылка на первый элемент этого списка. И пусть требуется включить в данный список после элемента, ссылка на который является значением переменной q, новый

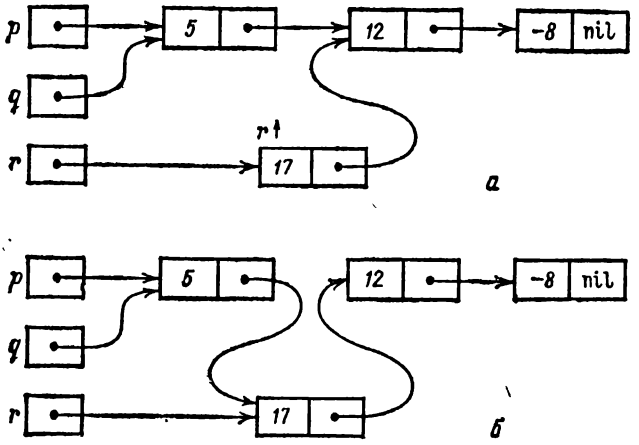


Рис. 21

элемент, содержащий число 17. Для этого нужно выполнить последовательность операторов

new(*r*); $r\uparrow.$*инф* := 17; $r\uparrow.$*след* := $q\uparrow.$*след*; $q\uparrow.$*след* := r

Для списка, изображенного на рис. 20, *в*, результат такого включения показан на рис. 21 (*а* — ситуация перед выполнением присваивания $q\uparrow.$*след* := r, *б* — после).

Пусть теперь значением переменной q является ссылка на некоторый (не последний) элемент списка целых чисел и требуется исключить из списка элемент, следующий за тем, ссылка на который является значением переменной q. Для этого нужно выполнить последовательность операторов

$r := q\uparrow.$*след*; $q\uparrow.$*след* := $q\uparrow.$*след*$\uparrow.$*след*; $r\uparrow$*след* := **nil**

Второе из этих присваиваний — это собственно исключение элемента из списка, а первое выполняется для того, чтобы сохранить ссылку на исключенный элемент, т. е. чтобы после исключения из списка он оставался доступным и с ним можно было бы выполнять некоторые

действия (например, вставить его на другое место в списке или освободить занимаемую им память с помощью *dispose*). Третье присваивание выполняется для того, чтобы сделать исключение окончательным, т. е. чтобы из исключенного элемента по ссылке нельзя было попасть в список, из которого этот элемент исключен.

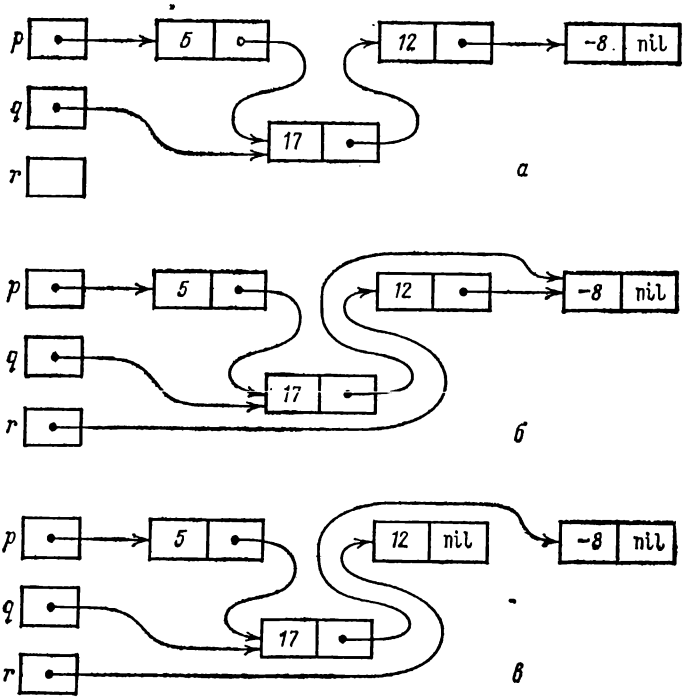


Рис. 22

На рис. 22 показан пример исключения из списка (*а* до исключения, *б* — перед выполнением присваивания $r\uparrow.след := \mathbf{nil}$, *в* — после). Таким образом, операции включения и исключения элементов представляют собой изменение значений полей ссылочного типа некоторых элементов списка и выполняются довольно просто. Однако рассмотренные методы включения и исключения не подходят для включения нового элемента в начало списка и для исключения первого элемента списка, так как у первого элемента нет предшествующего. Пусть по-прежнему значением переменной *p* типа *ссылка*

является ссылка на первый элемент списка целых чисел. Для того чтобы включить в начало списка новый элемент, содержащий число 17, нужно выполнить действия:

new(*r*); *r*↑.*инф* := 17; *r*↑.*след* := *p*; *p* := *r*

Для списка, изображенного на рис. 20, *в*, результат

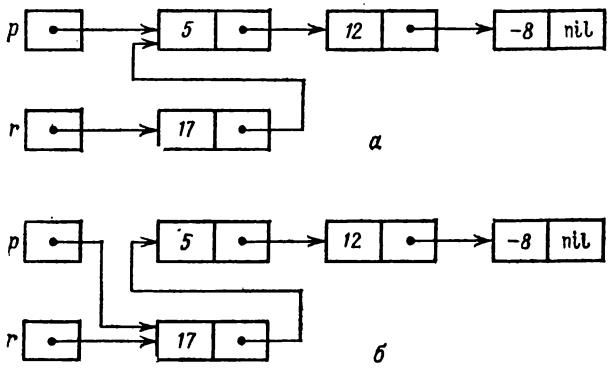


Рис. 23

такого включения показан на рис. 23 (*а* — перед выполнением присваивания *p* := *r*, *б* — после).

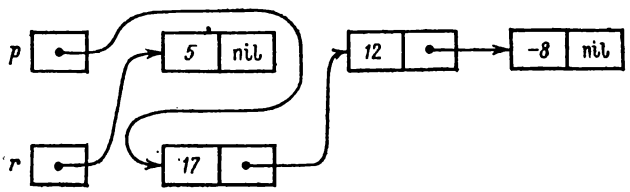


Рис. 24

Для исключения первого элемента из списка нужно выполнить:

r := *p*: *p* := *p*↑.*след*; *r*↑.*след* := **nil**

Результат применения этих действий к списку, изображенному на рис. 22, *а*, показан на рис. 24.

То, что включение (или исключение) в зависимости от местоположения элемента в списке выполняется поразному, доставляет неудобства при программировании. В программах приходится предусматривать дополнительные проверки для того, чтобы выполнить включение или исключение одним из двух способов. Чтобы сделать действия, выполняемые при включении элемента в список (исключении элемента из списка), единообразными, обычно применяется следующий прием. В начало каждого списка добавляется *заглавный элемент*. Он никогда не исключается из списка и перед ним в список никогда не включаются новые элементы. Информационная часть заглавного элемента или не используется вообще, или используется для специальных целей. Например, в случае списка целых чисел она может содержать число, равное количеству элементов списка. Добавление заглавного элемента в список приводит к тому, что теперь у всех элементов, в том числе и у первого элемента, имеется предшественник, и действия по включению новых элементов в список (или исключению элементов из списка) проводятся одним способом.

Пример. Во входном файле расположена непустая последовательность целых чисел $a_1, a_2, \ldots, a_n$. Определить n — количество этих чисел и вывести их в порядке возрастания. Для решения этой задачи будем строить список, элементы которого упорядочены по возрастанию содержащихся в них целых чисел. Построение выполняется за n шагов. Первый шаг — это создание списка, состоящего из одного элемента, который содержит число a_1. Очевидно, что такой список является упорядоченным. На i-м шаге ($i = 2, 3, \ldots, n$) переходим от упорядоченного списка, элементы которого содержат числа $a_1, \ldots, a_{i-1}$, к упорядоченному списку, элементы которого содержат числа $a_1, \ldots, a_{i-1}, a_i$. Для выполнения такого перехода достаточно включить в список новый элемент, содержащий число a_i. Его надо вставить непосредственно за последним по порядку элементом, содержащим число, меньшее чем a_i. Если же все элементы исходного списка содержат числа, не меньшие чем a_i, то новый элемент должен быть вставлен в начало списка. Для упрощения структуры программы будем строить упорядоченный список с заглавным элементом, поле *инф* которого используется для подсчета исходных чисел.

Программа:

```
program Lsort (input, output);
 label 1;
 type ссылка = ↑элем;
        элем = record
                 инф : inteqer;
                 след : ссылка
                 end;
 var  b, p, q, v : ссылка;
 begin new (p); read (p↑.инф); p↑.след := nil;
       new (b); b↑.след := p; b↑.инф := 1;
   while not eof (input) do
     begin new (v); read (v↑.инф);
       q := b; p := b↑.след; b↑.инф := b↑.инф + 1;
       while p ≠ nil do
         if p↑.инф < v↑.инф then
           begin q := p; p := p↑.след end
                                 else goto 1;
    1 : v↑.след := p; q↑.след := v
     end;
   writeln ('количество _ чисел _ равно _', b↑.инф);
   p := b↑.след;
   writeln ('список _ упорядоченных _ чисел :');
   while p ≠ nil do
     begin writeln (p↑.инф); p := p↑.след end
end.
```

Пояснение. Вначале создается список из заглавного элемента и элемнета, содержащего первое число из данной последовательности. На каждом шаге цикла **while not** *eof* (*input*) **do**... создается новый элемент, в его информационную часть вводится очередное число из данной последовательности и этот элемент помещается на свое место в строящемся списке. По ходу выполнения этого цикла подсчитывается количество обработанных чисел (соответствующее значение присваивается полю

инф заглавного элемента списка). Заключительные операторы программы предназначены для вывода полученных результатов.

Разобранную задачу нельзя решить с помощью построения упорядоченного массива целых чисел, поскольку заранее неизвестно, сколько чисел будет упорядочиваться. Решение же этой задачи с помощью формирования файла, компоненты которого — целые числа, расположенные в порядке возрастания, явно проигрывает разобранному решению в эффективности. При включении новой компоненты в файл (и исключении компоненты из файла) его почти всегда приходится переписывать целиком. Исключение составляет случай, когда новые компоненты дописываются в конец файла, открытого для записи.

Во всех рассмотренных примерах информационная часть элементов списка состояла из одного поля типа *integer*. В общем же случае она может содержать произвольное (фиксированное для данного типа элемента) количество полей разных типов. Ясно, что если значением переменной *p* является ссылка на элемент списка, то присоединяя к обозначению $p\uparrow$ через точку имя соответствующего поля, можно манипулировать с любым полем информационной части элемента списка.

ЗАДАЧИ

1. Во входном файле расположены целые числа. Вывести вначале все положительные числа, затем — все отрицательные

а) в любом порядке;

б) с сохранением порядка следования среди положительных и среди отрицательных чисел.

2. Во входном файле расположены два набора натуральных чисел, упорядоченных по возрастанию. Между ними стоит число 0. Вывести все данные натуральные числа в порядке возрастания.

Указание. Вначале построить по исходным данным два упорядоченных списка. Затем слить их в один упорядоченный список, изменяя только значения полей ссылочного типа элементов списков.

3. Компоненты файла *ассортимент* являются объектами типа *игрушка* (см. задачу 4 § 4 гл. III). Вывести названия игрушек, которые подходят детям 5 лет, в порядке возрастания цен.

Указание. Определить тип элементов списка с информационной частью типа *игрушка* и тип ссылок на эти элементы:

```
type ссылка = ↑элем;
     элем = record
              инф : игрушка;
              след : ссылка
            end;
```

Построить упорядоченный по возрастанию цен список тех игрушек из файла *ассортимент*, которые подходят детям 5 лет.

4. Во входном файле находится целое число n. Пользуясь построенным при решении задачи 3 списком, решить следующую задачу: определить, какое максимальное количество различных игрушек для детей 5 лет можно приобрести на сумму n копеек.

5. Во входном файле расположена последовательность целых чисел. Вывести те числа, которые встречаются в этой последовательности ровно один раз.

§ 4. Двоичные деревья

При решении многих задач математики полезным оказывается использование понятия *графа*. Граф — это набор точек на плоскости (эти точки называются *вершинами* графа), некоторые из которых соединены отрезками (эти отрезки называются *ребрами* графа). Вершины могут располагаться на плоскости произвольным образом. Причем неважно, является ли ребро, соединяющее две вершины, отрезком прямой или кривой линии. Важен сам факт соединения двух данных вершин ребром. Примером графа может служить схема линий метрополитена. Если требуется различать вершины графа, их нумеруют или обозначают разными буквами. В изображении графа на плоскости могут появляться точки пересечения ребер, не являющиеся вершинами. (Граф с вершинами A, B, C, D и ребрами, соединяющими A и B, B и C, C и D, D и A, A и C, B и D, может быть изображен так, как это сделано на рис. 25). Чтобы отличать вершины от таких точек, будем изображать вершины кружками. Заметим, что граф, показанный на рис. 25, можно изобразить и без пересечений ребер

(рис. 26). Однако существуют графы, которые невозможно изобразить на плоскости без пересечения ребер (примеры таких графов даны на рис. 27, *а* и *б*).

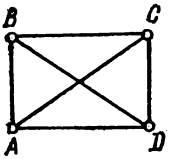


Рис. 25

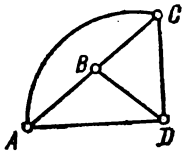
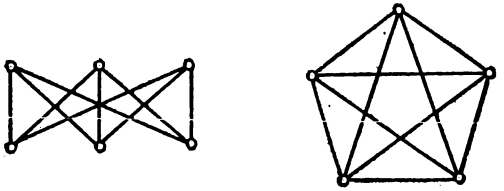


Рис. 26

. Говорят, что две вершины графа соединены *путем*, если из одной вершины можно пройти по ребрам в другую вершину. Путей между двумя данными вершинами может быть несколько. Поэтому обычно путь обозначают



а *б*

Рис. 27

перечислением вершин, которые посещаются при движении по ребрам. Например, вершины A и C в графе, изображенном на рис. 26, соединены путями AC, ABC, ADC и т. д. Граф называется *связным*, если любые две его вершины соединены некоторым путем. Все рассмотренные до сих пор графы являются связными. На рис. 28 приведен пример несвязного графа. Далее будут рассматриваться только связные графы.

Состоящий из различных ребер замкнутый путь называется *циклом*. В графе, изображенном на рис. 26, циклами являются, например, $ABCA$ и $BCDB$. Связный граф, в котором нет циклов, называется *деревом*. Пример дерева дан на рис. 29. Одним из основных отличительных свойств дерева является то, что в нем любые две вершины соединены единственным путем. Дерево называется *ориентированным*, если на каждом его ребре указано направление. Следовательно, о каждой вершине можно сказать, какие ребра в нее входят, какие

выходят. Точно так же о каждом ребре можно сказать, из какой вершины оно выходит и в какую входит. *Двоичное* дерево — это такое ориентированное дерево, в котором

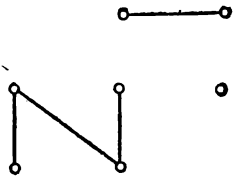


Рис. 28

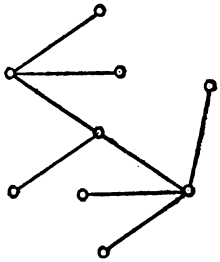


Рис. 29

1) имеется ровно одна вершина, в которую не входит ни одного ребра (эта вершина называется *корнем* двоичного дерева);

2) в каждую вершину, кроме корня, входит одно ребро;

3) из каждой вершины (включая корень) исходит не более двух ребер.

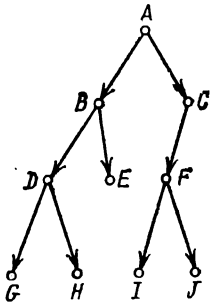
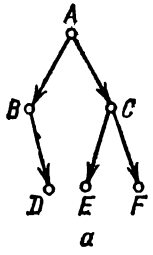


Рис. 30



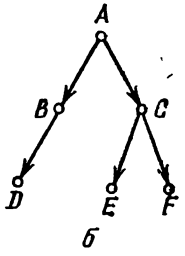


Рис. 31

Пример двоичного дерева дан на рис. 30. Вершина A — корень этого дерева. Для удобства дальнейших рассмотрений зафиксируем способ изображения двоичных деревьев. Корень будет располагаться выше других вершин (как вершина A на рис. 30). Для ребер, выходящих из любой вершины, имеется две возможности — быть направленными влево вниз и вправо вниз. Причем если реализована одна из этих возможностей (из вершины

выходит одно ребро), то указание направления ребра влево вниз или вправо вниз является существенным (деревья, изображенные на рис. 31, *а* и *б*, различны). При таких предположениях нет необходимости указывать направление на ребрах дерева: все ребра ориентированы вниз. Далее будут рассматриваться только двоичные деревья.

При решении многих прикладных задач бывает удобно представлять наборы объектов в виде двоичных деревьев. Рассмотрим задачу кодирования непустой последовательности целых чисел. Пусть дана последовательность целых чисел $a_1, a_2, \ldots, a_n$ и функция целочисленного аргумента $F(x)$, принимающая целые значения. Значение $F(x)$ будем называть кодом числа x. Пусть требуется закодировать данные числа, т. е. вычислить значения $F(a_1), F(a_2), \ldots, F(a_n)$, и пусть в последовательности $a_1, a_2, \ldots, a_n$ имеются частые повторения значений элементов. Если способ вычисления $F(x)$ достаточно сложен, то нужно избежать повторных вычислений одного и того же значения. Для этого следует по ходу кодирования данных чисел строить таблицу (справочник) уже найденных кодов. Организация этой таблицы должна быть такой, чтобы, во-первых, в ней довольно быстро можно было находить элемент с данным значением (или устанавливать факт отсутствия такого элемента в таблице) и, во-вторых, без особых усилий добавлять в таблицу новые элементы. Этим требованиям в наибольшей мере удовлетворяет представление таблицы в виде двоичного дерева, в вершинах которого расположены различные элементы из данной последовательности и их коды. Процесс построения двоичного дерева идет так: первое число и его код образуют корень дерева. Дерево пока состоит только из корня. Следующее число, которое нужно кодировать, сравнивается с числом в корне, и если они равны, то дерево не разрастается и код может быть взят из корня. Если новое число меньше первого, то проводится ребро из корня влево вниз, иначе — вправо вниз. В образовавшуюся вершину помещаются это новое число и его код.

Пусть уже построено некоторое дерево и имеется число x, которое нужно закодировать. Сначала сравниваем с x число, записанное в корне дерева. В случае равенства поиск закончен и код числа x извлекается из корня. В противном случае переходим к рассмотрению вершины, которая находится слева внизу, если число x

меньше только что рассмотренного, или справа внизу, если число x больше только что рассмотренного. Сравниваем число x с числом, содержащимся в этой вершине, и т. д. Процесс завершается в одном из двух случаев:

1) найдена вершина, содержащая число, равное x;

2) в дереве отсутствует вершина, к которой нужно перейти для выполнения очередного шага поиска.

В первом случае из найденной вершины извлекается код числа x. Во втором — выясняется, что число x еще не кодировалось, т. е. необходимо вычислить $F(x)$ и присоединить к дереву вершину, в которой расположены x, $F(x)$.

Пусть последовательность, которую нужно закодировать, начинается с чисел 8, 4, 13, 10, 14, 10. Тогда вначале дерево будет разрастаться так, как показано на

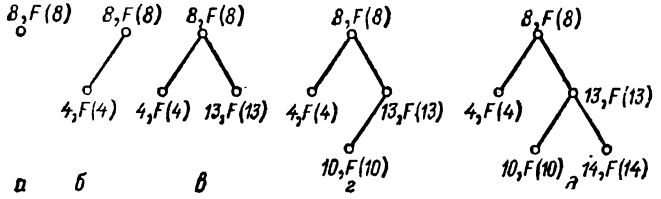


Рис. 32

рис. 32. При появлении числа 10 в качестве 6-го элемента последовательности к дереву не добавляется новой вершины, а значение $F(10)$ извлекается из имеющейся вершины.

В программе построения последовательности кодов можно определить следующие типы:

```
type ребро = ↑верш;
     верш = record
              лев, прав : ребро;
              число, код : integer
            end;
```

Объектами типа *верш* являются записи, в которых каждое из полей *лев* и *прав* есть либо **nil**, либо ссылка на конкретное место в памяти, отведенное с помощью *new* для объекта типа *верш*. Дерево в программе на паскале можно представлять в виде множества объектов типа *верш*, связанных ссылками. Сами эти объекты будут

вершинами дерева, а ссылки на места в памяти, отведенные для объектов типа *верш*, — ребрами дерева. Если при этом поле *лев* (*прав*) некоторого объекта типа *верш* есть **nil**, то это значит, что в дереве из данной вершины не исходит ребро, направленное влево вниз (вправо вниз). На рис. 33 показано представление дерева, изображенного на рис. 32, *д*, в памяти вычислительной машины.

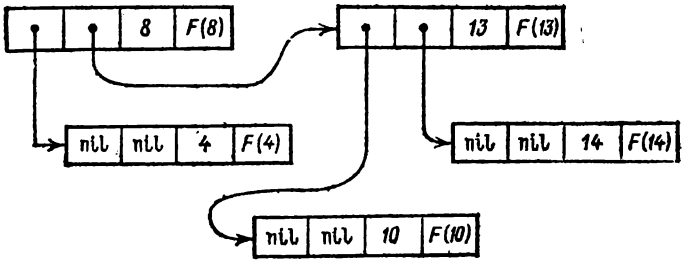


Рис. 33

Пусть значением переменной *v* типа *ребро* является ссылка на место в памяти, отведенное для объекта типа *верш* (т. е. на некоторую вершину дерева). Тогда выполнение присваивания $v := v\uparrow.лев$ ($v := v\uparrow.прав$) означает переход к вершине, расположенной непосредственно слева снизу (справа снизу) от данной (если, конечно, соответствующее поле данной вершины не есть **nil**). Ясно, что таким способом можно передвигаться по дереву от вершины к вершине сверху вниз. Включение новой вершины в дерево представляет собой (как и включение нового элемента в список) изменение значений полей ссылочного типа некоторых вершин данного дерева.

Вместе с каждым деревом рассматривается переменная, значением которой является ссылка на корень дерева. Если в дерево не входит ни одной вершины, значение этой переменной должно равняться **nil**.

Напишем программу, по ходу выполнения которой кодируется последовательность натуральных чисел, расположенных во входном файле. Для кодирования используется файл *коды*, компонентами которого являются целые числа. Код $F(k)$ числа k — это k-я по порядку компонента файла *коды*.

Программа:

```
program кодировка (input, output, коды);
 label 1, 2;
 type ребро = ↑верш;
      верш = record
                лев, прав : ребро;
                число, код : integer
             end;
      фк = file of integer;
 var  t, u, v : ребро; x : integer;
      коды : фк;
 function F (x : integer) : integer;
   var i, j : integer;
   begin reset (коды);
     for i := 1 to x do read (коды, j);
     F := j
   end;
 begin new (u); read (u↑.число);
   u↑.код := F (u↑.число); u↑.лев := nil;
   u↑.прав := nil; writeln (u↑.код);
   while not eof (input) do
     begin read (x); t := u;
   1 : if t↑.число = x then writeln (t↑.код)
       else
        begin
          if t↑.число > x then
            begin if t↑.лев ≠ nil then
             begin t := t↑.лев; goto 1 end
                  else begin new (v);
                             t↑.лев := v;
                             goto 2
                       end
             end
```

```
        else if t↑.прав ≠ nil then
                begin t := t↑.прав; goto 1 end
                else begin new(v);
                        t↑.прав := v
                     end;
    2 : v↑.число := x; v↑.код := F(x);
        v↑.лев := nil; v↑.прав := nil;
        writeln(v↑.код)
    end
  end
end.
```

Пояснение. Вначале кодируется первое число и строится дерево, состоящее из одной вершины. Затем в цикле **while not** *eof*(*input*) **do**... кодируются остальные числа. Переменная *u* по ходу выполнения этого цикла содержит ссылку на корень дерева. Переменная *t* используется для просмотра вершин дерева в нужном порядке, переменная *v* — для отведения места в памяти, в котором размещается новая вершина, и для присоединения новой вершины к дереву.

Разобранную задачу можно было решать с помощью построения списка, элементы которого содержат числа и их коды и упорядочены по возрастанию содержащихся в них чисел. При этом для поиска элемента списка, содержащего очередное число *x*, которое нужно закодировать, пришлось бы просматривать в среднем половину элементов списка и сравнивать содержащиеся в них числа с *x* *). В решении, использующем двоичное дерево, число сравнений при поиске не может превзойти числа вершин, входящих в самый длинный путь из корня в какую-нибудь концевую вершину. Если *m* закодированных

*) Мы не можем эффективно использовать метод поиска делением пополам (см., например, С. А. Абрамов. Поиск в упорядоченной совокупности и упорядочивание//Квант. — 1986, № 1, С. 46—48), потому что, работая со списком, не можем быстро добираться до его середины. Поиск делением пополам удобно применять к массиву, но в рассматриваемой задаче нельзя использовать массив, поскольку заранее неизвестно, сколько чисел будет кодироваться.

чисел $a_1, \ldots, a_m$ расположены так, что $a_1 > a_2 > \ldots > a_m$, то все дерево вытянется в одну линию (рис. 34) и будет похожим на список. Среднее число сравнений при поиске вершины, содержащей число x, в этом случае такое же, как и при поиске в списке. Однако если данные числа были хорошо перемешаны, то строящееся дерево не будет однобоким, а будет похоже на дерево с рис. 35. В случае,когда число вершин $m = 2^s - 1$, дерево, похожее на изображенное на рис. 35

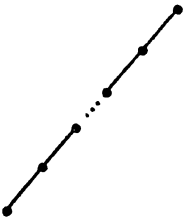


Рис. 34

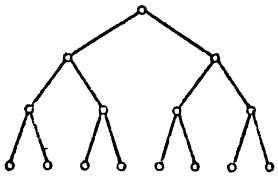


Рис. 35

(«полное двоичное дерево»), будет обладать тем свойством, что в каждый путь от корня до концевой вершины входит s вершин, т. е. $\log_2(m+1)$ вершин. В других случаях тоже можно считать, что среднее число вершин в пути от корня до концевой вершины примерно равно величине $\log_2(m+1)$. Таким образом, в случае хорошей перемешанности (неупорядоченности) данных чисел a_1, $a_2, \ldots, a_n$ представление таблицы кодов в виде дерева имеет явные преимущества перед представлением таблицы кодов в виде списка.

ЗАДАЧИ

1. Выписать все циклы графа, изображенного на рис. 26.

2. Построить дерево, соответствующее кодированию конечной последовательности 8, 12, 7, 8, 17, 6, 4, 10, 8, 4, 6, 7, 20, 5, 16, 9, 5, 4.

3. Доказать, что в любом дереве число вершин на единицу больше числа ребер.

4. Компонентами файла *чис* являются целые числа. Во входном файле расположена последовательность целых чисел. Вывести те числа из файла *чис*, которые не входят в данную последовательность.

Указание. Для быстрой проверки факта вхождения числа из файла *чис* в данную последовательность построить дерево, содержащее все различные числа из входного файла.

5. Во входном файле расположена последовательность целых чисел. Выяснить, есть ли в этой последовательности совпадающие числа. Решить задачу с помощью дерева.

Сергей Александрович Абрамов,
Евгений Викторович Зима

НАЧАЛА ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПАСКАЛЬ

Редактор *О. И. Сухова*
Художественный редактор *Т. Н. Кольченко*
Технический редактор *Е. В. Морозова*
Корректор *О. А. Бутусова*

ИБ № 32427

Сдано в набор 21.07.86. Подписано к печати 20.01.87. Т-05221. Формат 84×108/32. Бумага тип. № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 5,88. Усл. кр.-отт. 6,2. Уч.-изд. л. 5,63. Тираж 300 000 экз. Заказ № 245. Цена 20 коп.

Ордена Трудового Красного Знамени издательство «Наука»
Главная редакция физико-математической литературы
117071, Москва В-71, Ленинский проспект, 15

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29.