

**БИБЛИОТЕЧКА
ПРОГРАММИСТА**

А. Л. БРУДНО

Алгол



БИБЛИОТЕЧКА
ПРОГРАММИСТА

А. Л. БРУДНО

АЛГОЛ

ИЗДАНИЕ ВТОРОЕ, ИСПРАВЛЕННОЕ



ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
МОСКВА 1971

В книге дается полное изложение алгоритмического языка «алгол-60» который отличается удобством, строгостью и обзорностью записанных на нем алгоритмов и программ. Автор переписал «Официальное сообщение об алголе» в форме, удобной для изучения, и составил синтаксические таблицы для справок.

Знание алгола необходимо всем, кто занимается вычислительной математикой и программированием. Во втором издании иероглифы алгола написаны по-английски.

Александр Львович Брудно

АЛГОЛ

(Серия: «Библиотечка программиста»)

М., 1971 г., 80 стр.

Редактор *М. К. Антонова*

Техн. редактор *К. Ф. Брудно* Корректоры *Т. С. Плетнева, В. П. Сорокина*

Сдано в набор 30/XII 1970 г. Подписано к печати 23/IV 1971 г. Бумага 84×108/32
Физ. печ. л. 2,5 Условн. печ. л. 4,2 Уч.-изд. л. 4,01

Тираж 60 000 экз Т-02448. Цена книги 24 коп. Заказ 1759

Издательство «Наука»

Главная редакция физико-математической литературы

Москва В-71, Ленинский проспект, 15

2-я типография издательства «Наука», Москва, Шубинский пер., 10,

ПРЕДИСЛОВИЕ

0.1. Алгол отличается удобством, элементарностью, а главное обозримостью вычислительных программ и алгоритмов. Другое выигрышное свойство алгола — в его естественности: не много нужно выучить, чтобы писать и читать на алголе. Наконец, фундаментальные идеи алгола, благодаря смелости решений, стали основополагающими для алгоритмических языков. Поэтому математику, в первую очередь программисту, нужно знать алгол, независимо от собственных вкусов.

0.2. Авторы алгола опубликовали свою работу в «Официальном сообщении» *). Оно содержит целые страницы текста, который приобретает смысл только в конце. Это годится для транслятора, но игнорирует специфику человека, ибо человеку трудно заучивать бессмысленный текст. Есть и обратные примеры: правила расстановки скобок мы знаем из школы, а для транслятора их приходится определять.

В этой книге я попытался переписать «Официальное сообщение» в форме, рассчитанной на человека, а не на транслятор, и составил синтаксические таблицы для справок.

0.3. Существуют, разумеется, программы, которые на алголе написать затруднительно. Происходит это не только потому, что алгол не рассматривает содержимое ячейки как информацию, записанную последовательностью нулей и единиц. Важнее здесь распределение памяти и взаимодействие подпрограмм. В алголе легко сказать «запомни», но сложно говорить «забуди»: освободить ячейки можно только в порядке, обратном тому, в котором они занимались. Сложно в алголе и осуществить засылку передачи управления. Но алгол удобен для

*) Список литературы см. на стр. 4.

записи сложных алгоритмов и эффективен для решения вычислительных задач, не слишком трудных для данной машины.

0.4. Алгол демонстративно игнорирует специфику отдельных цифровых электронных машин (ЦЭМ). Алгольная программа вводится в любую ЦЭМ, и *транслятор* — программа, построенная для данной ЦЭМ, — заставляет ЦЭМ, в конечном счете, выполнить алгольную программу.

Все же современные трансляторы обладают такими особенностями, что для практической работы нужно посмотреть руководство по данному транслятору. Эта книга не может заменить его.

0.5. Кроме «Официального сообщения» и книг П. Наура, Мак-Кракена, Боттенбруха, Лаврова и Агеева, из которых я заимствовал все, что нашел кратким и ясным, я пользовался еще и толстыми руководствами, по которым видел, как не нужно писать. Перечислять их авторов было бы черной неблагодарностью.

Литература

1. Алгоритмический язык алгол-60, под ред. П. Наура, Русский перевод под ред. А. П. Ершова и др., «Мир», 1965.
2. Н а у р П., Курс программирования алгол-60, Русский перевод № 265 85, Бюро ВИНТИ.
3. Л а в р о в С. С., Универсальный язык программирования (Алгол-60), изд. 2, «Наука», 1967.
4. Сообщение о подмножестве алгола-60, Сборник «Современное программирование», «Советское радио», 1966.
5. К н у т Д. Э., Список неясных мест Алгола-60, Журнал вычислительной математики, т. 7, № 1, 1967.
6. А г е е в М. И., А л и к В. П., Г а л и с Р. М., Алгоритмы (1—50), (51—100) и (101—150), ВЦ АН СССР, 1966 и 1967.
7. Б а л у е в А. Н. и др., Сборник упражнений алгол-60, Изд-во ЛГУ, 1967.
8. М а к - К р а к е н Дж., Программирование на алголе, «Мир», 1964.
9. Б о т т е н б р у х Г., Структура алгол-60 и его использование, ИЛ, 1963.
10. А г е е в М. И., Основы алгоритмического языка алгол-60, ВЦ АН СССР, 1964.

ГЛАВА 1

ЭЛЕМЕНТАРНАЯ ЧАСТЬ АЛГОЛА

§ 1. Знаки и слова алгола

В алголе имеется *твердый перечень* знаков, которыми записывается программа. Каждому знаку должна соответствовать клавиша вводного устройства *) (а их хочется иметь мало), и каждый знак должен восприниматься машиной и расшифровываться транслятором. Полный список знаков алгола приводится в табл. 4 на стр. 79.

При построении фраз одни знаки алгола играют роль букв нашего алфавита, другие — роль иероглифов, обозначающих целые понятия, наконец, роль третьих (знаки арифметических действий, скобки и т. д.) нам представляется промежуточной. Собственно говоря, новыми для нас будут только иероглифы, но их всего 20, и познакомимся мы с ними постепенно. Сперва займемся знаками, отнесенными к алфавиту.

1.1. А л ф а в и т алгола содержит:

цифры от 0 до 9;

два булевских числа `true` и `false` **);

большие и малые буквы латинского алфавита.

Кроме того, в «Официальном сообщении» было предусмотрено, что, например, русский вариант алгола может содержать русские буквы.

Таким образом, *греческих и готических букв в алголе нет. Все символы алгола пишутся в строку. Нельзя их подчеркивать или надчеркивать. Нельзя обычным образом писать степени и ставить индексы снизу или сверху.* Пробел или переход на новую строку в алголе не принимается во внимание; их можно свободно использовать, чтобы облегчить чтение программы.

*) Или последовательность клавиш, что несколько облегчает дело.

**) Числа `true` и `false` являются самостоятельными знаками алгола, не имеющими отношения к буквам, которыми они написаны. То же будет относиться и к иероглифам. Все они, во избежание путаницы, печатаются полужирным шрифтом, а при письме — подчеркиваются.

Из знаков алфавита составляются **ч и с л а и и м е н а**, играющие роль слов алгола.

1.2. Ч и с л а в алголе бывают булевские **true** и **false** и арифметические. Арифметические числа составляются из цифр.

Арифметические числа бывают двух типов: *целые и реальные* *). Основное различие между ними в том, что *целые* числа будут изображаться (и, по возможности, преобразовываться) в машине *точно*, а *реальные* — лишь *приближенно*. Таким образом, введя в машину *реальное* число *три*, мы обязаны считаться с тем, что фактически ввели число, быть может, чуть меньшее или большее тройки.

1.2.1. Ц е л ы е числа изображаются, как обычно, строкой цифр. Впереди ставят знак **+** или **-** (**+** не обязателен). Запрещается ставить запятые или точки (например, для отделения разрядов).

Примеры записей <i>целых</i> чисел	
верно	неверно
0	17.38
6	14.0
+ 400	14,0
- 1234	14.
01000	1.000
7000000	123456789000

Последнее число может оказаться недопустимо велико; многие трансляторы требуют, чтобы целые числа имели меньше десяти знаков.

1.2.2. Реальное число записывается в одной из трех форм:

1) α 2) знак $_{10}\beta$ 3) $\gamma_{10}\beta$,
которые понимаются соответственно как величины
 α ; $\pm 10^\beta$; $\gamma \cdot 10^\beta$.

Здесь α — (десятичная) дробь, β — (обязательно) целое, γ — целое или дробь, а *опущенная десятка* « $_{10}$ » — самостоятельный значок алгола.

Дробь — это знак (+ не обязателен) и последовательность цифр, где перед цифрами или между ними (но не после них!) стоит точка, отделяющая целую часть

*) По традиции в Москве их иногда называют действительными, а в Ленинграде — вещественными.

от дробной. Таким образом, в алголе: (1) целая часть отделяется от дробной точкой, а не запятой — преимущество этого может оценить всякий, кому случалось писать перечень десятичных дробей; (2) дробь может не иметь целой части, но должна содержать дробную (хотя бы из нулей). Приведем примеры записи дробей:

— 200.53	+ 00.53	200.53	07.0
— 0.2	+ 0.2	.2	— .2

Целое (напоминаем) — это знак (+ не обязателен) и последовательность цифр.

В форме 2) знак + перед всем числом не обязателен. Приведем примеры записей в форме 2):

10^3	$+_{10}3$	$10+3$	$+_{10}+3$
$-_{10}-2$	$-_{10}2$	$10-2$	

Приведем примеры записей в форме 3):

— .083₁₀—02 +07.43₁₀ 3 9.0₁₀—2 2₁₀2

Таким образом, в алголе целое число не является частным случаем реального, а числа 0.0 и 3.14₁₀2 реальные, но не целые.

К о н т р о л ь н о е у п р а ж н е н и е. Половина нижеследующих примеров представляет запись *реальных* чисел, а половина — не представляет. Какие записи верны?

- | | | |
|---------------------|------------------|------------------|
| 1) —.008 | 5) $\cdot_{10}3$ | 9) 13.411 732 |
| 2) $+13.47_{10}+18$ | 6) $2_{10}3.0$ | 10) $2.48_{10}n$ |
| 3) $4 \times_{10}2$ | 7) $-88_{10}-7$ | 11) $0.0_{10}1$ |
| 4) (16.20) | 8) $1.24_{10}3$ | 12) $12_{10}8$ |

О т в е т. Верны записи 1, 2, 7, 8, 9, 11.

У п р а ж н е н и е. Перепишите нижеследующие числа, не используя «опущенной десятки»:

- | | | |
|-------------------|-------------------|-----------------|
| 1) $+7.293_{10}8$ | 3) $10+3$ | 5) $-_{10}-6$ |
| 2) $98.12_{10}+2$ | 4) $-1834_{10}-5$ | 6) $-4.8_{10}3$ |

1.3. И м е н а служат для обозначения различных объектов, например для величин, нужных в программе.

Именем может служить любая последовательность букв и цифр, начинающаяся с буквы.

Примеры имен:

<i>A</i>	<i>x</i>	<i>B3</i>
<i>a</i>	<i>a1b2c3d4e5</i>	<i>b3</i>
<i>Alpha</i>	<i>Moskwa</i>	<i>diagonal</i>

Индексы, как нижние, так и верхние, запрещаются. Ограничений на длину имени нет.

Выбор имен находится полностью в руках программиста. Удобно выбирать имена, указывающие на смысл объекта. Например, x^2 для обозначения x в квадрате и π для числа π .

Заметим еще, что большие и малые буквы — буквы разные, так что

A и *a*; *Alpha* и *alpha* или *aLpha*

для алгола различны. Имя *B3* само по себе не будет восприниматься ни как $B \times 3$, ни как B в кубе, ни как B_3 . Некоторые ограничения на выбор имен накладывают общепринятые обозначения стандартных функций, например *sin*, с которыми мы встретимся далее.

В математике величина, как правило, обозначается одной буквой. Но есть и исключения:

$\sin \pi$, r_x , r_y , c_p , c_v .

Обозначения $\sin \pi$ никто не воспринимает как произведение букв. В разложении r_x , r_y вектора r по координатным осям буква x не может принимать никаких значений. Таким образом, обозначения алгола — его имена, состоящие из нескольких букв и цифр, — не так уж нам непривычны. Но, забегаая вперед, заметим, что в алголе нельзя пропускать знак умножения. Действительно, запись

ab

будет восприниматься как обозначение нового объекта, отличного от a или b и от их произведения $a \times b$.

Контрольное задание. Какие из нижеследующих записей могут служить именами?

- | | | |
|----------------|-------------------|------------|
| 1) 4711 | 4) <i>gamma</i> | 7) $a + b$ |
| 2) <i>PPP3</i> | 5) <i>start 1</i> | 8) $x(3)$ |
| 3) <i>3PPP</i> | 6) $+ \epsilon$ | 9) $x[3]$ |

О т в е т: 2, 4, 5. Пробел в записи (имени 5) не играет роли.

§ 2. Описание синтаксических таблиц

То, что было описано в § 1, — это, к сожалению, почти все, что можно было сказать, не излагая всей системы определений алгола. Любое другое понятие алгола, во всей его общности, определяется только через все остальное. Это создает особые трудности как для изложения, так и для изучения алгола.

Авторы алгола разделили его грамматику на *синтаксис* и *семантику*, т. е. на правила построения фраз и на описание смысла фраз (а также запрещение некоторых словообразований).

2.1. *Семантика* алгола описывается обычным образом, а для описания *синтаксиса* авторы алгола употребляют специальный язык *Бэкуса*. Проще всего объяснить этот язык на следующем примере. Понятие *имя*, с которым мы познакомились в § 1, определяется на языке Бэкуса так:

$$\begin{array}{c} \langle \text{имя} \rangle \\ \downarrow \\ \overline{\langle \text{буква} \rangle \mid \langle \text{имя} \rangle \langle \text{буква} \rangle \mid \langle \text{имя} \rangle \langle \text{цифра} \rangle} \end{array}$$

Уголки $\langle \rangle$ сами по себе ничего не означают, в них заключаются отдельные понятия, чтобы избежать путаницы при слитном чтении. Уголки — не знаки алгола. Они используются, чтобы писать про алгол, а не на алголе. Стрелка $\downarrow$ читается: «по определению есть». Тонкие черточки (в данном случае вертикальные) читаются: «или». Все определение читается так:

«Имя по определению есть:

или буква,

или $\langle \text{имя} \rangle$ (т. е. то, что уже является $\langle \text{именем} \rangle$),
вслед за которым написана буква,

или $\langle \text{имя} \rangle$, за которым написана цифра».

Таким образом, определение Бэкуса — это задание способа построения объекта. Мы видим, что *a* есть имя, ибо это буква; *ai* — тоже имя, ибо это имя *a*, за которым написана буква *i*. Строка *ai2* — тоже имя и т. д. В данном случае легко сообразить, что имя — это любая

*) В языке Бэкуса стрелок нет и вместо них пишется знак $::=$. Но в этой книге всюду пишутся стрелки.

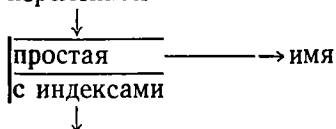
строка букв и цифр, начинающаяся с буквы. Но не всегда определение Бэкуса так просто перевести, а главное понять.

2.2. Обозначения в синтаксических таблицах (см. стр. 76). Рассмотрите первую синтаксическую таблицу (только не пытайтесь пока ее разбирать). В ней при первом упоминании понятие пишется полностью, в дальнейшем — сокращенно, с точкой на конце или без точки. Например:

ОПЕРАТОР или ОПЕР. или ОПЕР

Кроме этого случая точки употребляются в таблицах только при перечислении (три точки) в смысле «и так далее». Например, запись

переменная



<имя> [<арифметическое выражение>, ..., <ар. выр.>]

означает: <переменная> бывает <простая> или с <индексами>. В первом случае это <имя>. Во втором — это <имя, за которым в квадратных скобках написана строка арифметических выражений, разделенных запятыми; упомянутая строка не может быть пустой; если она состоит из одного арифметического выражения, то запятых не будет>.

§ 3. Простейшие арифметические выражения

Выражение, грубо говоря, — это правило, по которому можно вычислить значение выражения (когда заданы значения нужных переменных величин).

Основные определения выражений находятся в табл.2. Познакомьтесь с нею, не вникая в суть дела. Что такое <выражение>, мы узнаем позднее. Пока заметим, что выражения бывают разных классов и характеров. Класс бывает *арифметическим, булевым, именуемым*. Характер — *безусловным* или *условным*. Чтобы условное выражение сделать безусловным, надо заключить его в скобки.

3.1. Согласно сказанному арифметическое выражение бывает условным или безусловным. Условными мы зай-

мемся позже. Теперь прочтите в табл. 2 определение <безусловного арифметического выражения>. Там встречаются два новых понятия: <арифметическая переменная> и <арифметическая функция>. Эти понятия определены в табл. 1 и 3. Однако понятие функции для нас еще слишком сложно, а под <арифметической переменной> будем пока понимать имя (в дальнейшем это будет уточнено). Таким образом, для нас временно арифметическое выражение будет иметь смысл, определяемый рис. 1.

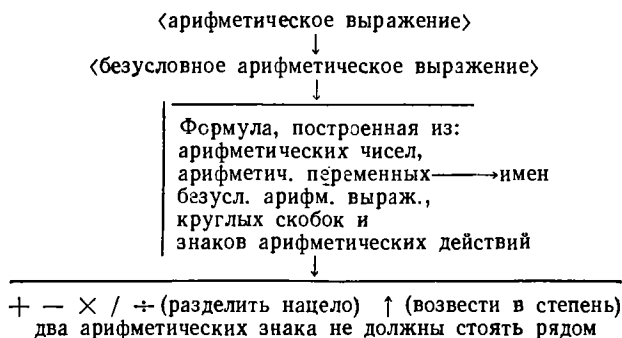


Рис. 1. Временное определение арифметического выражения.

3.2. Простейшие арифметические выражения строятся, как обычные алгебраические формулы, из арифметических чисел и переменных с помощью знаков арифметических действий и круглых скобок.

Запрещается употреблять иные знаки. Нельзя пропускать знак умножения или обозначать его точкой. Деление нельзя изображать двоеточием или дробью с горизонтальной чертой. Запрещается вместо круглых скобок писать квадратные или фигурные.

Примем временно следующее правило. При вычислении арифметического выражения порядок действий определяется скобками, а внутри них (или без них) сперва выполняются все возведения в степень, в порядке их следования (слева направо), потом все умножения и деления (в порядке следования) и, наконец, все сложения и вычитания (в том же порядке).

Это правило будет уточнено в 14.13.

3.3. Примеры арифметических выражений:

- 1) 16.48 2) fi 3) $3_{10}-5$ 4) $a + 2.83 - volt$
 5) $a/b \times c$ означает $(a/b) \times c$, но не $a/(b \times c)$

Примеры обычной записи и записи на алголе:

величина	верно	неверно
$-a^b$	$-a \uparrow b$ или $-a \uparrow (b)$	$(-a) \uparrow b$
a^{-b}	$a \uparrow (-b)$	$a \uparrow -b$
a^{b+c}	$a \uparrow (b+c)$	$a \uparrow b+c$
$10^{-4.7}$	$10 \uparrow (-4.7)$	$10-4.7$
$10^{4.7}$	$10 \uparrow 4.7$	$10^{4.7}$ или 104.7
$A \cdot B$	$A \times B$	AB или $A \cdot B$
$A \cdot (-B)$	$A \times (-B)$	$A \times -B$
a^{bc}	$a \uparrow (b \uparrow c)$	$a \uparrow b \uparrow c$
$(a^b)^c$	$a \uparrow b \uparrow c$ или $(a \uparrow b) \uparrow c$	$a \uparrow b \times c$
$\frac{a \cdot b}{c \cdot d}$	$a \times b / (c \times d)$	$a \times b / c \times d$
$a \cdot 10^4$	$a \times 10^4$	a_{10^4}

3.4. Действия $+$, $-$, $\times$ дают результат типа *целый*, когда оба аргумента — *целого* типа, иначе результат будет иметь тип *реальный*. Деление $/$ всегда дает *реальный* результат. Деление нацело ($\div$) определено только для *целых* аргументов и дает *целый* результат, равный целой части частного. Точнее:

$$a \div b = \text{целой части } \lceil a/b \rceil \text{ со знаком } a/b$$

Сложнее обстоит дело с возведением в степень. Возведение в степень с показателем типа *целый* понимается как умножение соответствующего числа нужных сомножителей. Например,

$$2.75 \uparrow (-3) = (1/2.75) \times (1/2.75) \times (1/2.75)$$

Результат возведения в степень бывает целым *только* в случае целого основания и целого неотрицательного показателя степени. Возведение отрицательного числа в реальную степень не определено. Не определено и $0 \uparrow 0$.

3.5. Остается указать на разницу между числом $314_{10}-2$ и выражением $314 \times 10 \uparrow (-2)$.

Запись $314_{10}-2$ указывает транслятору, чтобы он сделал нужные вычисления и в дальнейшем пользовался результатом, как готовым числом. В противоположность этому, запись $314 \times 10 \uparrow (-2)$ может побудить некоторые трансляторы пересчитывать выражение заново всякий раз, как оно понадобится.

З а м е ч а н и е. Обычно в алгебре принято пользоваться скобками разного вида: внутренние скобки зачатую пишут круглыми, внешние — квадратными, еще более внешние — фигурными и т. д.

Но если в алгебраической формуле все скобки заменить на круглые, то можно заметить, что смысл выражения не пострадает. Таким образом, достаточно и одних круглых скобок (которые для этой цели приняты в алголе). Скобки, разумеется, должны быть расставлены правильно — каждой открывающей должна соответствовать закрывающая и т. д. (нет нужды выписывать эти правила — они известны из школы).

§ 4. Простейшие операторы

Работа по программе алгола состоит в выполнении операторов. Операторы бывают (см. табл. 1): безусловные, условные, цикла. Безусловный оператор в свою очередь бывает: основным, составным, блоком. Наконец, основной оператор бывает: пустым, присваивания, перехода и процедуры. Пустой оператор не вызывает действий и никак не обозначается. Полезность его выяснится в дальнейшем. Изучение операторов начнем с оператора присваивания.

4.1. О п е р а т о р п р и с в а и в а н и я в простейшем случае имеет вид

переменная := выражение

и означает: вычислить выражение, стоящее справа, и полученное значение (число!) присвоить переменной, стоящей слева. Если выражение справа является числом, то, разумеется, вычислять ничего не надо. Знак присваивания := лучше всего понимать как «присвоить значение» или «в дальнейшем считать (таким-то) числом». Напомним, что для нас в настоящее время переменная — это имя.

Прежде чем двигаться дальше, отметим, что оператор $\langle := \rangle$ присваивает переменной слева именно число, а никак не алгебраическое выражение, стоящее справа. Так, например, оператор

$$x := a \times b$$

отнюдь не «обозначит через x произведение переменных a и b ». Он лишь перемножит числовые значения a и b

и присвоит букве x значение произведения. Если потом значения a и b сменятся, то значение x сохранится. Если же к моменту выполнения оператора присваивания переменные a или b не будут еще иметь численных значений, то переменной x может присвоиться случайное (непредвиденное) значение.

Вот несколько примеров присваивания:

Обычно	На алголе
$\beta = \frac{A + Bx}{C + Dx}$	$beta := (A + B \times x) / (C + D \times x)$
$F_y = 4k - k_1 k_2$	$Fy := 4 \times k - k1 \times k2$

Здесь имена переменных алгола выбраны так, чтобы они напоминали обычные обозначения. Но их, разумеется, можно было взять любыми.

Вот три *неверные* записи:

$$\begin{aligned} -m &:= 3.14 & 2A &:= A + A \\ a \times x \uparrow 2 + b \times x + c &:= 0 \end{aligned}$$

Действительно, слева должно находиться имя переменной. Оно должно начинаться с буквы (не может начинаться со знака минус или цифры). Не может оно содержать и знаков арифметических действий, как в последнем примере. Эти утверждения, впрочем, забегают вперед, ибо мы еще не определили понятия переменной в общем случае.

С помощью оператора присваивания можно задать значение переменной, которая еще не имеет значения. Но можно и изменить значение переменной, которая уже имеет значение. Например,

$$k := k + 1$$

увеличит значение k на единицу. Здесь хорошо видно отличие знака присваивания от знака равенства.

С помощью оператора присваивания можно *один и то же* значение присвоить нескольким переменным. Пишется это так:

$$\text{переменная} := \text{пер} := \dots := \text{пер} := \text{выражение}$$

«Многократное» присваивание не эквивалентно тем же присваиваниям, выполненным поочередно. Именно

$$a := 1; b := 1; a := b := a + b$$

присвоит переменным a и b одно и то же значение 2, в противоположность строке операторов

$$a := 1; b := 1; a := a + b; b := a + b$$

которая сделает a равным 2, но b равным 3.

У п р а ж н е н и я. Написать на алголе

$$u = a \cdot b + c^d - (2x)^3, l = \left(\frac{x + a + 3.14}{2x} \right)^3;$$

$$z = x + y^3, f = \frac{x}{(x)^2} \cdot \frac{1}{1 + \frac{(2x)^2}{3 + 5 + (3x)^3}}.$$

Между символами алгола можно оставлять пробелы и писать их в разных строках, что никак не влияет на смысл написанного. Этим пользуются, чтобы человеку было легче прочесть программу. Но *нельзя повторять знаки действий* при переносе выражения в новую строку (как принято в математике). Вот удобное, неудобное и неверное решение последнего примера:

- 1) $f := x / (1 + x \uparrow 2 / (3 + (2 \times x) \uparrow 2 / (5 + (3 \times x) \uparrow 2)))$;
- 2) $f := x / (1 + x \uparrow 2 / (3 + (2 \times x) \uparrow 2 / (5 + (3 \times x) \uparrow 2)))$;
- 3) $f := x / (1 + x \uparrow 2 / (3 + (2 \times x) \uparrow 2 / (5 + (3 \times x) \uparrow 2)))$;

Это замечание не мешает нам ставить, как обычно, лишние скобки, которые помогают человеку читать формулу. Например,

$$h := (a \times x + u) + (b \times y + v) + (c \times z + w);$$

4.2. Метки и операторы перехода. Программа алгола записывается последовательностью операторов, *отделенных друг от друга точкой с запятой* (;). Обычно операторы выполняются в порядке следования слева направо. Чтобы изменить порядок выполнения операторов, употребляются, в частности, метки (см. табл. 2) и операторы перехода (см. табл. 1).

Меткой может служить как имя, так и целое число без знака. В последнем случае нули слева не считаются, т. е. метки 025 и 25 считаются одинаковыми *).

*) Впрочем, числовые метки не рекомендуются: в некоторых трансляторах они запрещены.

Перед любым оператором можно поставить метку, отделив ее от оператора двоеточием. В этом случае говорят, что оператор помечен (помечен данной меткой)

метка: оператор

Вот несколько примеров:

$M: x := x + 1; \text{ mini } t: s := s + v \times t;$
 $\text{variant 25: } x := x \uparrow 2; y := t \uparrow 2/2$

Метка перед оператором никак не влияет на его выполнение. При выполнении программы она пропускается.

О п е р а т о р п е р е х о д а имеет вид

go to <именующее выражение>

где **go to** — иероглиф алгола, а под именующим выражением мы будем пока понимать только метки. После оператора перехода будет выполняться оператор, помеченный данной меткой. Поэтому метку, написанную вслед за **go to** называют меткой-передатчиком, а метку, написанную перед оператором, — меткой-приемником.

Простейшим употреблением оператора перехода является возвращение машины к началу программы. Например, последовательность операторов

$s := 0; n := 1; M: s := s + 1/n \uparrow 2; n := n + 1; \text{ go to } M$

будет вычислять бесконечную сумму $s = 1/1^2 + 1/2^2 + \dots$

Пустой оператор тоже может быть помечен, например:

$a5;$

Так как помеченный оператор тоже является оператором, то и перед ним можно поставить метку. Впрочем, подобная конструкция

$u : a5: \text{ mini } t : s := k$

возникает редко.

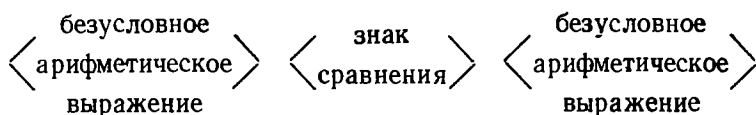
4.3. Простейшие булевские выражения.

4.3.1. Мы уже ввели два булевских числа **true** и **false** (по-русски — *да* и *нет*). Другим источником булевских чисел служат сравнения безусловных арифметических выражений (см. табл. 2). Например,

$5 > 3; \quad 7 \times 3 \uparrow 2 = 3 + 5; \quad x + a \leq 3 \times y$

Первое сравнение является числом **true**, второе — **false**. Значение третьего сравнения зависит от текущих значений целых или реальных величин x, a, y .

Вообще конструкция



определяет булевское число **true** или **false**.

Знак сравнения — это один из 6 знаков

$< \leq = \neq \geq >$

4.3.2. Простейшие булевские выражения строятся из булевских чисел и переменных с помощью круглых скобок и знаков булевских действий (см. табл. 2) подобно тому, как строились арифметические выражения.

Знаков булевских действий в алголе пять:

$\equiv \supset \vee \wedge \neg$

Называются они так: *совпадает, влечет, или, и, не*. Приведены они в порядке возрастания старшинства, которое нужно учитывать при определении порядка выполнения действий. Это правило будет уточнено в 14.13.

Значения булевских действий дает рис. 2.

a b	true true	true false	false true	false false
$a \equiv b$	true	false	false	true
$a \supset b$	true	false	true	true
$a \vee b$	true	true	true	false
$a \wedge b$	true	false	false	false
$\neg a$	false	false	true	true

Рис. 2.

Мы установили строгий порядок старшинства булевских действий. Тем не менее читателю рекомендуется не жалеть скобок и не пользоваться знаками, к которым он не привык. Употребительных знаков всего три $\vee \wedge \neg$ и

остальные через них выражаются легко. Например,

$$a \equiv b$$
 можно записать так:

$$(a \wedge b) \vee (\neg a \wedge \neg b).$$

Скобок здесь можно было и не писать.

4.3.3. С помощью оператора присваивания можно присвоить значение булевского выражения одной или нескольким переменным:

переменная $:= \dots$ перем $:=$ булевское выражение

Правила при этом действуют те же, что и в случае арифметического выражения.

	<i>ra</i>	<i>la</i>	<i>rb</i>	<i>bb</i>	<i>b</i>
1)	7.5	5			
2)			12.5		
3)					false
4)	4			┐	
5)					false
6)				false	
7)					false

Рис. 3.

Пример. Присвоить величине y значение **true**, если $0 \leq x < 1$, и значение **false** в противном случае.

Решение. $y := 0 \leq x \wedge x < 1$

Пример. Пусть x и y имеют булевские значения. Положить f равным **true**, если x и y совпадают или если они различны, но x равно **true**. В остальных случаях положить f равным **false**.

Приведем четыре решения:

- 1) $f := (x \equiv y) \vee (x \equiv \text{true});$ 2) $f := (x \equiv y) \vee x;$
 3) $f := x \vee \neg y;$ 4) $f := y \supset x;$

У п р а ж н е н и е. Найдите значение b в результате выполнения программы

$$\begin{aligned}ra &:= 7.5; ia := 5; \\rb &:= 3 \times ra - 2 \times ia; \\b &:= rb > ia \wedge ia > ra; \\ra &:= 2 \times (ra - ia) - 1; \\b &:= -ra > ia \vee b; \\bb &:= (b \equiv rb > ia) \wedge ra < rb; \\b &:= \neg (b \equiv bb)\end{aligned}$$

Р е ш е н и е. Напишите таблицу переменных и после каждого оператора заносите в нее новое значение вычисленной переменной (см. рис. 3). Нужно приобрести навык в такого рода выполнениях программ на бумаге.

§ 5. Условные выражения

Найдите в табл. 2 определение условного выражения, а в табл. 1 — определение условия. Согласно этим определениям *условное выражение* любого класса (арифметического, булевого или именующего) задается конструкцией

$$i: \left\langle \begin{array}{c} \text{булевское} \\ \text{выражение} \end{array} \right\rangle \text{ then } \left\langle \begin{array}{c} \text{безусловное} \\ \text{выражение} \\ \text{любого класса} \end{array} \right\rangle \text{ else } \left\langle \begin{array}{c} \text{выражение} \\ \text{этого же} \\ \text{класса} \end{array} \right\rangle$$

Здесь **if**, **then** и **else** — иероглифы алгола. Заметьте, что после **then** идет обязательно безусловное выражение. Запомните, что в алголе *запрещено писать подряд then if*.

Если выражение, написанное между **if** и **then**, имеет значение **true** (**false**), то условное выражение заменяется выражением, написанным между **then** и **else** (соответственно: написанным после **else**).

Условные выражения могут «вставляться» друг в друга, ибо между **if** и **then**, а также после **else** могут опять идти условные выражения.

Что делать, если по самому смыслу задачи выражение, идущее после **then**, является условным? Надо взять его в круглые скобки — от этого оно станет безусловным!

Теперь рассмотрим по отдельности арифметические, булевские и именующие условные выражения.

5.1. Условное арифметическое выражение. Пусть выражение должно равняться 0, если b имеет значение **true**, и единице, если b имеет значение **false**. Это выражение можно записать так:

if b then 0 else 1

Так же легко написать выражение, равное 3 при $x < 5$ и равное 7 при $x \geq 5$:

if $x < 5$ then 3 else 7

В правой части оператора присваивания (см. табл. 1) стоит числовое (т. е. арифметическое или булевское) выражение. До сих пор мы могли справа писать только безусловные выражения. Теперь наши возможности расширились.

Пусть x и y имеют арифметические значения и нужно положить $z = \max(x, y)$. Это можно сделать оператором

$z := \text{if } x \geq y \text{ then } x \text{ else } y$

Прочтите еще раз определение безусловного арифметического выражения в табл. 2 и ответьте на

К о н т р о л ь н ы й в о п р о с . Допустимо ли арифметическое выражение

$a + \text{if } x \geq y \text{ then } x \text{ else } y$

О т в е т : недопустимо. В арифметическом выражении слагаемым может быть только *безусловное* выражение.

Необходимо было писать скобки:

$a + (\text{if } x \geq y \text{ then } x \text{ else } y)$

К о н т р о л ь н ы й в о п р о с . Чему равно выражение

$\text{if } 5 > 3 \text{ then } 0 \text{ else } 1 + 3$

О т в е т : нулю. Действительно, это выражение расшифровывается только так:

$\text{if } 5 > 3 \text{ then } 0 \text{ else } (1 + 3)$

Если бы мы хотели написать

$(\text{if } 5 > 3 \text{ then } 0 \text{ else } 1) + 3$

то необходимо было писать скобки.

Задача. Задать y условиями:

если $x < 1$, то $y = \max(a, x)$,

если $x \geq 1$, то $y = 1$.

Решение можно записать так:

$$y := \text{if } x < 1 \text{ then (if } a \geq x \text{ then } a \text{ else } x) \text{ else } 1$$

и скобки в этом выражении опустить нельзя.

5.2. Аналогичным образом пользуются и условными булевскими выражениями. Только сами выражения могут выглядеть запутаннее.

Рассмотрим несколько примеров. Пусть

$$x := \text{if } k < 1 \text{ then } s > w \text{ else } h \leq c$$

Найдите значение x в следующих трех случаях (см. рис. 4). Ответы помещены в колонке x .

N	k	s	w	h	c	x
1	-1	2	2	4	3	false
2	2	2	2	4	3	false
3	1	4	5	2	2	true

Рис. 4.

Найдите значение y после выполнения оператора

$$y := \text{if } 3 < 5 \text{ then true else true} \wedge \text{false}$$

О т в е т: y принимает значение **true**. Наш оператор эквивалентен записи

$$y := \text{if } 3 < 5 \text{ then true else (true} \wedge \text{false)}$$

но не записи

$$y := (\text{if } 3 < 5 \text{ then true else true}) \wedge \text{false}$$

ибо знаки логических действий (подобно арифметическим знакам) ставятся лишь между безусловными выражениями.

Рассмотрим более сложное выражение

$$\begin{aligned} & \text{if if } a \text{ then } b \text{ else if } c \text{ then } d \text{ else } e \\ & \text{then true else if } f \text{ then false else true} \end{aligned}$$

где все переменные принимают булевские значения. Для расшифровки такого выражения нужно выделить группу идущих подряд **if then else**, за которыми стоят безусловные выражения. Эту группу надо заключить в скобки и снова искать такую группу. В нашем примере сперва найдутся группы

1) **if c then d else e**

2) **if f then false else true**

и выражение примет вид

**if if a then b else (if c then d else e) then true else
(if f then false else true)**

Теперь можно выделить группу

if a then b else (if c then d else e)

Выражение примет вид

**if (if a then b else (if c then d else e))
then true else (if f then false else true)**

Можно его переписать и так:

if A1 then true else A2

где в свою очередь

A1 есть **if a then b else A3**

A3 есть **if c then d else e**

A2 есть **if f then false else true**

5.3. Условные именуемые выражения. Пусть на некотором этапе решение задачи раздваивается. При $x \leq 0.05$ нужно закончить вычисления с помощью одного оператора, а при больших — с помощью другого. Пусть эти операторы помечены метками *L* и *M*. Тогда такое раздвоение можно осуществить с помощью условного именуемого выражения, написанного за иероглифом **go to**

go to if $x \leq 0.05$ then L else M

5.4. Приведем несколько распространенных примеров трудночитаемых выражений. С помощью скобок их можно написать гораздо яснее. Сделайте это сами.

Арифметические выражения

- 1) $\text{if } 5 < 7 \equiv 7 > 5 \text{ then } 5 + 7$
 $\text{else if } 3 > 2 \text{ then } 3 + 2 \text{ else } 1 + 2$
- 2) $\text{if } q > 0 \text{ then } s + 3 \times q/a \text{ else } 2 \times s + 3 \times q$
- 3) $\text{if } a > 0 \text{ then } u + v \text{ else if } a \times b > 17 \text{ then } u/v$
 $\text{else if } k \neq y \text{ then } v/u \text{ else } 0$

Булевские выражения

- 4) $\text{if if if } a \text{ then } b \text{ else } c \text{ then } d \text{ else } f \text{ then } g$
 $\text{else } h < k$
- 5) $g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$

О т в е т ы:

- 1) $\text{if } (5 < 7 \equiv 7 > 5) \text{ then } (5 + 7)$
 $\text{else (if } (3 > 2) \text{ then } (3 + 2) \text{ else } (1 + 2))$

Отсюда видно, что выражение равно 12.

- 2) $\text{if } (q > 0) \text{ then } (s + 3 \times q/a) \text{ else } (2 \times s + 3 \times q)$
- 3) $\text{if } (a > 0) \text{ then } (u + v) \text{ else (if } (a \times b > 17) \text{ then } (u/v)$
 $\text{else (if } k \neq y \text{ then } (v/u) \text{ else } 0))$
- 4) $\text{if (if (if } a \text{ then } b \text{ else } c) \text{ then } d$
 $\text{else } f) \text{ then } g \text{ else } (h < k)$
- 5) $g \equiv (((\neg a) \wedge b \wedge (\neg c)) \vee d \vee e) \supset (\neg f)$

§ 6. Условные и составные операторы

Прочтите определение условного оператора в табл. 1, опустив конструкцию с оператором цикла. Таким образом, условный оператор пока может иметь две формы.

6.1. Первая форма условного оператора:

$\text{if } \langle \text{булевское} \rangle \text{ then } \langle \text{безусловный} \rangle$
 $\text{выражение} \quad \text{оператор}$

или короче:

$\text{if } V \text{ then } P \quad (*)$

Действует этот оператор так: *если V имеет значение true, то P выполняется, а если false, то P пропускается.*

После этого выполняется оператор, написанный вслед за P (если, разумеется, при выполнении оператора P не произойдет отсылка к какому-либо иному оператору *)).

Поясним это двумя примерами. Рассмотрим

if V then $x := a$; Q

При V , равном **true**, произойдет, а при V , равном **false**, будет пропущено присваивание $x := a$, после чего в обоих случаях будет выполняться оператор Q .

В операторе

if V then go to M ; Q

при $V \equiv \text{false}$ будет выполняться Q , а при $V \equiv \text{true}$ выполнится оператор

go to M

программа перейдет к оператору с меткой M и, может быть, никогда не вернется к оператору Q .

У п р а ж н е н и е. Положить $x = \max(a, b)$.

Р е ш е н и е.

$x := a$; **if $b > a$ then $x := b$**

Несверно было бы писать решение так:

if $b > a$ then $x := b$; $x := a$

6.2. Вторая форма условного оператора имеет вид

if V then P else Q (*)

Здесь V — булевское выражение, P — безусловный оператор, Q — любой оператор (в том числе и условный).

Действует этот оператор так. При $V \equiv \text{true}$ выполняется P , а при $V \equiv \text{false}$ выполняется Q . После этого выполняется оператор, написанный вслед за Q (если, разумеется, сам выполненный оператор не содержал отсылки *)).

П р и м е р. Положить $x = \max(a, b)$.

) И если сам условный оператор () не является частью более общего оператора. С такой возможностью мы встретимся в дальнейшем.

Р е ш е н и е.

if $b > a$ **then** $x := b$ **else** $x := a$

6.3. Составной оператор. Если нужно объединить несколько операторов в *один*, или превратить условный оператор в безусловный, то нужные операторы (или оператор) заключаются в операторные скобки **begin ... end**. От этого они превращаются в один составной и, следовательно (см. табл. 1), *безусловный* оператор.

В условном операторе

if V **then** P

при $V \equiv \text{false}$ пропускается ровно один оператор, следующий за **then**. Как быть, если нужно пропустить несколько операторов? Для этого их нужно объединить в один *составной* оператор с помощью *о п е р а т о р н ы х* *с к о б о к* **begin** и **end**. Пусть, например, при $c < 0$ нужно положить $x := 0$; $y := c$, а при $c \geq 0$ надо положить $x := 1$; $y := -c$. Сделать это можно так:

$x := 0$; $y := c$; **if** $c \geq 0$ **then**
begin $x := 1$; $y := -c$ **end**

Или, используя вторую форму условного оператора,

if $c < 0$ **then** **begin** $x := 0$; $y := c$ **end**
else **begin** $x := 1$; $y := -c$ **end**

Между оператором и скобкой **end** можно не ставить точку с запятой (;).

Оператор P в условных операторах (*) должен быть безусловным. Если по самому смыслу задачи он оказывается условным, то для соблюдения грамматических правил его нужно заключить в операторные скобки.

З а м е ч а н и е. Операторные скобки **begin end** являются иероглифами алгола. В этой книжке наравне со скобками **begin end** будут употребляться фигурные скобки { }.

Если составной оператор невелик (умещается на одной строке), то мы его будем заключать в операторные скобки { }. Если же он не помещается в одной строке, то будем писать скобки **begin end**. При этом удобно писать соответствующие пары **begin** и **end** друг под другом, а строки между ними — сдвинутыми вправо.

Примеры.

- 1) $x := 0; y := c; \text{if } c \geq 0 \text{ then}$
 $\text{begin } i := 1; y := -x$
 end
- 2) $x := 0; y := c; \text{if } c \geq 0 \text{ then}$
 $\{x := 1; y := -c\}$
- 3) $\text{if } V1 \text{ then}$
 $\text{begin if } V2 \text{ then}$
 $\text{begin } P1; P2;$
 $\text{if } V3 \text{ then } \{P3; c := 0\} \text{ else } P4$
 end
 end .

Здесь $V1 - V3$ — булевские выражения, а $P1 - P4$ — операторы.

§ 7. Переменная с индексами

В математике часто употребляются переменные с индексами, например $a_4, a_i, a_{i,j}, \dots$. В алголе буквально так написать нельзя. В алголе *список индексов* заключается в квадратные скобки и индексы отделяются друг от друга запятыми: $a[4]; a[i]; a[i, j]; \dots$

7.1. Прочтите определение переменной в табл. 1, обратив внимание на определение переменной <с индексами>. Перечитайте определения безусловных арифметических и булевских выражений в табл. 2, понимая под <арифметическими переменными> и <булевскими переменными> пока что <переменные>.

Согласно этим определениям безусловные выражения могут строиться не только из простых переменных, но и из переменных с индексами. Кроме того, из определений видно, что индексом может служить не только число или простая переменная, но и любое арифметическое выражение, в частности снова переменная с индексами.

Вот несколько примеров переменных с индексами:

Обычно	на алголе
a_{17}	$a[17]$
$a_{i,j}$	$a[i, j]$
a_{2i+1, i_k}	$a[2 \times i + 1, j[k]]$

Но в алголе индексы могут иметь и более сложный вид:

$a[i/k, \text{if } u > T \text{ then } 5.2 \text{ else } j[t]]$

В тот момент, когда потребуется вычислить фактическое значение индекса, он будет вычислен по обычным правилам и затем округлен до ближайшего целого *).

7.2. В связи со сделанными обобщениями понятия переменной оператор присваивания может оказаться недоопределенным. Действительно, выполним строку операторов

$k := 2; a[k + 1] := k := k + 2$ (*)

В результате k станет равно 4, но какой переменной: $a[3]$ или $a[5]$ нужно присвоить значение 4? Это не ясно. При выполнении оператора присваивания в алголе предполагается *сначала вычислить индексные выражения у переменных, которым присваивается значение*, а затем выполнять присваивание. Тогда (*) даст $k := 4; a[3] := 4$.

При дальнейшем расширении понятия выражения (за счет привлечения <функций> и вызываемого ими побочного эффекта) и это доопределение может иногда оказаться неоднозначным. Поэтому сразу же приведем окончательное определение. Оператор присваивания

$a := \dots := u := A$

(где $a, \dots, u$ — переменные, а A — выражение) выполняется в три приема:

(1) все индексные выражения, встречающиеся у переменных $a, \dots, u$, вычисляются в порядке следования (слева направо);

(2) вычисляется выражение A ;

(3) полученное значение A присваивается переменным $a, \dots, u$ с теми значениями индексов, которые вычислены на шаге (1).

§ 8. Оператор цикла

О п е р а т о р ц и к л а, пожалуй, самая изящная конструкция алгола. Он имеет вид

$\text{for } x := \alpha, \beta, \dots, \nu \text{ do } P$

*) В отличие от деления нацело $\div$, где берется целая часть.

Здесь **for** и **do** — иероглифы; x — переменная (простая или с индексами), которую называют *параметром* цикла; $\alpha, \beta, \dots$ — *элементы* цикла; P — оператор.

Оператор цикла заставляет выполняться оператор P нуль или более раз для каждого элемента цикла, в порядке следования этих элементов. При этом происходит и изменение параметра x . Как это делается, определяется типом элемента цикла.

Найдите определение оператора цикла в табл. 1. Элементы цикла бывают трех типов — они называются арифметическим, прогрессии и пересчета и имеют вид

- 1) A
- 2) $A \text{ step } B \text{ until } C$
- 3) $A \text{ while } V$

где A, B, C — арифметические выражения, а V — булевское. Эти типы могут идти в списке $\alpha, \beta, \dots$ попеременно, в любом порядке. Знаки **step**, **until**, **while** — иероглифы алгола.

8.1. Короче всего определить оператор цикла, заменив его набором уже известных операторов, по следующему правилу:

1) Вместо каждого арифметического элемента написать оператор (с соответствующим выражением A)

$x := A; P;$

2) Вместо элемента прогрессии написать оператор

$x := A; M : \text{if } (C - x) \times B \geq 0 \text{ then}$
 $\{P; x := x + B; \text{go to } M\};$

3) Вместо элемента пересчета написать оператор

$M : x := A; \text{if } V \text{ then } \{P; \text{go to } M\};$

где A, B, C, V — выражения из соответствующих элементов цикла.

К этому нужно только добавить, что по исчерпанию всех элементов цикла значение параметра цикла считается неопределенным.

Так, например, оператор цикла

for $x := A1, A2, A3 \text{ step } A4 \text{ until } A5, A6 \text{ while } V1,$
 $A7 \text{ step } A8 \text{ until } A9, A10 \text{ do } P$

где $A1 - A10$ — арифметические выражения, $V1$ — булевское, заменяется последовательностью операторов (где x , $A1 - A10$, $V1$ и P заменены соответствующими переменными, выражениями и операторами, написанными на алголе):

```

 $x := A1; P; x := A2; P;$ 
 $x := A3; M1: \text{if } (A5 - x) \times A4 \geq 0 \text{ then}$ 
 $\{P; x := x + A4; \text{go to } M1\};$ 
 $M2: x := A6; \text{if } V1 \text{ then } \{P; \text{go to } M2\};$ 
 $x := A7; M3: \text{if } (A9 - x) \times A8 \geq 0 \text{ then}$ 
 $\{P; x := x + A8; \text{go to } M3\};$ 
 $x := A10; P;$  значение  $x$  не сохраняется;

```

Если определение 8.1 показалось вам недостаточно ясным, вернитесь к нему после 8.2—8.4.

8.2. Для каждого элемента арифметического типа оператор P выполняется один раз. Переменной x присваивается значение арифметического выражения A , после чего выполняется оператор. На этом элемент считается выполненным. Это можно описать так:

$$x := A; P;$$

Вот пример цикла, состоящего из элементов арифметического типа:

```

for  $x := 2.5, 1.1 + 0.9, 2 \times x + 0.5$  do  $y := x \uparrow 2$ 

```

Оператор $y := x \uparrow 2$ будет выполняться три раза: для $x = 2.5$, для $x = 1.1 + 0.9 = 2.0$ и для $x = 2 \times x + 0.5 = 4.5$.

Легко может случиться, что выполнятся не все элементы цикла, а оператор цикла прекратит работу. Например,

```

for  $x := 1, 2, 3$  do  $\{y := x \uparrow 2; \text{if } x > 1 \text{ go to } M\}$ 

```

Здесь выполнятся только два элемента. При выполнении второго элемента программа уйдет на метку M .

8.3. Элемент типа прогрессии вызывает несколько (в том числе и ни одного) выполнений оператора P . Работу по выполнению этого элемента приблизительно можно описать так. Сперва переменной x присваивается значение выражения A и выполняется оператор P :

$$x := A; P$$

Затем значение x увеличивается на величину B и спят выполняется оператор P :

$$x := x + B; P$$

Снова x увеличивается на B и опять выполняется P . И так до тех пор, пока значение x не выйдет за границу значения C . Таким образом, оператор P будет выполнен для x , пробегающего значения от A до C с шагом B . Нужно только иметь в виду, что значения шага B и границы C вычисляются заново перед каждым увеличением x и проверкой окончания (на случай, если они изменились при выполнении оператора P или при вычислении B и C) и что x сразу же может оказаться вне границ A, C .

Рассмотрим несколько п р и м е р о в.

Сложить два вектора: $c[i] := a[i] + b[i]$ для i от 1 до 5:

for $i := 1$ **step** 1 **until** 5 **do** $c[i] := a[i] + b[i]$

Сосчитать сумму квадратов нечетных чисел от 3 до 15:

$s := 0$; **for** $i := 3$ **step** 2 **until** 15 **do** $s := s + i \uparrow 2$

Сосчитать ту же сумму в обратном порядке:

$s := 0$; **for** $i := 15$ **step** -2 **until** 3 **do** $s := s + i \uparrow 2$

Последние два примера показывают, что надо уточнить фразу «пока параметр цикла не перейдет границу значения C ».

В алголе элемент типа прогрессий

for $x := \dots, A$ **step** B **until** $C, \dots$ **do** P

выполняется до тех пор, пока разность $C - x$ имеет тот же знак, что и шаг B (включая случаи, когда $C - x$ или B равно нулю).

К о н т р о л ь н ы й в о п р о с. Сколько раз выполнится оператор P в операторах цикла

1) **for** $i := 5$ **step** -1 **until** 6 **do** P

2) **for** $i := 5$ **step** 1 **until** 5 **do** P

О т в е т ы: нуль и один (в предположении, что сам оператор P не меняет значения i).

В элементе типа прогрессии шаг B и граница C могут быть и не целыми числами. Могут они быть и переменными. Например, оператор

$s := 0; \text{for } i := 1 \text{ step } i \text{ until } 64 \text{ do } s := s + i$

сосчитает сумму $s = 1 + 2 + 4 + \dots + 64$.

К о н т р о л ь н ы й в о п р о с . Сколько раз выполнится оператор P в цикле

$\text{for } i := 1 \text{ step } 3 - 2 \times i \text{ until } 5 \text{ do } P$

О т в е т : один раз. Действительно, на второй раз будет $i = 2$ и шаг $3 - 2i = -1$ получит знак, обратный знаку разности $5 - i = 3$, так что оператор P выполняться не будет.

При выполнении оператора цикла циклически выполняется ровно один оператор, написанный вслед за иероглифом **do**. Если нужно, чтобы выполнялось несколько операторов, то их объединяют в один составной оператор с помощью операторных скобок **begin end**.

З а д а ч а . Перемножить две квадратные матрицы $a[i, k], b[i, k]$ порядка n , т. е. вычислить

$$c[i, k] := \sum_{j=1}^n a[i, j] \times b[j, k]$$

Решение.

```
for i := 1 step 1 until n do
  for k := 1 step 1 until n do
    begin c[i, k] := 0; for j := 1 step 1 until n do
      c[i, k] := c[i, k] + a[i, j] × b[j, k]
    end
```

8.4. Элемент типа пересчета

$\text{for } x := \dots, A \text{ while } V, \dots \text{ do } P$

полагает переменную x равной значению выражения A и выполняет оператор P . Затем снова полагает x равной значению A и опять выполняет P . И так до тех пор, пока V имеет значение **true**. В частности, V может сразу принять значение **false**, и тогда оператор P не выполнится ни разу.

Примеры. 1) Сосчитать $s = 2^2 + 4^2 + \dots + 10^2$

$s := i := 0; \text{for } i := i + 2 \text{ while } i \leq 10 \text{ do}$
 $s := s + i \uparrow 2$

2) Для извлечения квадратного корня $y = \sqrt{x}$ часто пользуются алгоритмом Герона:

$$y_1 = 1; y_{n+1} = \left(\frac{x}{y_n} + y_n \right) / 2$$

Сделайте программу, продолжающую вычисления до достижения точности

$$|y_n^2 - x| \leq \varepsilon.$$

Решение.

$y := 1; \text{for } r := y \text{ while } y \uparrow 2 - x > \text{eps} \vee$
 $x - y \uparrow 2 > \text{eps} \text{ do } y := (x/y + y)/2$

Любопытно, что неверно решение

$\text{for } y := 1, y \text{ while } y \uparrow 2 - x > \text{eps} \vee x - y \uparrow 2 > \text{eps}$
 $\text{do } y := (x/y + y)/2$

Действительно, по окончании работы оператора цикла параметр y станет неопределенным и результат будет утерян.

8.5. Несколько примеров. Требуется выполнить оператор P для всех $x = 0, 0.1, \dots, 1$, кроме $x = 0.3$. Предостережем от неверного решения

$\text{for } x := 0 \text{ step } 0.1 \text{ until } 1 \text{ do if } x \neq 0.3 \text{ then } P$

Действительно, реальное число 0.1 записано приближенно. Поэтому мало надежды, что на четвертом цикле x будет точно равно 0.3. Да и на одиннадцатом цикле может оказаться $x > 1$. Верное решение будет, например, таким:

$\text{for } y := 0 \text{ step } 1 \text{ until } 10 \text{ do if } y \neq 3 \text{ then } \{x := y/10; P\}$

Выражения в элементах цикла могут быть и условными. Пусть оператор P надо выполнить для x , изменяющегося от 0 до 10 через 1, а от 10 до 100 через 5. Решение:

$\text{for } x := 0 \text{ step if } x < 10 \text{ then } 1 \text{ else } 5 \text{ until } 100 \text{ do } P$

Возможны и другие решения:

- 1) **for** $x := 0$ **step** 1 **until** 10, 15 **step** 5 **until** 100 **do** P
- 2) **for** $x := 0, x + 1$ **while** $x < 10, 10$ **step** 5 **until** 100 **do** P
- 3) **for** $x := 0, x + 1$ **while** $x \leq 10, 15$ **step** 5 **until** 100 **do** P

Но неверны решения:

- 4) **for** $x := 0$ **step** 1 **until** 10, $x + 5$ **while** $x \leq 100$ **do** P
- 5) **for** $x := 0$ **step** 1 **until** 9, $x + 5$ **while** $x \leq 100$ **do** P

ибо в случае 4) оператор P будет выполнен для

$$x = 0, 1, \dots, 10, 16, 21, \dots, 96$$

а в случае 5) — для

$$x = 0, 1, \dots, 9, 15, 20, \dots, 100$$

Происходит это потому, что согласно 8.1 элемент цикла типа прогрессии

for $x := \dots, A$ **step** B **until** $C, \dots$ **do** P

меняет параметр x вплоть до первого значения, вышедшего за границу C . Таким этот параметр и достается следующему элементу цикла. Это же обстоятельство надо учитывать и для элементов типа пересчета.

Но и оператор цикла может входить в условный оператор (см. определение оператора <условный> в табл. 1). Заметьте, что оператор цикла не может находиться между **then** и **else**. Если это потребуется, то оператор цикла придется взять в операторные скобки $\{ \}$. От этого он станет составным оператором и, следовательно, безусловным (см. табл. 1).

Объяснение. В алголе запрещена конструкция **then if**. Запрещение вызвано тем, что (недопустимый!) оператор

If $V1$ **then If** $V2$ **then** $P1$ **else** $P2$

допускает две расшифровки:

- 1) **if** $V1$ **then** (**if** $V2$ **then** $P1$ **else** $P2$)
- 2) **if** $V1$ **then** (**if** $V2$ **then** $P1$) **else** $P2$

Попутно конструкция **then if** оказалась запрещена и в выражениях (где она уже не вызывает затруднений).

Конструкция

If V **then for** $i := B$ **while** $V1$ **do if** $V2$ **then** P **else** Q

расшифровывается однозначно, но только (!), в силу запрета конструкции **then** <опер. цикла> **else**

§ 9. Стандартные функции и процедуры

9.1. В алголе имеются следующие стандартные функции (A — арифметическое выражение):

- $abs(A)$ — абсолютное значение A
- $sign(A)$ — знак A , равный соответственно $+1, 0, -1$ для $A > 0, A = 0, A < 0$
- $sqr(A)$ — квадратный корень из A
- $sin(A)$ — синус, аргумент в радианах
- $cos(A)$ — косинус, аргумент в радианах
- $arctan(A)$ — главное значение (от $-\pi/2$ до $+\pi/2$)
- $ln(A)$ — натуральный логарифм
- $exp(A)$ — экспонента, т. е. e^A
- $entier(A)$ — целая часть A (читается: антье) — наибольшее целое, не превышающее значения A

Функции $sign(A)$ и $entier(A)$ дают ответ типа *целый*, остальные — типа *реальный*.

Аргумент функции нужно обязательно заключать в скобки. Нужно, например, писать $sin(x)$, ибо запись $sin x$ воспринимается алголом не как функция, а как имя.

К о н т р о л ь н ы й в о п р о с . Чему равно

$entier(2.0)$

О т в е т : не ясно. Число 2.0 — число реальное, в машине записано приблизительно. Поэтому $entier(2.0)$ может оказаться 1 или 2 в зависимости от машины и транслятора.

Стандартные функции могут использоваться в арифметических выражениях (как <арифметические функции> — см. табл. 2). Тем самым мы еще раз расширяем понятие <выражения>.

Вот примеры, содержащие стандартные функции:

- 1) $a := sin(3.14/4); fi := arctan(y/x) [5]$
- 2) $if abs(x - y) < eps then go to M$
- 3) $exp(if x < y then x else y)$

Аргументы функций являются выражениями и, следовательно, могут снова содержать стандартные функции. Например,

$$a := sin(a + b \times cos(y/x))$$

Если одна и та же функция встречается несколько раз, например:

$$a := (a \times \sin(x) + b) / (c \times \sin(x) \uparrow 2 + d)$$

то для ускорения счета лучше вычислить ее однажды и запомнить:

$$\sin x := \sin(x); a := (a \times \sin x + b) / (c \times \sin x \uparrow 2 + d)$$

Обратите внимание на разницу между именем $\sin x$ и функцией $\sin(x)$.

9.2. Процедуры для ввода и вывода в алголе не были определены, так как они слишком связаны с составом оборудования конкретной машины (по мнению авторов алгола). Потом эти процедуры дорабатывались. И прежде чем пользоваться ими практически, необходимо прочесть описание транслятора, с которым придется работать.

В этой книге мы ограничимся стандартными процедурами ввода — вывода, бытующими в учебной литературе, но недостаточными для практической работы:

$$\text{input}(x, y, \dots, z); \text{output}(a, b, \dots, c)$$

Будем пока считать $x, y, \dots$ переменными (простыми или с индексами); $a, b, \dots$ — числовыми (т. е. арифметическими или булевскими) выражениями. Предполагается, что процедура *input* введет с перфоленты нужное количество очередных чисел и присвоит их значения переменным $x, y, \dots$; процедура *output* вычислит значения выражений $a, b, \dots$ и отпечатает эти значения подряд на бумаге.

Примеры использования ввода — вывода.

1) $\text{input}(a, b, c)$

2) $k := 3; \text{input}(a, b[k], c[5])$

3) $\text{for } k := 3 \text{ step } 1 \text{ until } 7 \text{ do input}(a[k])$

4) $\text{output}(1.7_{10} - 2)$

5) $k := 3; a[k] := 3.14/k; \text{output}(k \uparrow 2 \times \sin(a[k]/7.1))$

Программа 3 введет с перфоленты 5 очередных чисел и присвоит их значения переменным $a[3], \dots, a[7]$.

ГЛАВА 2

ОПИСАНИЯ

§ 10. Описания переменных

Программы, которые мы до сих пор писали, не являются на самом деле программами алгола. Это были всего лишь части программ. В действительности в алголе *все* имена должны быть *описаны*. И ни одно имя не может описываться *более одного раза* в начале данного блока.

Только в трех случаях имена не нуждаются в описаниях:

- 1) имена стандартных функций (*sin*, *cos*, ...) и процедур *input* — *output* не описываются. Ими можно пользоваться всюду;
- 2) метка не имеет явного описания;
- 3) не описываются формальные параметры в описаниях процедур.

С этими случаями мы встретимся в дальнейшем.

Познакомьтесь с определением блока в табл. 1. Заметьте, что в начале блока идут <описания>, разделенные точкой с запятой (;).

10.1. Прочтите <описание простой переменной> в табл. 3. Начало блока с этими описаниями может, например, выглядеть так:

- 1) {integer *i*; ...
- 2) {integer *i, j, k*; real *volt, x*; ...

Порядок описаний произволен. После иероглифа можно перечислить все простые переменные этого типа. Но

можно и повторить иероглиф. Равносильны описания:

- 1) {integer i, j, k ; Boolean $start$; ...}
- 2) {integer i , integer j, k ; Boolean $start$; ...}
- 3) {integer i, j ; Boolean $start$; integer k ; ...}

Пусть, для примера, нужно вычислить и отпечатать

$$a = 1 + 1/2 + \dots + 1/10.$$

Раньше мы сделали бы это так:

```
a := 0; for k := 1 step 1 until 10 do a := a + 1/k;  
  output (a)
```

Теперь мы видим, что имена a и k должны быть описаны и что программу нужно оформить в виде блока:

```
begin real a; integer k; a := 0;  
  for k := 1 step 1 until 10 do a := a + 1/k; output (a)  
end
```

Описания указывают транслятору, что после входа в блок через **begin** и до выхода из блока он должен резервировать ячейки для запоминания описанных величин. Попутно они дают ему и другие сведения, вытекающие из характера описаний (**integer**, **Boolean**, ...). После выхода из блока все эти ячейки (вообще говоря) освобождаются, а величины — исчезают. Выход из блока может произойти либо через его **end**, либо с помощью оператора перехода.

10.2. Описание массива. Если у нас есть три <простые переменные> x, y, z , то нужно описать все три имени: x, y, z . Но если мы пользуемся тремя реальными <переменными с индексами> $x[1], x[2], x[3]$, с общим именем, то они образуют *массив* и только одно имя — имя массива — нуждается в описании.

В описании массива (см. табл. 3) указывается тип переменной и границы значений его индексов. В нашем случае описание массива может выглядеть так:

```
{real array x[1:3]; ...}
```

Впрочем, иероглиф **real** перед **array** можно опустить.

Если переменная имеет несколько индексов, то в описании будет несколько **г р а н и ч н ы х** **п а р**. Массив

целых чисел $n [j, k]$, где j меняется от -3 до 17 , а k — от 6 до 8 , описывается так:

{integer array $n [-3 : 17, 6 : 8]; \dots$ }

Границы — не обязательно целые числа. Они *могут быть любыми арифметическими выражениями*. Надо только иметь в виду следующие обстоятельства: (1) При входе в блок программа должна *фактически вычислить* значения граничных пар. Поэтому переменные, встречающиеся в граничных парах, должны к этому моменту иметь значения*). (2) После соответствующих вычислений полученные значения границ будут округлены до ближайшего целого. (3) Левая граница пары не должна быть больше правой — иначе соответствующий индекс не сможет принимать никаких значений.

Приведем заведомо экзотическое описание

begin real a ; $a := 3.14/2 \times 0.8$;

{array $u [\sin(a) : 2 \times \sin(a)/\cos(a)]; \dots]; \dots$ }

Несколько массивов можно описать совместно, если они имеют одинаковый тип. Например, описание

1) Boolean array $i [-5 : 15], l [3 : 17, 6 : 8]$

равносильно строке описаний

2) Boolean array $i [-5 : 15];$ Boolean array $i [3 : 17, 6 : 8]$

Заметьте, что в первом случае массивы разделяются запятой (а не точкой с запятой). Если при этом несколько массивов (идущих подряд через запятые) имеют одинаковые границы индексов, то граничные пары можно указывать однажды — за последним из них. Например, описание

array $i [-5 : 15], k, l, [3 : 17, 6 : 8], r [1 : 3];$

равносильно описанию

array $i [-5 : 15], k [3 : 17, 6 : 8], l [3 : 17, 6 : 8], r [1 : 3];$

Пример 1. Пусть надо определить *треугольную* матрицу $a [i, j]$ для $i = 1, 2, \dots, 10$; $j = 1, 2, \dots, i$. Как описать этот двухиндексный массив на алголе?

*) Или быть выражениями, которые можно вычислить сейчас.

О т в е т: никак. На алголе можно описывать только прямоугольные массивы.

П р и м е р 2. Образовать массив

$$a[k] := 1/k, k = 1, 2, \dots, 10,$$

затем вычислить и отпечатать сумму

$$a = a[1] + a[2] + \dots + a[10].$$

Вот *недопустимое* решение:

```
begin integer k; real a; array a[1:10];  
  for k := 1 step 1 until 10 do a[k] := 1/k; a := 0;  
  for k := 1 step 1 until 10 do a := a + a[k];  
  output(a)  
end
```

Ошибка в том, что имя *a* оказалось описанным дважды (как реальная переменная и как массив) в начале одного блока. Выполнить наше задание буквально — затруднительно. Проще сменить имя *a* на *b* (для простой переменной).

§ 11. Влияние описаний на выполнение операторов и структуру выражений

11.1. Описание переменной влияет на значения, *присваиваемые* этой переменной. Происходит это как при выполнении оператора присваивания, так и при присваивании значений параметру цикла (при выполнении оператора цикла).

11.1.1. Если переменная описана типом **Boolean**, то ей могут присваиваться только булевы значения, если как **real** или **integer** — то только арифметические (т. е. реальные и целые).

11.1.2. Если переменная описана как **real**, то ей присваиваются реальные значения. Таким образом, программа

{real i; i := 0; ...

может присвоить переменной *i* значение и чуть большее, и чуть меньшее нуля.

Пр и м е р. Чему будет равно a после выполнения программы

```
{real  $a, k$ ;  $a := 0$ ;  
  for  $k := 1$  step 1 until 10 do  $a := a + k$ ; ...
```

О т в е т: неизвестно, ибо k — переменная **real**. На десятом цикле она может оказаться чуть больше десяти, и оператор $a := a + k$ выполнится 9 раз (a не 10).

11.1.3. Переменной, описанной как **integer**, присваиваются значения типа *целый*. Если присваиваемое значение окажется реальным, то оно будет предварительно округлено до ближайшего целого.

11.1.4. Согласно правилам алгола при выполнении программы (*неверной!*).

```
{integer  $k$ ; real  $a$ ;  $k := a := 1.2$ ; ...
```

переменным k и a должны быть присвоены *одинаковые* значения. Но в результате должно стать $k = 1$, $a = 1.2$. Как же быть?

Авторы алгола вышли из затруднения, запретив употреблять в одном операторе присваивания переменные разных типов (т. е. типа **real** и **integer**). Это и написано в табл. 1. Цена, разумеется, слишком дорогая. Программа

```
{integer  $k$ ; real  $a$ ;  $k := a := 0$ ; ...
```

в алголе недопустима. Нужно писать отдельно:

```
{integer  $k$ ; real  $a$ ;  $k := 0$ ;  $a := 0$ ; ...
```

В некоторых трансляторах оператор присваивания с переменными разного типа все же разрешен.

11.2. Теперь мы можем еще раз уточнить понятие <выражения>. Прочтите определение <безусловного арифметического выражения> в табл. 2. Обратите внимание на <арифметическую переменную>. Это — переменная (простая или с индексами), чье имя описано как *арифметическое*, т. е. как **integer** или **real**. Аналогичный смысл имеет и понятие <булевой переменной> в определении <безусловного булевского выражения>.

11.3. В процедурах

```
input ( $x, y, \dots, z$ ) output ( $a, b, \dots, c$ )
```

мы договорились считать параметры $x, y, \dots$ переменными

ми (простыми или с индексами); $a, b, \dots$ — числовыми выражениями (арифметическими или булевыми). Теперь дополнительно разрешим этим параметрам быть именами массивов. Если среди параметров *input* встретится имя массива, то будем считать, что с перфоленты введется нужное количество очередных чисел и их значения присвоятся всем элементам массива. Сколько нужно ввести чисел (т. е. сколько этих элементов), указано в описании массива.

Пр и м е р. Пусть на ленте идут подряд числа 1, 2, ... Тогда программа

```
begin integer i; array a[1:2]; input(i, a); ...
```

присвоит значения $i := 1$; $a[1] := 2$, $a[2] := 3$.

Если имя массива встретится среди параметров *output*, то будут отпечатаны подряд текущие значения всех переменных с индексами, составляющих этот массив.

Порядок ввода и вывода элементов многомерного массива ясен из следующего: $a_{11}, a_{12}, \dots, a_{1n}; a_{21}, a_{22}, \dots$

§ 12. Область действия описаний

Область действия описаний — одно из самых сильных решений задач теории программирования, сделанных авторами алгола. Оно существенно облегчает организацию программирования, хотя и вносит свои трудности в решение задач с большой информацией. Прочтите определения программы, блока и составного оператора в табл. 1. В начале блока идут описания — это отличает его от составного оператора. Заметьте, что блоки или не пересекаются, или один целиком содержится в другом.

Область действия описаний мы определим последовательно, начиная с самых внешних блоков (т. е. не содержащихся в других блоках) и переходя ко внутренним.

Во внешнем блоке программы, но вне его подблоков действуют те объекты, имена которых описаны в начале этого блока.

Пусть далее некоторый блок A непосредственно содержит блок B (т. е. не существует блока, содержащегося в A и содержащего B). Тогда в блоке B , но вне его подблоков действуют следующие объекты:

- 1) описанные в начале B ;
- 2) действующие в A , если только их имена не описаны в начале B .

Внутри блока можно пользоваться только объектами, которые в нем действуют. Все операторы блока работают только с этими объектами.

12.1. Рассмотрим программу

$$\{ \underset{A}{\text{integer } i, j; \dots} \{ \underset{B_1}{\text{integer } j, k; \dots} \}; \dots \{ \underset{B_2}{\text{integer } k, m; \dots} \}; \dots \} \underset{A}{}$$

содержащую всего три блока: A , B_1 , B_2 . Пусть в этой программе описаны только те имена, которые мы привели. Какова область действия переменных?

Переменная i действительна во всей программе — в блоке A и обоих его подблоках B_1 и B_2 . Переменная m действительна только в блоке B_2 .

Буквой k фактически обозначены две совершенно различные переменные. Переменная k , описанная в B_1 — назовем ее для краткости речи kB_1 , — действительна только в блоке B_1 . По выходе из этого блока kB_1 перестает существовать. Так же ведет себя переменная kB_2 в блоке B_2 .

Буквой j тоже обозначены две переменные — jA и jB_1 . Переменная jA действительна всюду вне блока B_1 . На время работы блока B_1 переменная jA как бы получает иное наименование, не встречающееся в программе, и сохраняет свое значение. В блоке B_1 действительна переменная jB_1 , к ней относятся действия операторов блока B_1 , если они содержат переменную j . После выхода из блока B_1 переменная jB_1 исчезает, а переменная jA снова приобретает свое наименование j и действует, в частности, в блоке B_2 .

12.2. *Имя метки считается описанным в минимальном блоке, охватывающем метку-приемник.* Поэтому в любом блоке, вне его подблоков, может быть только одна метка-приемник с данным именем*). Она недоступна для операторов, находящихся вне этого блока. Следовательно, войти в блок можно только через его **begin**.

Заметим, что в составной или условный оператор войти извне можно.

Кроме того, в алголе запрещено входить извне внутрь оператора цикла.

*) И не может находиться описание иного объекта с тем же именем, что у рассматриваемой метки-приемника.

12.3. Несколько примеров. Что отображают нижеследующие программы?

- 1) `{integer i; k := 1; {integer k; k := 2; output (k)}}`
- 2) `{integer k; k := 1; {integer k; k := 2; output (k);
output (k)}}`
- 3) `{integer k; {integer k; k := 3; output (k)}; output (k)}`

О т в е т ы. Программа 1 неверная. Оператор `k := 1` работает с переменной, которая не описана. Программа 2 отображает 2 и 1. Программа 3 неверная. Последний `output (k)` выполнить нельзя, ибо во внешнем блоке переменная `k` не получила значения.

4) Какой метке передаст управление оператор `go to` в нижеследующей программе, если все скобки `{ }` обозначают блоки и иных блоков в программе нет?

`{... M: ... { ... { ... goto M; ... { ... M: ... } ... } ... { ... } ... M: ... } ... }`
 1 2 3 4 4 3 5 5 2 1

О т в е т: самой правой. Действительно, метка `M` блока 4 считается описанной в нем и недоступна оператору блока 3. Самая левая метка `M` недоступна оператору блока 3 потому, что в блоке 2 уже есть метка-приемник `M`. Эта последняя метка считается описанной в блоке 2, и ее описание прекращает в блоке 2 действие метки `M` из блока 1.

Если какой-нибудь из примеров 12.3 вы сделали неверно, то вернитесь к началу § 12.

12.4. Пусть в большой программе используются переменные `x`, `y`, `z`, `u`, `v`. Пусть, далее, выделены две части задачи. В одной части по `y` должно находиться `u = u (y)`. В другой — по `z` находится `v (z)`. Эти части поручаются двум сотрудникам. Первому сообщают, что к моменту входа в его блок переменная `y` получит определенное значение, и указывают алгоритм для вычисления `u = u (y)`. Второй получает аналогичные сведения про `z` и `v`.

Теперь оба сотрудника могут работать независимо друг от друга (и от составителей всей программы). Им могут в процессе работы понадобиться дополнительные переменные, которые каждый волен обозначать по-своему. Дополнительные обозначения будут описаны в своих блоках. Не беда, если они окажутся одинаковыми у обоих сотрудников и если снова встретится обозначение `x`. Величина `x` общей программы от этого не пострадает.

12.5. Перед описанием *типа* какой-либо величины можно поставить дополнительно иероглиф **own** (собственный). При выходе из блока значение собственной величины сохраняется неизменным вплоть до повторного входа в этот же блок (в отличие от несобственной величины, значение которой при выходе из ее блока теряется).

Чтобы использовать эту особенность собственной величины, нужно при *первом* входе в ее блок присвоить ей значение, а при повторных — обойти это присвоение. Сделать это можно только с помощью значения какой-либо величины, меняющейся во внешнем блоке. Например,

```
begin integer i; for i := 1 step 1 until 10 do
  begin own real s;
    if i = 1 then s := 0; s := s ↑ 2 + 1/i;
    if i = 10 then output (s)
  end
end
```

При первом входе во внутренний блок переменной *s* будет присвоено значение 0. В дальнейшем присваивание будет обходиться и новые значения *s* будут получаться из старых. Если опустить **own**, то программа станет иной.

Нижеследующие примеры иллюстрируют использование алгола для написания программ и могут служить читателю в качестве упражнений с ответами.

1) Вычислить и напечатать таблицу значений полиномов Чебышева $T_n(x) = \cos(n \arccos x)$ для $n = 0, 1, \dots, 10$ и $x = 0, 0.1, \dots, 2$. Воспользоваться рекуррентной формулой

$$T_0(x) = 1; T_1(x) = x, T_{n+2}(x) = 2x T_{n+1}(x) - T_n(x).$$

Р е ш е н и е.

```
begin integer i, n; real a, b, c, x;
  for i := 0 step 1 until 20 do
    begin x := i/10; output (1); output (x); a := 1; b := x;
      for n := 2 step 1 until 10 do
        {c := 2 × x × b - a; a := b; b := c; output (c)}
      end
    end
```

2) Сосчитать и отпечатать число сочетаний

$$C_{0;0}^0; C_{1;1}^0, C_{1;1}^1; \dots, C_{10;10}^{10},$$

пользуясь рекуррентной формулой

$$C_n^0 = 1; C_n^m = \frac{n-m+1}{m} C_n^{m-1}.$$

Р е ш е н и е.

```
begin integer m, n; real c;
  for n := 0 step 1 until 10 do
    begin c := 1, output (c); for m := 1 step 1 until n do
      {c := (n - m + 1)/m × c; output (c)}
    end
  end
```

3) Сосчитать и отпечатать таблицу значений функции

$$I_0(x) = 1 - \frac{(x/2)^2}{(1!)^2} + \frac{(x/2)^4}{(2!)^2} - \dots$$

для $x = 0, 0.01 \dots, 2.00$. Вычисление ряда продолжать, пока очередной член станет меньше накопленной суммы в 10^{-6} раз по абсолютной величине.

Р е ш е н и е.

```
begin integer k; real y;
  for k := 0 step 1 until 200 do
    begin y := (k/200) ↑ 2;
      begin integer k; real a, s;
        a := s := 1;
        for k := 1, k + 1 while abs (a) ≥10 - 6 × s do
          {a := -a × y / (k × k); s := s + a}; output (s)
        end
      end
    end
```

4) Вычислить и отпечатать (для $x = 0.8$; $a = 3.5$; $b = 4.5$)

$$y = \frac{1}{a \sqrt{a^2 + b^2}} \operatorname{arctg} \frac{a \times x}{\sqrt{a^2 + b^2}}.$$

Р е ш е н и е.

```
begin real r, y; r := sqrt (3.5 ↑ 2 + 4.5 ↑ 2);
  y := 1/(3.5 × r) × arctan (3.5 × 0.8/r); output (y)
end
```

§ 13. Переключатели

13.1. После иероглифа **go to** должно стоять именующее выражение (см. табл. 2). В общем случае именующее выражение составляется не только из меток, но и из *переключателей* (см. табл. 2).

Оператор перехода с переключателем имеет вид

$$\text{go to } P [A] \quad (1)$$

где P — имя (данного переключателя), A — арифметическое выражение. Переключателю (1) соответствует описание переключателя (см. табл. 3):

$$\text{switch } P : = I, J, K, \dots, M \quad (2)$$

где **switch** — иероглиф алгола, P — имя того же переключателя (1), а $I, J, \dots$ — переключательный список, состоящий из именующих выражений, разделенных запятыми.

Работа оператора состоит из трех действий:

(1°) Вычисляется значение A и округляется до ближайшего целого (об округлении напоминают квадратные скобки). Пусть для примера получено число 3.

(2°) Отыскивается (2) — описание данного переключателя P . В его переключательном списке находится член с нужным номером (считая слева направо). В нашем случае номер равен 3 и искомый член есть K .

(3°) Выполняется воображаемый оператор

$$\text{go to } K \quad (3)$$

Это требует трех уточнений. Во-первых, описание (2) отыскивается в начале блока, охватывающего оператор; если таких описаний несколько, то берется то, которое в минимальном блоке. Короче говоря, описание данного переключателя находится по тем же правилам, что метка-приемник. Во-вторых, воображаемый оператор (3) выполняется так, как если бы он был первым оператором в блоке, где найдено описание (2) (а не так, как если бы оператор (3) стоял на месте оператора (1), что иногда не одно и то же). В-третьих, если действия 1° и 2° не приведут к результату, то оператор (1) считается неопределенным и пропускается. Это случится, если целое число $[A]$ будет < 1 или превзойдет число членов переключательного списка (2).

Может быть, не лишне отметить, что выбранное имеющее выражение K может оказаться не меткой, а переключателем (с тем же именем P или иным), и тогда выполнение (3) в свою очередь отошлет к описанию соответствующего переключателя.

Описание переключателя (2) само по себе никаких действий не вызывает и при выполнении программы пропускается (как метки-приемники и иные описания). Только оператор перехода (1) заставляет работать описание (2).

13.2. Вот несколько примеров:

```
1) begin integer k; switch P := L, M, N;
    go to P [2]; L: go to P [k + 2]; M: k := 1; go to P [k];
    N: end
```

Сперва `go to P [2]` через описание переключателя P отошлет на M . Потом $P [k]$ через описание переключателя P отошлет на L . Наконец, $P [k + 2]$ посредством описания P отошлет на N .

```
2) begin integer k; switch P := M, g [k]; switch g := S, T;
    M: k := 1; T: go to P [if k = 1 then k + 1 else 1];
    S: end
```

Порядок действий можно описать так:

```
k := 1, go to P [2], switch P, go to g [1], switch g, go to S, S
```

3) Что отпечатает нижеследующая программа?

```
begin integer k; switch P := M, Q [k]; switch Q := R, S;
    k := 1; {integer k; k := 2; go to P [k]; R: output (1)};
    go to R; M: output (2); go to R; S: output (3);
```

$R: end$

О т в е т: ничего. Сменив обозначение k внутренней переменной на $k1$, мы сможем так описать порядок действий:

```
k := 1, k1 := 2, go to P [k], go to Q [k], go to R; R: end
```

«Воображаемый» оператор `go to Q [k]` выполнялся так, как будто он написан во внешнем блоке. Поэтому его аргумент $k = 1$. По этой же причине воображаемый оператор `go to R` отослал к метке-приемнику R внешнего блока.

13.3. Применение переключателей.
Пусть программа должна содержать пять непересекающихся блоков с метками U, A, B, C, D :

$\{ \dots U: \{ \}; A: \{ \}; B: \{ \}; C: \{ \}; D: \{ \}; \dots \}$

Вначале работает блок U , а затем один из блоков $A—D$. Какой именно — выясняется при выполнении блока U . Если это выясняется в конце блока U , то естественно сочетать «выяснение» с нужным переходом. Но легко предвидеть задачи, в которых выяснение произойдет в середине блока U , когда переходить еще рано. Сперва нужно будет закончить блок U . Тогда придется *запомнить*, куда именно следует перейти после блока U . Как это сделать?

Вот естественное, но недопустимое для алгола решение. В конце блока U поставить $\text{go to } M$. Если в середине U выяснится, что перейти нужно будет к A , то тут же метке M присваивается значение A с помощью оператора $M := A$. *Это решение неверно*, ибо оператор присваивания работает с арифметическими или булевскими переменными, но не с именующими.

Правильное решение может быть таким. Введем переменную i и будем присваивать ей значения $1, 2, \dots$ в случаях, когда перейти нужно будет к меткам $A, B, \dots$. Переход произведем с помощью операторов, написанных в конце блока U . Их можно написать по-разному.

Вот самый громоздкий способ:

$\text{if } i = 1 \text{ then go to } A \text{ else if } i = 2 \text{ then go to } B \text{ else}$
 $\text{if } i = 3 \text{ then go to } C \text{ else go to } D$

Вот еще вариант

$\text{go to if } i = 1 \text{ then } A \text{ else if } i = 2 \text{ then } B$
 $\text{else if } i = 3 \text{ then } C \text{ else } D$

Но если таких блоков U окажется в программе несколько, то лучше в начало программы ввести описание переключателя $\text{switch } P := A, B, C, D$ и выход из блока U закончить оператором $\text{go to } P[i]$. Вся программа примет вид

$\text{begin switch } P := A, B, C, D; \{ \text{integer } i; \dots \text{go to } P[i] \};$
 $A: \{ \}; B: \{ \}; C: \{ \}; D: \{ \} \text{ end}$

ГЛАВА 3

ПРОЦЕДУРЫ

§ 14. Процедуры

По целому ряду причин, ясных для программиста, возникает необходимость пользоваться подпрограммами, написанными вне тех мест, где этими подпрограммами, пользуются, и употребляющими иные обозначения переменных, чем введенные в местах пользования. Авторы алгола решили эту проблему с максимальной универсальностью.

14.1. Чтобы обратиться к подпрограмме, в алголе используется *оператор процедуры*, а сама подпрограмма задается *описанием* соответствующей *процедуры*.

Прочтите в табл. 1 второе определение <процедуры> и в табл. 3 второе определение <описания процедуры>, игнорируя все, что относится к <строкам>, <значениям>, <спецификациям>, <коду> и <замене запятых>.

Таким образом, для нас сейчас обращение к подпрограмме осуществляется оператором процедуры вида

$$P(A, B, \dots, E) \quad (1)$$

где P — имя данной процедуры, за которым в круглых скобках идет список фактических параметров — *выражений*, разделенных запятыми.

Нужная подпрограмма сейчас задается описанием процедуры вида

$$\text{procedure } P(U, V, \dots, W); Q \quad (2)$$

где *procedure* — иероглиф алгола; P — то же имя, что и в (1), за которым в круглых скобках идет список

формальных параметров — попарно различных *имен*, разделенных запятыми; наконец, Q — оператор, называемый телом процедуры. Фактических параметров в (1) должно быть столько же, сколько формальных параметров в (2).

Описание процедуры (2) само по себе никаких действий не вызывает. При выполнении программы оно пропускаяется целиком (вместе с телом Q). Работу описания (2) вызывает оператор процедуры (1).

Оператор процедуры (1) выполняется так:

1°. Отыскивается описание (2), и фактические параметры ставятся в соответствие формальным параметрам в порядке следования.

2°. В операторе Q формальные параметры заменяются фактическими параметрами. Оператор Q после замены называется $\tilde{Q}$ — *модифицированным* телом процедуры.

3°. Выполняется $\tilde{Q}$, как если бы он стоял на месте оператора процедуры (1).

В сложных случаях это требует многих уточнений, которые мы отложим до 14.3—14.5.

14.2. Проиллюстрируем применение оператора процедуры на простейших примерах.

1) Пусть мы хотим иметь подпрограмму, переводящую прямоугольные координаты в полярные в правой полуплоскости, т. е. дающую по двум числам $x > 0$, y величины $r = \sqrt{x^2 + y^2}$, $\varphi = \arctan(y/x)$.

Описание этой процедуры можно сделать таким:

$$\begin{aligned} & \text{procedure polar } (x, y, r, \varphi); \\ & \{r := \text{sqrt}(x \uparrow 2 + y \uparrow 2); \varphi := \text{arctan}(y/x)\} \end{aligned} \quad (3)$$

Если теперь встретится оператор

$$\text{polar } (a, b, R, F) \quad (4)$$

то переменным R и F будут присвоены значения

$$R = \sqrt{a^2 + b^2}; \quad F = \arctan(b/a)$$

Предполагается, конечно, что в блоке оператора (4) уже действуют переменные a , b , R , F и что переменным a , b уже присвоены арифметические значения.

Но оператор процедуры может иметь и более сложный вид. Например,

$$\text{polar } (a - b, -d, ro, psi)$$

При его выполнении будут сделаны нужные замены в теле (1) и переменные *ro*, *psi* получат нужные значения. Все произойдет так, как если бы выполнялся оператор (модифицированное тело)

$$\begin{aligned} ro &:= \text{sqrt} ((a - b) \uparrow 2 + (-d) \uparrow 2); \\ psi &:= \text{arctan} (-d / (a - b)) \end{aligned}$$

В этом примере при замене формальных параметров фактическими мы ввели дополнительные скобки. Предполагается, что нужные скобки действительно вводятся при образовании модифицированного тела.

Имена некоторых формальных параметров могут совпадать с фактическими. Это не исказит результата.

Оператор процедуры

$$\text{polar} (y, x, F, r)$$

выполнится как

$$F := \text{sqrt} (y \uparrow 2 + x \uparrow 2); r := \text{arctan} (x/y)$$

Оператор процедуры

$$\text{polar} (x, x, r, F)$$

выполнится как

$$r := \text{sqrt} (x \uparrow 2 + x \uparrow 2); F := \text{arctan} (x/x)$$

Наконец, оператор

$$\text{polar} (a, b, u + v, f)$$

не сможет быть выполнен, ибо в модифицированном теле возникнет недопустимая конструкция

$$u + v := \text{sqrt} (a \uparrow 2 + b \uparrow 2);$$

2) Опишем процедуру для суммирования элементов какого-либо массива *a* [*i*], *i* = 1, 2, ...

procedure *S2* (*a*, *n*, *S*);

begin integer *k*; *S* := 0;

for *k* := 1 **step** 1 **until** *n* **do** *S* := *S* + *a* [*k*]

end

Теперь оператор

$$S2 (b, 3, u)$$

даст

$$u := b[1] + b[2] + b[3]$$

если только в массиве b имеются соответствующие члены.

Оператор процедуры

$$S2(b, k, u)$$

тоже выполнится верно и даст

$$u := b[1] + b[2] + \dots + b[k]$$

Ясно, что во время этого вычисления имя k из оператора процедуры (которое заменит n в описании процедуры) и имя k , описанное в самом теле процедуры, должны считаться различными.

3) Для вычисления сумм типа

$$S3 = a[m] + a[m+1] + \dots + a[n]$$

процедура $S2$ не годится. Придется ввести более общую

procedure $S3(a, m, n, s);$

{integer $k; s := 0; \text{for } k := m \text{ step } 1 \text{ until } n \text{ do } s := s + a[k];$

4) Процедура $S3$ суммирует члены подряд. Можно построить процедуру, свободную от этого ограничения. Нужно только параметр цикла ввести в число формальных параметров:

procedure $S4(a, k, m, n, s);$

{s := 0; for $k := m \text{ step } 1 \text{ until } n \text{ do } s := s + a;$

Чтобы найти $s = \sum_{k=m}^n a[k]$, к этому описанию нужно обратиться оператором

$$S4(a[k], k, m, n, s)$$

Но теперь можно найти и сумму

$$s = a[2] + a[4] + \dots + a[2 \times r]$$

Ее доставит оператор

$$S4(a[2 \times i], i, 1, r, s)$$

при выполнении которого в описании процедуры $S4$ возникнет модифицированное тело

{s := 0; for $i := 1 \text{ step } 1 \text{ until } r \text{ do } s := s + a[2 \times i];$

Введение параметра цикла в число формальных параметров — прием мощный и неожиданный. Тщательно разберите описание процедуры S4. Убедитесь, что суммы $\sum$, равные

$$\sum_{i=m}^n (2i+1)^3; \quad \sum_{i=m}^n x[i] \times y[i];$$

$$\sum_{i=0}^n a[i] \times x \uparrow i; \quad \sum_{j=1}^m a[i, j] \times b[j, k];$$

доставляются соответственно операторами

S4((2 × i + 1) ↑ 3, i, m, n, s)

S4(x[i] × y[i], i, m, n, s)

S4(a[i] × x ↑ i, i, 0, n, s)

S4(a[i, j] × b[j × k], j, 1, m, s)

К о н т р о л ь н ы й в о п р о с . Что отпечатает программа?

```
begin integer i; real s; procedure V(s); s := 0;
  {s := 1; for i := 1 step 1 until 3 do s := s + i};
  V(s); output(s)
end
```

О т в е т : нуль. Действительно, тело процедуры V состоит из оператора $s := 0$ и только из него. Вообще-то говоря, в этой программе три оператора: в фигурных скобках, V и печать. Так что первым выполнялся оператор в фигурных скобках. Затем оператор процедуры — он отослал к описанию процедуры, где выполнилось тело $s := 0$. Потом произошла печать.

14.3. Уточним порядок, в котором отыскивается описание процедуры. Описание процедуры (2) отыскивается в начале какого-либо блока, содержащего оператор процедуры (1). Если таких описаний окажется несколько, то берется то, которое в минимальном блоке (т. е. так же, как отыскивается метка-приемник).

14.4. Уточним образование и выполнение модифицированного тела. Неясности здесь могут возникнуть главным образом потому, что различные (описанные в разных блоках) объекты программы могут оказаться обозначены одинаковыми именами и метками.

14.4.1. Предварительно разобьем объекты тела на три группы: локальных, формальных и глобальных.

И запомним про каждый объект, в какую группу он попал.

Разбиение проведем так:

1°) **Локальные**. Это объекты, описанные в самом теле. Сюда же отнесем метки, *помечающие операторы тела*. Тело процедуры всегда считается блоком, а упомянутые метки считаются описанными в нем.

(2°) **Формальные**. Сюда отнесем объекты тела, обозначенные именами формальных параметров, если только эти объекты не попали уже в группу локальных.

(3°) **Глобальные** — не попавшие в предыдущие две группы.

Таким образом, сперва выделяются локальные объекты, а уже *среди оставшихся* — формальные.

Приведем примеры определения локальных, формальных и глобальных объектов тела процедуры:

1) **procedure** $f1(x); \{ \text{integer } k; k := 1; x := i + k \}$

В теле этой процедуры k — локальный, x — формальный, i — глобальный.

2) **procedure** $f2(x); \{ k := 1; \{ \text{integer } k; x := i + k \} \}$

Здесь k (во внутреннем блоке) — локальный, x — формальный, k (во внешнем операторе) и i — глобальные.

3) **procedure** $f3(x); \{ \text{integer } x; x := i + k \}$

Здесь x — локальный, i, k — глобальные.

4) **procedure** $f4(x);$
 $\{ \{ \text{integer } x; x := k; k := 2 \times x \}; x := k \}$

Здесь x (во внутреннем блоке) — локальный, x (во внешнем операторе) — формальный, k — глобальный.

5) **procedure** $f5(M); \text{if } V \text{ then } N : u := 5 \text{ else go to } M;$

6) **procedure** $f6(M); \text{if } V \text{ then } M : u := 5 \text{ else go to } M;$

В теле $f5$ метка M — объект формальный, а в теле $f6$ — локальный (так как помечает оператор тела)

14.4.2. Перейдем к образованию модифицированного тела:

а) Все формальные объекты тела и только они заменяются фактическими параметрами (при этом добавляются

нужные круглые скобки). Объекты, *вставленные* в тело процедуры в результате этой замены, понимаются в том смысле, который действует там, где *расположен оператор процедуры*.

б) Обозначение вставленного объекта может иногда совпасть с обозначением локального объекта тела. Но и тогда эти объекты следует считать различными.

в) Глобальные объекты должны быть действены там, где *расположено описание* процедуры. И понимаются именно в этом смысле. Обозначение глобального объекта может, конечно, совпасть с обозначением некоторого объекта, действующего там, где расположен оператор процедуры. Но если эти объекты различны, то их следует считать различными. И с этой целью нужно временно сменить обозначение объекта, действующего в расположении оператора процедуры.

После этих преобразований модифицированное тело должно оказаться правильно написанным оператором алгола. Выполняется модифицированное тело так, как будто оно находится на месте оператора процедуры.

Таким образом, в настоящее время в алголе-60 принято половинчатое решение: глобальные объекты тела понимаются в смысле, действующем в блоке, где находится *описание* процедуры, но выполняется модифицированное тело так, как будто оно написано на месте оператора процедуры.

Вот несколько запутанных примеров программ с процедурами:

```
7) begin procedure f(x); x := i; integer i; i := 0;  
   {integer i, y; i := 1; f(y); output(y)}; ...
```

Эта программа отпечатает 0, ибо переменная *i* в теле процедуры — глобальная. Она отлична от переменной *i*, описанной во внутреннем блоке.

```
8) begin procedure f(x); x := i;  
   {integer i, y; i := 1; f(y); output(y)}; ...
```

Эта программа ошибочна. Глобальная переменная *i* тела процедуры не действует в блоке, где находится описание процедуры.

9) Операторы процедур *f5(L)* и *f6(L)*, будучи помещены в блоки описаний процедур примеров 5 и 6, приведут

соответственно к образованию модифицированных тел:

if V then $N : u : = 5$ else go to L
if V then $M : u : = 5$ else go to M

10) Что отпечатает программа

begin integer i ; procedure $F(u)$; go to u ; $i : = 0$;
go to T ; $M : output(i)$; go to u ;
 $T : \{integer\ k; F(M); M : k : = 1; output(k)\}$;

$u : end$

О т в е т: единицу. Действительно, сперва выполнится оператор $i : = 0$ (а не **go to u** , являющийся телом процедуры F). Затем оператор **go to T** . Теперь на месте оператора процедуры $F(M)$ должно выполниться модифицированное тело

go to M

где M — метка, вставленная в тело вместо формального параметра. Эта метка имеет смысл, действующий в блоке оператора процедуры $F(M)$. Поэтому модифицированное тело отправляет к оператору $k : = 1$.

11) В противоположность предыдущему примеру программа

begin integer i ; procedure $F(u)$; go to M ; $i : = 0$;
go to T ; $M : output(i)$; go to u ;
 $T : \{integer\ k; F(M); M : k : = 1; output(k)\}$;

$u : end$

отпечатает 0. Выполняться она будет в таком порядке:

$i : = 0$; go to T ; integer k ; $F(M)$

и модифицированное тело **go to M** с глобальным объектом M отошлет к оператору $M : output(i)$.

12) Нижеследующая программа ошибочна:

begin array $a[1 : 10, 1 : 10]$; procedure $F(b)$;
 $\{integer\ i$; for $i : = 1$ step 1 until 10 do $b[i] : = i$; $F(a)$
end

Дело в том, что в модифицированном теле возникнет оператор **$a[i] : = i$** , не являющийся правильным

алгольным оператором, поскольку переменная с именем a должна содержать не один индекс, а два (согласно описанию массива a).

14.5. Процедуры без параметров. Прочтите первое определение *оператора* процедуры в табл. 1 и первое *описание* процедуры в табл. 3. Они определяют: процедуру без параметров, которая для нас пока имеет вид $\langle P \rangle$ и ее описание

$\langle \text{procedure } P; Q \rangle$

где P — имя процедуры, а Q — ее тело. Эта процедура вовсе не имеет фактических (а ее описание — формальных) параметров. Все объекты тела Q — локальные и глобальные.

14.6. Пример применения процедуры без параметров. Пусть на протяжении некоторого расчета часто бывает нужно вычислять полярные координаты r, f по прямоугольным $x > 0, y$. При этом полярные координаты всегда будут обозначаться r, f , а декартовы x, y . Меняться от случая к случаю будут только их числовые значения. Тогда удобно ввести описание процедуры без параметров

procedure polar;

$\{r := \text{sqrt}(x \uparrow 2 + y \uparrow 2); f := \text{arctan}(y/x)\}$

и обращаться к нему коротким оператором процедуры

polar

Пусть надо вычислить и отпечатать

$$d = \prod_{i=1}^{10} r_i; \quad s = \sum_{i=1}^{10} f_i,$$

где r_i, f_i — полярные координаты вектора с прямоугольными координатами

$$x_i = i / (1 + i \uparrow 2); \quad y_i = 1/i.$$

Это можно сделать программой

begin real x, y, r, f ; procedure polar;

$\{r := \text{sqrt}(x \uparrow 2 + y \uparrow 2); f := \text{arctan}(y/x)\};$

integer i ; real d, s ;

$d := 1; s := 0;$

```

for  $i := 1$  step 1 until 10 do
begin  $x := i/(1 + i \uparrow 2)$ ;  $y := 1/i$ ;
      polar;  $d := d \times r$ ;  $s := s + f$ 
end; output ( $d, s$ )
end

```

Остается запомнить, что оператор и описание процедуры без параметров пишутся в форме P , а отнюдь не $P()$, с пустыми скобками.

14.7. Ф у н к ц и и. Оператор функции по виду не отличается от оператора процедуры. Отличается он по использованию в программе и по виду начала описания.

В описании функции перед **procedure** пишется тип: **Integer**, **real** или **Boolean**. И это уже является описанием простой переменной с именем функции. Ни в программе, ни в теле процедуры-функции описывать ее уже больше не надо. И все же имя функции — переменная особая. Присваивать ей значения можно лишь в теле описания ее процедуры. Заметим, что функция, подобно процедуре, может быть с параметрами или без них.

Оператор функции (с соответствующим набором параметров) может употребляться в арифметических и булевских выражениях (см. табл. 2) так же, как переменная. Если переменная арифметическая, то в описании функции перед **procedure** должен быть тип **integer** или **real**. Если переменная булевская — то **Boolean**. Тем самым мы (в последний раз) расширили понятие выражения.

Когда в некотором выражении встречается функция, то программа находит соответствующее описание. В теле описания вычисляется значение функции, и программа использует полученное значение.

Поэтому в теле функции должно выполняться одно (или несколько) присваивание определенных значений имени функции. В теле функции ее имя пока разрешим писать только в качестве переменной, которой присваивается значение. Иначе это вызовет рекурсивное обращение к описанию функции (см. 14.8).

Рассмотрим описание функции без параметров

real procedure r ;

{real $r1$; $r1 := x \uparrow 2 + y \uparrow 2$; $r := A/r1$ }

Имея его, можно писать оператор

$u := r/(1 + r \uparrow 2) + A \times \text{sqrt}(r)$

если только u описано как **integer** или **real**, а переменных x , y , A уже имеют значения. Заметьте, что тело этой процедуры нельзя писать так:

$$\{r := x \uparrow 2 + y \uparrow 2; r := A/r\}$$

ибо во втором операторе присваивания имя процедуры встретится в правой части. Как пример вычислений эти программы составлены неразумно, но нам важно пояснить, как используются функции.

Описание функции для суммирования

$$S = a[m] + \dots + a[n]$$

где a — имя массива, можно приготовить таким:

```

real procedure  $S(i, m, n, a);$ 
begin real  $s1; s1 := 0;$ 
  for  $i := m$  step 1 until  $n$  do  $s1 := s1 + a; S := s1$ 
end

```

И здесь по указанной причине нельзя собирать сумму сразу в S . Использоваться функция S может, например, в операторе

$$u := \exp(S(i, 1, 10, 1/i \uparrow 3) + C \times S(i, 1, 10, x[i] \times y[i]))$$

Оператор-функцию можно употреблять в программе и как оператор процедуры, но вычисленное значение ее имени при этом потеряется.

14.8. Рекурсии процедур. Модифицированное тело из описания процедуры может само содержать операторы процедур (как иных, так и той же самой). При выполнении модифицированного тела эти операторы будут выполняться по обычным правилам для операторов процедур. Таким образом, модифицированное тело само обратится к описаниям соответствующих процедур. Если при этом повторно встретится та же самая (еще неоконченная) процедура, то все произойдет так, как будто мы обратились к другому экземпляру (копии) ее описания.

Рекурсии в процедурах возникают или из-за вида оператора процедуры, или из-за структуры самого описания процедуры.

14.8.1. Рекурсивное использование процедур. Рассмотрим только что приведенное описание функции для суммирования

$$S = a[m] + \dots + a[n].$$

Если в этом же блоке встретится оператор

$$U := S(k, 2, 11, k \uparrow 2)$$

то будет вычислено $U := 2^2 + 3^2 + \dots + 11^2$. Но если употребить оператор

$$U1 := S(j, 1, 8, S(k, 2, 11, \text{sqrt}(j \uparrow 2 + k \uparrow 2)))$$

то описание S рекурсивно обратится само к себе и будет вычислена двойная сумма

$$U1 = \sum_{j=1}^8 \sum_{k=2}^{11} \sqrt{j^2 + k^2}$$

Рекурсивные обращения к процедурам вызывают и всякого рода операторы типа

$$U2 := \cos(n \times \sin(m \times \cos(x)))$$

14.8.2. Рекурсивные процедуры. Простой пример дает рекурсивное вычисление факториала. Вот описание этой функции:

integer procedure factorial (n);

if $n > 1$ then

factorial: = $n \times \text{factorial}(n - 1)$ else 1

Если теперь употребить оператор функции

$u := \text{factorial}(k)$

с целым значением k , то будет получено $u = k!$. Рассмотрим подробнее, как это произойдет. Если $k = 1$, то модифицированное тело присвоит имени $\langle \text{factorial} \rangle$ значение 1.

Если $k = 2$, то модифицированное тело

if $k > 1$ then

factorial: = $k \times \text{factorial}(k - 1)$ else 1

снова обратится к описанию функции *factorial*. Все произойдет так, как будто обращение произошло ко второму экземпляру этого описания. В результате

образуется второе модифицированное тело

if $k - 1 > 1$ then

$factorial := (k - 1) \times factorial (k - 2)$ else 1

При его выполнении имя $\langle factorial \rangle$ получит значение 1. Это значение будет подставлено в первое модифицированное тело вместо $\langle factorial (k - 1) \rangle$ и даст окончательно: $factorial := 2$. При больших значениях k рекурсивных обращений будет больше.

Конечно, транслятор не обязан фактически образовывать копии процедур — он может обойтись и одним экземпляром, специально составленным и дающим тот же результат.

Рассмотрим более трудный пример. Пусть $t [1], t [2], \dots, t [n]$ — десятичные цифры и уже имеется процедура $F (t, n)$, печатающая число $\{t [1] t [2] \dots t [n]\}$, изображенное последовательностью этих цифр. Тогда ниже следующая программа (точками обозначено тело процедуры F) отпечатает все целые числа от 000 до 999:

begin integer $n; n := 3;$

begin integer array $t [1 : n];$ procedure $F (t, n); \dots;$

procedure $count (i);$

for $t [i] := 0$ step 1 until 9 do if $i < n$

then $count (i + 1)$ else $F (t, n); count (1)$

end

end

Не обращая внимания на простоту задачи, тщательно проделайте вручную эту программу, выписывая все модифицированные тела. Здесь хорошо видны возможности рекурсивных процедур.

14.9. Спецификации. Аргументы в описаниях процедур (т. е. формальные параметры) не имеют описаний. Нужные сведения о них возникают автоматически после замены формальных параметров фактическими. Но чтобы облегчить работу транслятора, можно заранее указать характер некоторых формальных параметров. Для этого служат спецификации (см. табл. 3). Спецификации похожи на описания. Но видов здесь больше: в спецификациях, например, есть иероглиф **label**, а в описаниях его нет, ибо метка не имеет явного описания. Кроме того, в спецификациях массивов не указываются граничные пары. В спецификациях процедур и

функций не указываются формальные параметры (т. е. пишется лишь имя процедуры или функции).

Спецификации формальных параметров должны соответствовать типам фактических параметров.

Вот описание функции суммирования вместе со спецификациями:

```
real procedure S (i, m, n, f); integer i, m, n; real f;  
begin real s1; s1 := 0;  
  for i := m step 1 until n do s1 := s1 + f; S := s1  
end
```

Обозначение *S* помещать в спецификации не надо: его тип уже указан в начале описания функции. Спецификации ограничивают возможности оператора процедуры. Так, например, приведенное описание процедуры суммирования непригодно, если мы хотим в операторе процедуры использовать *f* как функцию (имеющую свое собственное описание). Для этого случая спецификацию «**real f**» нужно заменить на «**real procedure f**».

14.10. Значения. В описании процедуры (см. табл. 3) может находиться перечень формальных параметров, объявленных <значениями>; параметры-значения обязательно должны иметь спецификации.

Список значений изменяет правила для выполнения оператора процедуры следующим образом:

1) Создается блок, охватывающий тело.

2) В начале этого блока описываются обозначения формальных параметров, попавших в список значений. Этим обозначениям присваиваются значения соответствующих фактических параметров. Такие присваивания выполняются однажды, перед входом в тело.

3) При модификации тела формальные параметры, попавшие в список значений, сохраняют свои обозначения (т. е. не заменяются на фактические параметры).

В список значений можно заносить только такие формальные параметры, у которых фактические параметры вообще способны принимать значения, т. е. являются выражениями или массивами.

Значением массива считается соответствующая последовательность арифметических или логических чисел. Значением именуемого выражения считается метка.

Не принимает значений, например, имя переключателя или процедуры (не функции).

Помещение формального параметра в список значений имеет следующий смысл:

1) Ускоряется работа оператора процедуры в том случае, когда соответствующий фактический параметр является сложным выражением.

2) «Защищается» соответствующий фактический параметр; можно быть уверенным, что он не изменится при выполнении оператора процедуры.

3) Допускается использование формального параметра в качестве внутренней «работающей» переменной в теле процедуры.

Обычно в список значений заносят «входные» параметры процедуры, которые целесообразно вычислить заранее.

Проиллюстрируем применение значений:

```
procedure E(A, B, C); value A, C;  
real A, B; integer C;  
begin integer k; A := A ↑ 2; B := 0;  
  for k := 1 step 1 until C do B := B + A × (B + 1)  
end
```

Если к этому описанию обращаются оператором

$$E(a, b, n \times (n - 1)/2)$$

то можно опустить $\langle \text{value } A, C \rangle$. Однако в этом случае значение переменной a изменится, а выражение $n(n - 1)/2$ будет вычисляться на каждом шаге (ибо при каждом увеличении k должно проверяться неравенство $k \leq n \times (n - 1)/2$, правая часть которого, вообще говоря, могла бы зависеть от k или B).

Но если к описанию E обращаются оператором

$$E(u \uparrow v, b, c)$$

то $\langle \text{value } A \rangle$ необходимо — иначе в модифицированном теле возникнет недопустимая конструкция

$$u \uparrow v := (u \uparrow v) \uparrow 2$$

14.11. Собственные объекты. Чтобы закончить уточнение правил для выполнения оператора, процедуры, надо сказать, как будет пониматься объект, описанный в теле как **own**. Будет он принадлежать описанию процедуры и сможет меняться при каждом обращении к описанию или будет свой для каждого оператора

процедуры и будет сохраняться, пока «его» оператор не обратится к описанию? Этот вопрос в алголе-60 пока еще не уточнен.

14.12. Побочный эффект. Оператор процедуры или функции обращается к своему описанию, которое в конечном счете выполняется как подпрограмма. Вполне естественно, что эта подпрограмма может менять значения различных переменных.

Так, например, описание функции

real procedure $y(x); y := x := 1 + x$

не только вычисляет $y := 1 + x$ но и изменяет аргумент. Поэтому значения x в операторах

1) $s := x + y(x)$

2) $s := y(x) + x$

оказываются различными. Программисты к этому привыкли, но новичку такая некоммутативность кажется дикой.

Еще «коварнее» описание функции

real procedure $r; \{r = \text{sqrt}(x \uparrow 2 + y \uparrow 2); a := x \times y\}$

Здесь при выполнении операторов

1) $s := r + a$

2) $s := a + r$

изменение a происходит как бы без ведома «потребителя». Нужно учитывать, что r — не простая переменная, а функция и при ее вычислении может смениться значение a .

14.13. В связи с возможной некоммутативностью арифметических действий в алголе принят следующий жесткий порядок. При вычислении арифметических выражений всегда выполняется первое (считая слева направо) действие (в том числе и вычисление функции), которое можно выполнить в данный момент в силу расстановки скобок и старшинства действий. Так, например, порядок вычисления

$$a + b + c \times d \uparrow c - f$$

можно описать следующим образом:

$Ra := a; R1 := Ra + b;$

$Rc := c; Rd := d;$

$R2 := Rd \uparrow Rc; R3 := Rc \times R2;$

$R4 := R1 + R3; R5 := R4 - f;$

Здесь оператор $Ra := a$ запоминает (а если a — функция, то предварительно вычисляет) значение a на случай, когда вычисление b (если это функция) может изменить значение a . С той же целью выполняются присваивания

$$Rc := c \quad \text{и} \quad Rd := d.$$

Аналогичные правила употребляются при вычислении булевских и именующих выражений.

14.14. Тело процедуры, согласно определению (табл. 3), может быть не только оператором (этот случай мы уже разобрали), но и *кодом*, т. е. программой, написанной не на алголе. Это дает возможность написать тело непосредственно в командах машины особо тщательно, поместить в ячейку несколько чисел и т. д. Правила связи алгольного текста с *кодом* остаются на совести составителей трансляторов.

§ 15. Пояснительный текст

15.1. Варианты запятой. В списке фактических или формальных параметров эти параметры можно разделить не только запятыми, но и поясняющими конструкциями вида (см. табл. 1 и 3)

) последовательность букв: (

Так, например, для функции $P = R \times T/v$ оператор

$$P(R, T, v)$$

можно написать еще и в любом из трех видов:

1) $P(R), \text{temperature: } (T, v)$

2) $P(R, T) \text{ volume: } (v)$

3) $P(R) \text{ temperature: } (T) \text{ volume: } (v)$

Те же (или иные) пояснения можно сделать и в описании этой функции, т. е. в списке формальных параметров.

15.2. Строки. Для печати текста используются <строки> (см. табл. 3), а оператор вывода строит так, чтобы написанное в строчных кавычках ' ' печаталось

буквально. Применение этого показывает программа

```
{integer a; a := 3; output ('a = ', a)}
```

которая напечатает $a = 3$, в отличие от программы

```
{integer a; a := 3; output (a)}
```

которая напечатает только число 3.

Строка может быть фактически параметром процедуры. В теле она заменит формальный параметр. Если модифицированное тело тоже содержит оператор процедуры, то строка может «перекочевать» в тело и этого оператора. Но в конечном счете строка должна попасть в оператор вывода или в код. Внутри строки может применяться значок `—` для пробела печати.

15.3. П р и м е ч а н и я. В программу алгола можно вставлять примечания — пояснительный текст, пользуясь следующими правилами:

КОНСТРУКЦИЯ ПРИМЕЧАНИЯ

ЕЕ ЭКВИВАЛЕНТ

- 1) **;** **comment** <последовательность символов > ;
до ближайшего знака; включительно>
- 2) **{comment** <последовательность символов > {
до ближайшего знака; включительно>
- 3) **}** <последовательность символов > }
до ближайшего знака; или } или
else не включая этого знака >

Здесь **comment** — еще один иероглиф алгола, а замена примечаний «эквивалентами» выполняется в порядке следования, слева направо (если только примечание не стоит внутри <строки>).

Таким образом, после знака **}** можно сразу писать примечание, а после знака **;** или **{** надо сперва помещать **comment**, а потом уже примечание. Внутри примечания нельзя пользоваться знаком **;** а иногда и знаками **}** и **else**. Наконец, «примечание», написанное внутри <строки>, считается частью строки, а не <примечанием>.

С помощью примечания после **end** удобно указать, что именно окончено. Так, например, вычисление

$$PS = \prod_{i=1}^n \sum_{j=1}^n a_{ij} \text{ удобно написать с комментариями:}$$

```
real procedure PS (a, n); value n;
integer n; array a;
begin integer i, j; real s1, s2; s1 := 1;
  for i := 1 step 1 until n do
    begin s2 := 0; for j := 1 step 1 until n do
      s2 := s2 + a [i, j]; s1 := s1 × s2
    end i; PS := s1
end PS;
```

Процедуры алгола регулярно публикуются в журналах и сборниках. Приведем несколько примеров:

1) Процедура *cheb* вычисляет полином Чебышева для заданных x и n и по рекуррентной формуле

$$T_0(x) = 1; T_1(x) = x; T_{n+2}(x) = 2xT_{n+1} - T_n.$$

```
real procedure cheb (n, x);
value n, x; integer n; real x;
begin integer i; real a, b, c;
  a := 1; b := x;
  if n = 0 then c := 1 else if n = 1 then c := x
  else for i := 2 step 1 until n do
    {c := 2 × x × b - a; a := b; b := c}; cheb := c
end cheb;
```

2) Процедура *max* находит строку i и столбец j , где расположен максимальный по модулю элемент матрицы a порядка $n \times n$.

```
procedure max (a, n, i, j); value n;
integer n, i, j; array a;
begin integer i1, j1; real t;
  i := j := 1; t := abs (a [1, 1]);
  for i1 := 1 step 1 until n do
    for j1 := 1 step 1 until n do
      if abs (a [i1, j1]) > t then {i := i1; j := j1; t := abs (a[i, j])}
    end;
end;
```

3) Нахождение корней функции методом деления интервала пополам. Процедура *bisec 1* вычисляет значения функции на концах заданного интервала (a, b) . Если знаки этих значений совпадают, то процедура *bisec 1* отправляет к метке *s1*. Если знаки разные, то процедура вычисляет значение функции в середине интервала, выбирает полуинтервал, где функция меняет знак, и т. д. Процесс прекращается в трех случаях: (1) когда длина интервала станет меньше *eps 1*; (2) когда вычисленное значение функции окажется по модулю меньше *eps*; (3) когда интервал перестанет убывать (в результате исчерпания точности счета машины). В первых двух случаях процедура находит корень функции. В третьем случае она отправляет к метке *s2*.

Чтобы процедура *bisec 1* работала, надо, разумеется, чтобы программа содержала требующуюся процедуру-

функцию. Хотя метод *bisec* 1 относится к самым медленным, он применим к любой непрерывной функции. Это делает процедуру очень надежной.

```

procedure bisec 1 (a, b, eps, eps1, f, s1, s2) result : (x);
value a, b, eps, eps1;
real a, b, eps, eps1, x; real procedure f; label s1, s2;
begin real y, z, t, t1;
  procedure fun (y); real y;
  {y := f (x); if abs (y) ≤ eps then go to fin};
  x := a; fun (y); x := b; fun (z);
  if sign (y) = sign (z) then go to s1; t := abs (b - a);
  iter : if t < eps1 then go to fin;
    t1 := t; x := (a + b)/2; fun (y);
    if sign (y) = sign (z) then b := x else a := x;
    t := abs (b - a);
    if t ≥ t1 then go to s2 else go to iter;
fin : end bisec 1;

```

4) Пусть уже описаны две процедуры-функции $y(x)$ и $\delta(x, \epsilon)$ такие, что $\delta(x, \epsilon) > 0$ при $\epsilon > 0$ и $|x_1 - x| < \delta(x, \epsilon)$ влечет $|y(x_1) - y(x)| < \epsilon$.

Составьте описание функции $\delta_2(x, \epsilon)$ такой, что $\delta_2(x, \epsilon) > 0$ при $\epsilon > 0$ и $|x_1 - x| < \delta_2(x, \epsilon)$ влечет $|y(x_1)^2 - y(x)^2| < \epsilon$.

Р е ш е н и е.

```

real procedure  $\delta_2$  (x, eps); real x, eps;  $\delta_2$  :=
 $\delta(x, (\text{if } \epsilon < 1 \text{ then } \epsilon \text{ else } 1)/(1 + 2 \times \text{abs}(y(x)))$ 

```

Действительно, полагая $y = y(x_1)$ и $y_0 = y(x)$, имеем

$$|y - y_0| < \frac{\min(\epsilon, 1)}{1 + 2|y_0|} \leq \frac{1}{1 + 2|y_0|} \leq 1,$$

откуда

$$|y^2 - y_0^2| < \frac{\epsilon}{1 + 2|y_0|} (1 + 2|y_0|) = \epsilon.$$

Читатель может заметить, что обычное ϵ -доказательство непрерывности квадрата непрерывной функции несколько не короче процедуры δ_2 .

ГЛАВА 4

ТУМАННОСТИ АЛГОЛА

§ 16. Подмножество алгола

В настоящее время выпущено *подмножество алгола*. Это — вариант алгола, облегченный для аппаратуры ввода — вывода и для составителей трансляторов. Всякая программа, написанная на *подмножестве*, является программой и на алголе, но не наоборот. Подавляющее число трансляторов удовлетворяет требованиям подмножества, но редко покрывает полностью алгол. Поэтому не стоит без нужды нарушать требования подмножества.

Основные отличия *подмножества* от алгола в следующем:

1. Исключены большие буквы *A, B, ...*
2. Имена различаются по первым шести символам (если у двух имен эти символы совпадут, то имена считаются одинаковыми).
3. Исключено деление нацело ($\div$).
4. Возведение целого основания в целую степень определено только для целого показателя без знака.
5. Параметр цикла должен быть простой переменной (без индексов).
6. Именуемые выражения разрешаются лишь безусловные и не заключенные в скобки.
7. Переключательный список должен состоять только из меток.
8. Запрещены целые в качестве меток.
9. Исключено понятие *own*.
10. Запрещены рекурсивные процедуры и рекурсивное использование процедур.
11. Запрещены описания функций, вызывающие побочный эффект.

12. Все формальные параметры должны иметь спецификации.
13. Имена не должны появляться раньше своих описаний (в алголе имя глобального параметра тела может встретиться раньше, чем описание этого имени).
14. Фактические параметры, вызываемые по наименованию (т. е. не содержащиеся в списке *value*), могут быть только именами или строками (в алголе они, кроме того, могут быть переменными с индексами и выражениями).

§ 17. Туманности алгола

Этот параграф в первом чтении должен быть, безусловно, опущен. Средство программирования — алгол, — будучи доведено до уровня *языка*, обрело возможность самостоятельного развития. Сочетания слов алгола могут образовывать грамотные фразы, смысл которых не предусмотрен авторами языка. Иногда это порождает неясности (подобные двусмысленностям человеческих языков), иногда придает фразам смысл, противный идеям авторов алгола. Перейдем к изложению.

1. Н е о п р е д е л е н н ы й о п е р а т о р . Начнем с синтаксически грамотного примера:

```
begin integer i, k; array a [1 : 1];
  a [1] := i := k := 10; i := a [k]; ...
```

При фактическом выполнении программы встретится оператор $i := a[10]$. Элемент $a[10]$, разумеется, не определен. Вслед за ним не определен и оператор $i := a[10]$. Что должна делать машина при выполнении такого оператора, в алголе оставлено неопределенным.

П р и т р а н с л я ц и и — переделке алгольной программы в машинную — ошибка не обнаружится, ибо еще не известно, чему будет равно k в момент выполнения оператора $i := a[k]$. Некоторые трансляторы создают медленные программы, всякий раз проверяющие, находится ли индексное выражение в предписанных границах. Такие трансляторы обнаружат ошибку в операторе $i := a[k]$ и сообщат о ней. Но большинство трансляторов отсчитает 10 ячеек памяти машины (считая $a[1]$ первой) и занесет в ячейку i информацию из десятой ячейки.

Перейдем к ошибке с более тяжелыми последствиями.

```
begin integer i, k; array a[1:1];  
  i := k := a[1] := 10; a[i] := k; ...
```

При выполнении этой программы многие трансляторы отсчитают 10 ячеек памяти (считая ячейку для $a[1]$ первой) и занесут в десятую ячейку (назовем ее R) число десять. Если в ячейке R находилась команда, которую еще предстоит выполнить, то это приведет к порче программы с необозримыми последствиями.

2. В ныне действующем «Официальном сообщении» сказано, что глобальные параметры процедуры понимаются в том смысле, который действует на месте расположения описания (а не оператора) процедуры. Пусть, например, блок M содержит подблоки $M1, M2, \dots$:

```
M: begin ...  
  M1: begin ... end;  
  M2: begin ... end;  
  . . . . .  
end
```

И пусть каждый подблок содержит описание переменных x и y и процедуру без параметров $r = \sqrt{x^2 + y^2}$, т. е.

```
M1: begin real x, y;  
  real procedure r; r := sqrt(x ↑ 2 + y ↑ 2); ...  
end
```

В таком случае описания процедур r нельзя «вынести» из подблоков и сделать один раз в блоке M . Если так поступить, то в лучшем случае переменные x, y не будут описаны в блоке M и транслятор зафиксирует ошибку. В худшем — использование величины r в подблоке приведет к неверному результату (так как x и y будут взяты не из этого подблока).

3. Много неясностей алгола связано с побочным эффектом. При употреблении оператора процедуры-функции происходит обращение к ее описанию, которое вычисляет значение функции. Но при этом модифицированное тело может изменить значения еще каких-либо переменных (и это называется побочным эффектом).

1) Рассмотрим условный оператор

if b then Q

По точному смыслу «Официального сообщения» этот оператор эквивалентен оператору Q или пустому оператору, когда b имеет значение **true** или (соответственно) **false**. Пусть, однако, b оказалось функцией. Например,

begin integer i ;

Boolean procedure b ; { $i := 1$; $b := \text{true}$ };

$i := 0$; if b then Q ; ...

При выполнении этой программы сначала i станет равно нулю. Затем, во время выполнения проверки $\langle \text{if } b \rangle$, величина i станет равна единице. Некоторые трансляторы по окончании проверки снова сделают i нулем (в точном соответствии с правилами алгола). Но большинство трансляторов оставят i единицей.

2) Рассмотрим другой пример:

begin Boolean procedure b ; { $i := 1$; $b := \text{true}$ };

$i := 0$; if $\text{false} \wedge b$ then Q ; ...

При проверке условия $(\text{false} \wedge b)$ некоторые трансляторы сразу же (по числу **false**) замечают, что условие не выполнено, не вычисляя величины b и оставляют i нулем. Но они сменяют i , если написать условный оператор в виде

$i := b \wedge \text{false}$ then Q

3) Рассмотрим оператор цикла с элементом арифметического типа

for $u := a$ step b until c do Q

По точному смыслу алгола он эквивалентен операторам

$u := a$;

$M : \text{if } (c - u) \times b \geq 0 \text{ then } \{ Q; u := u + b; \text{go to } M \}$ (*)

На каждом цикле кроме (после) оператора Q вычисляются значения $u + b$, $c - u$, b . В результате c вычисляется однажды, а b и u — по два раза. Если же u — переменная с индексами:

for $u[i] := a$ step b until c do Q

то индекс u параметра цикла (а этот индекс может быть выражением и даже функцией) вычисляется трижды: два раза в операторе $u[i] := u[i] + b$ и еще раз в разности $c - u[i]$.

Так, например, после выполнения программы

```
begin integer k; integer array U, A, C [1 : 1];  
  integer procedure i; {i := 1; k := k + 1};  
  k := 0; A [1] := 1; C [1] := 3;  
  for U [i] := A [i] step A [i] until C [i] do;  
    output (k)  
end
```

будет отпечатано значение k , равное 23 (проверить!).

Двукратные пересчитывания параметра цикла и шага производятся на тот случай, когда вычисление какой-либо из этих величин окажется связано с функцией-процедурой и сможет, благодаря побочному эффекту, повлиять на значения других величин. Чтобы ускорить выполнение цикла, в некоторых трансляторах цикл объявлен эквивалентным не операторам (*), а операторам

```
u := ut := a; bt := b;  
M : i      f (c - ut) × bt ≥ 0 then  
  {Q; bt := b; u := ut := u + bt; go to M }
```

Аналогично обстоит дело и с элементом типа пересчета.

4) Предложим себе написать процедуру *trans* (x, y) с целыми параметрами x, y , которая (при обращении к ней) переменит значения x и y . Точнее, пусть a и b — значения переменных u, v . Мы хотим, чтобы после выполнения оператора *trans* (u, v) переменная u получила значение b , а переменная v — значение a .

Я не встречал людей, которые бы не начинали со следующего неверного решения *):

```
procedure trans (x, y); integer x, y;  
{integer t; t := x; x := y; y := t}
```

Чтобы убедиться в ошибочности решения, рассмотрим программу с этой процедурой

```
begin integer i; integer array a [1 : 2];  
  procedure trans (x, y); integer x, y;  
  {integer t; t := x; x := y; y := t};  
  i := 1; a [1] := 2; a [2] := 3; trans (i, a [i])  
end
```

*) За исключением тех, которые вовсе отказывались решать задачу.

Перед обращением к оператору *trans* (*i*, *a* [*i*]) величина *i* равнялась единице, и по смыслу задачи оператор должен обменять значения *i* = 1 и *a* [*i*] = 2. Образует модифицированное тело

$$\{\text{integer } t; t := i; i := a[i]; a[i] := t\}$$

При его выполнении последовательно будет сделано

$$\begin{aligned} t &:= 1 \quad (\text{ибо } i \text{ равно } 1) \\ i &:= 2 \quad (\text{ибо } a[i] \text{ есть } a[1] \text{ и равно } 2) \\ a[2] &:= 1 \quad (\text{ибо } a[i] \text{ теперь уже есть } a[2]) \end{aligned}$$

Итак, вместо того, чтобы положить *i* := 2 и *a* [*i*] := 1, оператор сделал *i* := 2 и *a* [*i*] := 1;

Ничего не дают и попытки исправить тело процедуры различными проверками того, какой параметр изменился (ибо при этом могут возникнуть неопределенные операторы).

Верное решение достигается с помощью побочного эффекта:

```
pro cedure trans (x, y); integer x, y;
begin integer procedure f;
  {f := y; y := x};
  x := f
end
```

4. Ряд неясностей алгола порожден понятием *собственный*. Пусть, например, переменная *a* описана как **own real** *a* внутри тела процедуры. Для каких блоков переменная *a* является «собственной»: для тела процедуры или для блоков, содержащих операторы процедуры? Это не уточнено. В первом толковании величина *a* будет одна. При втором толковании величин *a* может образоваться столько же, сколько есть операторов данной процедуры. И значение каждой величины должно будет сохраняться на случай повторного выполнения «ее» оператора. Описанная картина может усложниться рекурсиями.

5. При записи и публикации алгоритмов на алголе бытует ряд сокращений. Пользоваться ими не обязательно, но нужно уметь их читать.

Элемент цикла *A* **step** *B* **until** *C* на практике часто бывает таким: **1 step 1 until** *C*. Поэтому некоторые пишут

сокращенно:

полная	форма	сокращенная
1 step B until C		step B until C
A step 1 until C		A until C
1 step 1 until C		until C

Вот несколько примеров сокращенной записи:

- 1) for $i :=$ until n do Q
- 2) for $i := m$ until n do Q
- 3) for $a[k] :=$ until n , A while V , step B until C do Q

Во всех этих случаях сохраняется часть $\langle \text{for } i := \rangle$, а элемент типа прогрессии узнается по иероглифу **until**.

При выполнении элемента цикла A while V иногда оказывается не нужно присваивать значение A параметру цикла. И тогда некоторые опускают выражение A . Если оператор цикла состоит только из таких элементов, то опускают и $\langle \text{for } i := \rangle$. Таким образом:

вместо $\text{for } i := A \text{ while } V \text{ do } P$ пишут $\text{while } V \text{ do } P$

6. Используя алгол для записи алгоритмов, иногда разрешают себе значительные отступления от официальных правил (за счет действий с частями слова, употребления машинных команд логического сложения и умножения, засылки передач управления и т. д.).

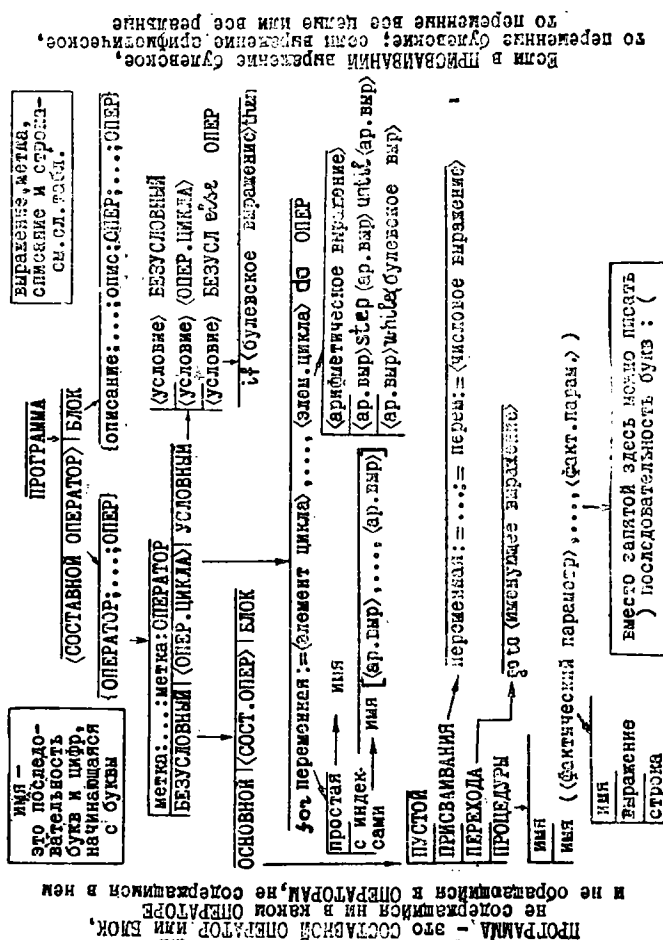
7. Если транслятор не работает с русскими буквами, а программист хочет написать русское слово (например, метку или имя), то для единообразия русские буквы заменяют латинскими по системе, принятой на телеграфе:

а a	и i	с s	щ sc
б b	к k	т t	ъ y
в v	л l	у u	ь y
г g	м m	ф f	ы i
д d	н n	х h	э e
е e	о o	ц c	й j
ж g	п p	ч ch	ю ju
з z	р r	ш sh	я ja

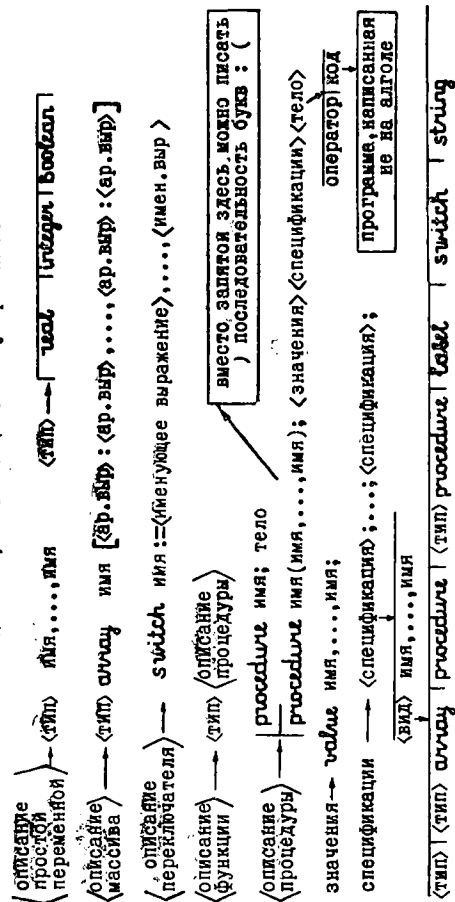
8. Процедуры *ввода* — *вывода* и процедура *код* (прежде всего включение в алгольную программу машинных команд) в алголе пока не определены. Поэтому необходимо узнать, как они введены в данном вам трансляторе (или данным вам экзаменатором).

СИНТАКСИЧЕСКИЕ ТАБЛИЦЫ

ТАБЛИЦА 1



Все имена (кроме меток и аргументов в описаниях процедур) должны быть описаны в начале блоков, где они употребляются. Перед описанием можно дополнительно поставить ошл. Обозначение метки считается описанным в начале минимального блока, заключенного метку-приемник.



Список знаков алгола

Большие и малые латинские буквы

Малые русские буквы (в русском варианте алгола)

Десятичные цифры, десятичная точка, опущенная десятка

Булевские числа **true** и **false**

Разделительные знаки , ; := —

Скобки () [] { } , ; := —

Вместо скобок { } можно писать иероглифы **begin end**

Знаки арифметических действий + — × / ÷ ↑

Знаки сравнений < ≤ = ≠ ≥ >

Знаки булевских действий ≡ ⊃ ∨ ∧ ⊔

Двадцать четыре иероглифа в русской и английской записи:

if (иф)	если	true (труэ)	да
then (дээн)	то	false (фолс)	нет
go to (гоу ту)	иди	switch (свитч)	ключ
else (элз)	иначе	procedure (просидюр)	процедура
for (фо)	для	begin (бэгин)	начало
step (степ)	шаг	end (энд)	конец
until (антил)	до	label (лэйбл)	метка
while (вайл)	пока	string (стринг)	строка
integer (интеджэр)	целый	do (ду)	цикл
real (рил)	реальный	value (вэлью)	значение
Boolean (Булен)	булевский	own (оун)	собственный
array (эррей)	массив	comment (коммент)	комментарий

Структура примечаний

КОНСТРУКЦИЯ ПРИМЕЧАНИЯ

ЕЕ ЭКВИВАЛЕНТ

- | | |
|---|---|
| 1) ; comment <последовательность символов до ближайшего знака; включительно> | ; |
| 2) { comment <последовательность символов до ближайшего знака; включительно> | { |
| 3) } <последовательность символов до ближайшего знака ; или } или else не включая этого знака> | } |

Замена примечаний «эквивалентами» выполняется в порядке следования, слева направо. Примечание, находящееся внутри строчных кавычек, не заменяется «эквивалентом».

ОГЛАВЛЕНИЕ

Предисловие	3
Г л а в а 1. Элементарная часть алгола	5
§ 1. Знаки и слова алгола	5
§ 2. Описание синтаксических таблиц	9
§ 3. Простейшие арифметические выражения	10
§ 4. Простейшие операторы	13
§ 5. Условные выражения	19
§ 6. Условные и составные операторы	23
§ 7. Переменная с индексами	26
§ 8. Оператор цикла	27
§ 9. Стандартные функции и процедуры.	34
Г л а в а 2. Описания	33
§ 10. Описания переменных	33
§ 11. Влияние описаний на выполнение операторов и структуру выражений	39
§ 12. Область действия описаний	41
§ 13. Переключатели	43
Г л а в а 3. Процедуры	49
§ 14. Процедуры	49
Определение. Отыскание описаний. Образование и выполнение модифицированного тела. Процедуры без параметров. Функции. Рекурсии. Спецификации. Значения. Собственные объекты. Побочный эффект. Порядок действий. Код.	
§ 15. Пояснительный текст	65
Г л а в а 4. Туманности алгола	69
§ 16. Подмножество алгола	69
§ 17. Туманности алгола	70
Синтаксические таблицы	76

Цена 24 к.

