4. Раздел операторов

Действуй, пока никто не успел тебе запретить.

Эрих Мария Ремарк.
«Время жить и время умирать»

В стандарте Паскаля все действия над данными производят только операторы из раздела операторов.[1] При описании раздела операторов мы будем предполагать, что метки и операторы перехода использоваться *не будут*, тем более, что во многих современных языках их уже нет совсем. Примечательно, что синтаксис, семантика и прагматика операторов при этом *существенно* упрощается.

<раздел операторов> ::= **begin** <оператор>{;<оператор>}... **end**

Как видим, в разделе не менее одного оператора, а если их два или больше, то они *разделяются* друг от друга точкой с запятой. Обратите внимание, что в Паскале (по сравнению с некоторыми другими языками) операторы не заканчиваются точкой с запятой, а отделяются точкой с запятой друг от друга. Например:

```
begin end     { Один (пустой) оператор }
begin ; end   { Два (пустых) оператора }
```

А вот, например, язык Фортран требует, чтобы точки с запятой разделяли операторы, но только *в пределах одной строки*, например:

```
x=1; y=3.14; z=2
y=x+y; z=x-z
```

Семантика раздела операторов (без меток!) очень проста, исполнитель по одному разу выполняет все операторы раздела, программа заканчивается, когда выполнен последний оператор (перед **end**.). Таким образом, специальный оператор конца программы (**halt**, **exit** и т.д.) не нужен.[2] Раздел операторов реализует структурность алгоритма, а сами операторы являются шагами этого алгоритма.

Все операторы делятся на простые и сложные. Простые операторы не содержат внутри себя других операторов, а сложные операторы, наоборот, включают внутрь себя хотя бы один оператор. Если образно сравнить операторы с ящиками, то простые являются сплошными и «не открываются», а сложные содержат внутри ещё хотя бы одни ящик-оператор. В стандарте Паскаля есть четыре простых и четыре сложных оператора.

4.1. Простые операторы

Простота не предшествует сложности, а вытекает из нее 🙄.

Алан Перлис,
первый лауреат премии Тьюринга

Пустой оператор. Синтаксис пустого оператора совсем прост, это пустая строка символов. Семантика тоже простая, встретив пустой оператор, исполнитель «ничего не делает», и переходит к следующему оператору. Труднее понять прагматику пустого оператора, ведь на первый взгляд от совершенно бесполезен.[3] Здесь всё дело в том, что синтаксис Паскаля требует, чтобы в определённых местах программы обязательно стоял (один) оператор. И вот, когда в каком-то месте программы нам ничего делать не надо, но требуется, чтобы там стоял оператор, то здесь нам и пригодится именно пустой оператор.

---

[1] Для более сложных языков это уже не так. Например, при порождении переменных в разделе переменных могут автоматически выполняться специальные процедуры-конструкторы, которые производят как угодно сложную обработку данных. Аналогично при выходе из блока, где порождены такие переменные, могут работать сложные процедуры-деструктоы.

[2] В языке Free Pascal есть оператор `exit`, это «замаскиро́ванный» **goto**, переходящий на последний **end**.

[3] Если бы мы использовали метки, то на помеченный пустой оператор можно было бы хотя бы сделать переход **goto** из другого места программы.

Оператор перехода.

Среди многих плохих идей, представленных на доске позора, идея оператора **`goto`** *подвергалась наибольшей критике. В языках программирования этот оператор является непосредственным двойником машинной инструкции перехода и может использоваться для конструирования условных и повторяющихся операторов. Но он также даёт возможность программистам конструировать запутанный или беспорядочный поток выполнения программы, игнорировать какую-либо регулярную структуру. Это затрудняет, если не делает невозможными, структурные рассуждения о таких программах.*

Никлаус Вирт
«Хорошие идеи: взгляд из Зазеркалья»

У этого оператора простой синтаксис: **`goto`** <метка>. Мы договорились, что эти операторы мы использовать не будем. Кроме того, во многих языках программирования есть «замаскиро́ванные» операторы перехода типа `break`, `exit` и `continue`, мы без них тоже обойдёмся.

Оператор присва́ивания. Это самый распространённый оператор языков программирования,[1] его синтаксис:

```
<оператор присваивания> ::= <левая часть>:=<выражение>
<левая часть> ::= <переменная> | <имя функции>
```

В языке Free Pascal при включённом режиме `{$COperators ON}` возможны некоторые операторы присваивания в стиле языка C, например

```
x+=y  вместо  x:=x+y
x-=x  вместо  x:=0
x*=y  вместо  x:=x*y  и т.д.,
      но всё же не разрешены  x++,  --y  и т.д.
```

Разрешёны и некоторые «странные» операторы, например, `x:=---y` или `x:=x-+-y`, как они выполняется мы рассмотрим позже, когда будем описывать приоритет операций в разд. 4.1.

В разных языках операция присваивания обозначается по-разному, например:

`x:=y`	Pascal, Ada
`x=y`	C, Fortran, Basic, Java, Python, PL/1
`x<-y`	APL (есть и «красивое» x←y)
`(SETQ x y)`	LISP
`x is y`	Prolog
`MOVE y TO x`	COBOL и т.д.

Сначала рассмотрим случай, когда выражение присваивается переменной, второй случай отложим до того, как будем изучать описание функций.

С понятием переменной мы уже начали знакомится. Правда, пока мы знаем только простые (скалярные) *именованные* переменные, например `X`, `My_perem` или `a123`, но постепенно мы изучим и сложные переменные (массивы, записи и т.д.). Пока учтите, что переменная в программе может иметь весьма сложное обозначение, например, вот каким может быть допустимый вид переменной:

`F(X↑<Y)↑↑.[g(i)+1]↑.Z↑`

Главное, что переменная имеет ссылку, её часто называют левым (left) адресом (l-value), она определяет место в памяти, куда надо направить присваиваемое значение, а само значение переменной называют правым (right) значением (r-value). Здесь главное понять, что r-value бесполезно получать, не зная l-value.

---

[1] Как мы уже упоминали, в функциональных языках программирования этого оператора чаще всего нет.

Теперь перейдём к выражению в правой части оператора присваивания. Как и в математике, в программировании выражение – это некоторая запись, которая позволяет вычислить значение этого выражения. Как Вы уже знаете, наши простые метаязыки не позволяют полностью описать синтаксис некоторых конструкций языка (в частности и выражений). Поэтому к описанию приходится прикладывать семантический фильтр (правило на естественном языке), с помощью которого можно отбраковывать неверно выводимые выражения.

Обычно в учебниках по стандарту Паскаля приводится одно такое правило, но при этом синтаксис выражения получается достаточно сложным. Мы пойдём по второму пути, приложив к описанию два семантических фильтра, но при этом синтаксис выражения существенно упроститься. Итак, синтаксис выражения:

<выражение> ::= [{+|-}]{**not**}...<терм>{<бинарная операция><терм>}...
<бинарная операция> ::= + | - | * | / | **div** | **mod** | **or** | **and** |
= | <> | > | < | >= | <= | **in**
<терм> ::= <переменная> | <константа> | <указатель функции> | (<выражение>)

Если присмотреться, то можно заметить, что выражение строится как цепочка термов (их ещё называют простыми выражениями), разделяемых бинарными операциями. Перед термом может стоять (а может и не стоять) знак унарной операции «плюс» или «минус», или знак операции логического отрицания **not** (один или несколько). Далее, как видим, если как угодно сложное выражение заключить в круглые скобки, то оно сразу волшебным образом становится простым. Термин «указатель функции» обозначает то значение, которая возвращает функция, например, при вычислении выражения на место `sin(x)` подставляется результат вычисления синуса от аргумента `x`.

Если, скажем, `a` и `b` – логические, а `x` и `y` – целые переменные, то будут выводиться правильные выражения:

```
+1-sin(3.2)    not a    not not(a or b)    (x+y)
```

Обратите внимание, что между **not** и `a` и между **not** и **not** необходим пробел, перевод строки или комментарий. В учебниках при написании метаформул это обычно явно не указано, чтобы не загромождать описание, но подразумевается. Конечно, здесь выводится много и неверных выражений, например

```
1-'A'    not x    sin('B')    ord(3.2)
```

Для фильтрации (отбрасывания) неверных выражений служит правило соответствия между операцией и операндом (операндами). Требуется, чтобы каждая операция либо применялась к допустимому типу операнда, либо тип этого операнда мог быть неявно преобразован в допустимый. Это правило легко проверяется программистом, так как он должен знать все операции, допустимые для каждого типа в выражении.

Вспомним теперь, как вычисляются выражения. Например, мы знаем, что при вычислении выражения `x+y*z` сначала выполняется операция умножения, как имеющая бóльшее старшинство (бóльший приоритет). В математике три уровня старшинства операций: самое старшее это возведение в степень, далее идут умножение и деление, а лишь потом сложение и вычитание. А как обстоит дело в языках программирования, где много операций (например, в стандарте Паскаля три унарные и 15 бинарных операций)?

Сколько уровней старшинства операций делать в языке программирования? Здесь можно пойти двумя взаимо исключающими путями. Во-первых, можно постараться, чтобы как можно больше часто используемых выражений можно было бы записывать без использования круглых скобок. К сожалению, по этому пути идёт большинство языков программирования, в результате у них получается много уровней старшинства. Например, в языке C и в языке Ассемблера MASM 13, а в языке C++ 17 (⚠) уровней старшинства операций. Запомнить их очень сложно, и программисты «на всякий случай» всё равно ставят круглые скобки, даже там, где они не нужны. Так, спрашивается, за что боролись? Во-вторых, можно уменьшить число уровней старшинства операций, но при этом иногда приходится ставить «лишние» круглые скобки.

С другой стороны, например, в старом языке APL вообще *один* уровень старшинства операций и все выражения вычисляются строго слева направо, так что там 3+4*5=35 ⚠. Естественно, в таком языке для изменения порядка вычисления выражений обязательно *нужны* скобки. Не надо думать, что это делает невозможным вычисление каких-то выражений без скобок. Известно, что любое вы-

ражение можно преобразовать в бесскобочную, так называемую прямую или обратную (префиксную или постфиксную) польскую запись. Например, выражение `(5+3)*(9-7)` можно записать как `*+5 3-9 7`, оно будет вычисляться так: `*` **`+ 5 3`** `- 9 7` → `*` `8` **`- 9 7`** → **`* 8 2`** → 16.

В стандарте Паскаля четыре уровня старшинства операций (в порядке убывания):

1. **not**
2. `*` `/` **div** **mod** **and**
3. `+` `-` **or**
4. `=` `<>` `>` `<` `>=` `<=` **in**

Обратите внимания, что в стандарте Паскаля унарные и бинарные знаки операций + и –, в отличие от многих других языков, имеют *одинаковое* старшинство.

В языке Free Pascal тоже четыре уровня старшинства операций, но приоритет немного другой:

1. **not** `@` `+` `-` { здесь *унарные* + и – }
2. `*` `/` **div** **mod** **and** **shl** **shr** `<<` `>>`
3. `+` `-` **or** **xor** `><` { здесь *бинарные* + и – }
4. `=` `<>` `>` `<` `>=` `<=` **in**

Пока незнакомые нам операции будут постепенно описываться по мере изучения языка. Сейчас лишь отметим, что *унарные* операции `+` и `-` «повышены в звании», они имеют такое же старшинство, как и операция **not**. Кроме того, *унарные* операции `+` и `-`, как и операция **not**, тоже право ассоциативные (т.е. выполняются справа налево, см. далее). Так сделано в большинстве языков.

В языке Free Pascal это приводит к интересным эффектам. В стандарте Паскаля запрещено писать два знака операции подряд, т.е. вместо `x+-y` надо писать `x+(-y)`. В языке Free Pascal это допускается, так как *унарные* операции `+` и `-`, в отличие от стандарта Паскаля, имеют бóльшее старшинство, чем *бинарные* операции `+`, `-`, `*` и `/`. Тогда, например, `x++-+y` будет эквивалентно `x+(+(-(+y)))` т.е. просто `x-y`, а `x*+-+y` эквивалентно `x*(+(-(+y)))` т.е. `x*(-y)`, а `x*+--y` это будет `x*y` Учтите, что это не имеет никакого отношения к языку C 🙂.

Как, например, записать условие принадлежности величины заданному диапазону `x∈[1..8]`? Запись `1<=x<=8` не проходит контроль типов, для записи `1<=x and x<=8`, снова не проходит контроль, так как сначала будет вычисляться ошибочное выражение `x and x`, поэтому приходится писать со скобками `(1<=x) and (x<=8)`.

Отметим, что, например, в Фортране аналогичная запись `1<=x .and. x<=8` и в языке C запись `1<=x && x<=8` будет *правильной*, так как там логические операции `.and.` и `&&` имеют *меньший* приоритет, чем операции сравнения. Так что, где найдёшь, где потеряешь 🙂. Любопытно, что в языке C выражение `1<=x<=8` будет синтаксически правильным, но бессмысленным (всегда истинным). Там сначала выражение `1<=x` даёт значение 1 (**true**) для `1<=x`, и 0 (**false**) для `1>x`, а затем `{0,1}≤8` всегда даёт значение 1 (**true**) 🙂.

Заметим также, что знак присваивания `:=` (walrus, «моржовый» оператор) во многих языках (Алгол, Паскаль, Go, Eiffel, Ада и других) не является операцией, как следствие, оператор присваивания, скажем `x:=1`, не имеет значения. Правда, некоторые языки при этом позволяют «совместить» несколько операторов присваивания, например, в языках PL/1 и Ада:

```
x,y=1;              { Язык PL/1 }
x,y: integer:=1 { Язык Ада }
```

В других языках (Алгол, C и т.д.) знак присваивания является операцией, а оператор присваивания имеет значение (присвоенное переменной), поэтому можно писать:

```
a=b=c=d=1                  { Язык Python }
x=y=1;    if (x=2) y=x  { Язык C, y=2 ⚠}
x:=y:=1; if x:=y then  { Язык Алгол }
x<-y<-1;                   { Язык APL <- это := ⚠}
```

В последних версиях языка Python сделано «всё это, вместе взятое, но наоборот»: есть и «сишное» `=` (является *оператором*, не имеющим значения ⚠) и «паскалевское» присваивание `:=`, (является *операцией*, имеющей значение ⚠), правда, эта операция может использоваться только внутри выражений, например:

```
x=x+(a:=b:=a+1)*3;          { Язык Python }
```

Теперь поймём, как выполняются операции *одинакового* старшинства. Для этого случая операции делятся на лево ассоциативные (left-associative – выполняются слева направо) и право ассоциативные (right-associative – выполняются справа налево). В математике из арифметических право ассоциативной является только операция возведения в степень:

A^{B^C} надо вычислять как $A^{(B^C)}$, а не как $(A^B)^C$

В стандарте Паскаля только одна право ассоциативная операция **not**, т.е. `not not x` означает `not (not x)`.

В языках программирования (да, и в математике) есть и не ассоциативные операции, две такие операции в выражении нельзя ставить подряд. В стандарте Паскаля (да и в математике) не ассоциативными являются унарные операции + и –, т.е. нельзя, например, писать `--X`, надо использовать скобки `-(-X)`. В Паскале не ассоциативны логические операции, так что у нас нельзя писать `A and B or C`, надо обязательно (безотносительно к старшинству операций) использовать скобки `(A and B) or C` 🙁.

Все остальные операции Паскаля лево ассоциативные, кроме того, как уже говорилось, «лишние» круглые скобки для операций *одинакового старшинства* могут быть проигнорированы, т.е. оператор `v:=x+(y+z)` может выполняться как `v:=(x+y)+z`.[1] Для гарантии порядка вычисления следует использовать вспомогательную переменную, например, `t:=x+y; v:=t+z`. Кроме того, для бинарных операций не гарантируется порядок вычисления их операндов. Например, при выполнении оператора `z:=f(x)+f(y)` сначала может вызываться функция `f(x)`, а потом функция `f(y)`, а может и наоборот. Для гарантии порядка вычисления операндов следует снова использовать вспомогательные переменные, например: `p:=f(x); z:=p+f(y)`.

Итак, при вычислении значения выражения после каждой операции однозначно определён тип её результата, следовательно, тип каждого выражения можно определить, не вычисляя значение этого выражения. Те выражения, которые имеют целый или вещественный тип, называются арифметическими, те, которые имеют логический тип – логическими и т.д.

Вычисление логических выражений имеет свою специфику. Например, рассмотрим логическое выражение

```
(x=0) and ({ Здесь что-то длинное и сложное })
```

Ясно, что при `x<>0`, какой бы ни был второй логический сомножитель, результат всё равно будет **false**. Так зачем же тратить машинное время и вычислять второй сомножитель? В языке Free Pascal (как и во многих других языках) по умолчанию включено так называемое сокращённое вычисление логических выражений (short-circuit Boolean evaluation), часто его называют также сокращённая *оценка* или *короткое замыкание* логических выражений. В этом режиме вычисление выражения прекращается, как только становится известен конечный результат. При этом могут быть удивительные результаты, например, при `x=0` выражение

```
(x=0) or (1/x>0)  { даст значение  true }
```

---

[1] Правильнее говорить, «проигнорированы для операций одинакового старшинства *и арности*». Арность (редко спользуемые «русские» термины: местность, вместимость или валентность) задаёт количества операндов у операции: унарная, бинарная, тернарная и так далее. Например, в стандарте языка Паскаля операция «унарный минус» `(-x)` и «бинарный плюс" `-(x+y)` имеют одинаковое старшинство, но круглые скобки в выражении `-(x+y)` убрать нельзя, они задают порядок выполнения операций разной арности.

Программист на языке Free Pascal может выключить такое сокращённое вычисление директивой `{$B+}`, это необходимо делать, в частности, когда в логическом выражении используются так называемые функции с побочным эффектом, о чём мы будем говорить при изучении функций. Некоторые языки (Lisp, Haskell) всегда применяют сокращённое вычисление, другие (Java, Delphi) допускают переключение из одного режима в другой.

Объясните, почему для логического выражения `f(x) xor f(y)` сокращённое вычисление никогда не применяется.

Отметим, что *арифметические* выражения в Паскале всегда вычисляются полностью, например, при `x=1`

```
(x-1)*({ Здесь всё будет вычисляться })
```

В качестве шутки можно упомянуть, что в физической теории струн было выведено уравнение:

<Формула$_1$>+(n-11)*<Формула$_2$>

Здесь n – размерность пространства, <Формула$_1$> ну о-о-очень сложная, а <Формула$_2$> такая, что вообще непонятно, как её вычислять. Тогда физики «струнники» сказали: «Очевидно, что размерность нашего пространства n=11…».

Сокращённое вычисление логических выражений является частным случаем так называемых ленивых (или отложенных) вычислений (lazy evaluation). В такой схеме работы выражение (или его часть) вычисляется только тогда, когда на самом деле понадобится его значение.

Вернёмся теперь к оператору присваивания. Обратите внимание, если выражение и переменная синтаксически правильные, то это не означает, что весь оператор присваивания будет синтаксически правильным, например:

```
var x: char; … x:=1; { Несоответствие типов }
```

Итак, чтобы оператор присваивания был правильным, надо потребовать, чтобы тип выражения соответствовал типу переменной, такие типы называются совместимыми (compatible) по присваиванию. В стандарте Паскаля определены только четыре случая, когда есть совместимость по присваиванию:[1]

1). Типы идентичные (тождественные), они именованные, имеют одинаковые имена или типы объявлены равными, например:

```
type int=integer; { Тип int идентичен типу integer }
var  x: integer; y: int; { Типы x и y идентичны }
```

Типы считаются идентичными и тогда, когда переменные описаны с одним (любым, даже безымянным) типом через запятую, например:

```
var a,b: array[1..5] of char; … a:=b; { a и b идентичного типа }
    p: array[1..5] of char; { a и p НЕ ИДЕНТИЧНОГО типа }
```

2). Один тип есть ограничение другого, или оба типа есть ограничения одного и того же базового типа, например:

```
var x: integer; y: 1..100; z: -100..10;
begin x:=y; x:=z; y:=z; { x,y,z совместимы }
```

В разделе 3.4.5 мы уже обсуждали, что будет, если значение в правой части оператора присваивания не будет попадать в диапазон значений переменной из левой части.

3). Тип переменной вещественный, а тип выражения целый (или ограниченный целый). Как мы знаем, в этом случае Паскаль производит неявное преобразование целого значения в вещественное.

4). Оба типа являются строковыми типами с одинаковым количеством компонент-символов, причём либо оба являются, либо оба не являются упакованными (**packed**) типами (отложим до раздела 7.2).

5). Оба типа являются множественными типами с совместимыми базовыми типами (отложим до главы 10).

---

[1] Существует ещё так называемое *структурное* соответствие (structure compatibility) типов, о нём мы будем говорить при изучении массивов в разд. 7.1.

Итак, мы ввели важное понятие *соответствие* (или *совместимость*) по присваивания. Далее при описании синтаксиса Паскаля будем говорить «здесь требуется соответствие (совместимость) по присваиванию» и Вы должны чётко понимать, что это такое.

В языке Free Pascal есть упомянутый ранее тип `variant`, для которого статическая проверка типов не производится. Вместо этого при необходимости производится *динамическое* преобразование типов, например (отслеживайте старшинство типов, кто куда преобразуется):

```
var X: variant;
  X:=2; X:=true; X:='10';
  Write(X+1)          { Вывод 11 }
  Write(X+'1')        { Вывод 101, это '101' }
  Write(X and true)   { Вывод true }
  Write(X*1.5)        { Вывод 15    как в калькуляторе! }
  Write(X+1.5)        { Вывод 11.5  как в калькуляторе! }
  Write(X+true)       { Вывод 9     ?? 😳 загадка...}
```

Итак, надо понять, что в языках со статической типизацией оператор присваивания меняет значение переменной (связывает переменную с новым значением), а в языках с динамической типизацией (в наших переменных типа `variant`) он меняет ссылку (связывает имя переменной со ссылкой на область памяти с новым значением). Некоторые языки (например, Python) имеют только динамическую типизацию.

Оператор процедуры. Этот весьма сложный оператор мы будем изучать постепенно. Сначала рассмотрим так называемые стандартные процедуры ввода/вывода. Паскаль о них уже «знает», их не надо описывать перед использованием. «Настоящие» процедуры, описываемые пользователем, мы изучим немного позднее, когда познакомимся со всеми операторами Паскаля.

4.2. Процедуры ввода/вывода

Чтобы понять программу, необходимо отождествить себя и с машиной, и с программой.

Алан Перлис,
первый лауреат премии Тьюринга

Важнейшем частью любого алгоритма является получение входных данных и вывод результатов работы. В одних языках (Fortran, Pascal, Python и др.) содержатся *встроенные* средства ввода и вывода, заданные служебными или стандартными процедурами. В других языках (C, Ассемблер и т.д.) таких средств нет, программы вынуждены вызывать для этого обычные процедуры (как правило, они включаются в так называемые библиотеки стандартных программ).

В Паскале для ввода/вывода предусмотрены четыре стандартные процедуры с именами `read` и `readln` (для ввода) и `write` и `writeln` (для вывода). Все эти процедуры работают с так называемыми потоками ввода/вывода. Все входные данные поступают в программу через потоки ввода, а все результаты выдаются во «внешний мир» через потоки вывода. Впрочем, существуют «универсальные» потоки, через которые возможен как ввол, так и вывод данных.

Сначала поймём, что такое поток. Каждый поток состоит из квантов (порций) данный. За один раз вводится и выводится одна порция данных. Большинство языков предполагает, что каждый поток состоит из одинаковых порций. Это, однако, необязательно, например, язык Фортран допускает потоки с порциями разной длины.

В Паскале особо выделены два стандартных потока, это стандартный поток ввода с именем `input` и стандартный поток вывода с именем `output`. Квантами этих потоков являются символы (character) текста.

В языке Free Pascal есть ещё стандартный поток вывода с именем `erorutput`, он предназначен для вывода сообщений об ошибках во время выполнения программы, обычно он, как и поток `output`, связывается с экраном консоли. Под влиянием языка C у имён стандартных потоков в языке Free Pascal есть синонимы `stdin`, `stdout` и `stderr`. Стандартные имена потоков являются так на-

зываемыми файловыми переменными. Связь между потоками и файлами мы рассмотрим позже, при изучении файлов в главе 11.

> Консоль – комплект устройств текстового интерактивного ввода/вывода (текстовый экран или окно на экране, физическая или экранная клавиатура, и, возможно, мышь или сенсорный указатель) для взаимодействия программы с внешним объектом (вероятно, с человеком 🙂).

Здесь важно понять, что стандартный поток ввода не есть просто поток символов, это символы текста. Как мы знаем, текст состоит из строк символов переменной длины (допускаются и пустые строки). Следовательно, например, при вводе символов из такого потока существуют моменты, когда кончается одна строка и мы переходим на новую строку текста.

Поток данных лучше представлять себе не как поток воды, а скажем, как кофейный автомат. За каждое обращение к нему автомат выдаёт одну порцию напитка. Учтите, что в любой момент поток может иссякнуть (закрыться), более того, он может быть пустым и изначально (автомат внезапно сломался 😐). Чтение из пустого потока приводит к исключительной ситуации (Run Time Error). Для проверки, не пуст ли поток `input` предназначена стандартная логическая функция без параметров с именем `eof` (End Of File). Когда эта функция возвращает значение **`true`**, то поток пуст и читать из него нельзя.

> В языке Free Pascal есть ещё полезная функция `SeekEof`, она удаляет из входного потока все последующие пустые строки и строки, содержащие только пробелы и символы табуляции, останавливаясь на первой «непустой» («содержательной») строке. Функция возвращает значение **`true`**, если входной поток после этого становится пустым.

Сначала рассмотрим стандартные процедуры ввода. Их синтаксис:

```
<процедура ввода> ::= read([input,]<список ввода>)|
                      readln[([input,]<список ввода>)]
<список ввода> ::= <переменная>{,<переменная>}…
```

Список ввода является списком *скалярных* переменных, дополнительно семантическое условие накладывает ограничение на их тип, он может быть только одним из следующих:

1). Целый (и ограниченный целый, помним, что они эквивалентны).
2). Вещественный.
3). Символьный и ограниченный символьный.
4). Символьная строка.

Здесь надо понять, что входной поток Паскалевской программе не принадлежит, его «хозяином» является операционная система компьютера. Программа `read` (`readln`) получает порцию данный (для потока `input` это один символ), обращаясь к служебной программе операционной системы, которая обслуживает так называемый буфер ввода этого потока. Работа этой программы очень похожа на работу кладовщика, обслуживающего склад, так что будем образно называть её «кладовщик».

Взаимодействие Паскаля с «кладовщиком» рассмотрим на простом примере. Не будем пока проверять, что поток `input` пуст.

> Обычно поток `input` подключён к клавиатуре (физической или экранной), он может оказаться пустым, если клавиатура неисправна (например, отключена от ЭВМ), у компьютера с *беспроводной* клавиатурой садится источник питания, программа сама закрыла поток процедурой `close(input)`, или же пользователь ввёл особую комбинацию клавиш (обычно `Ctrl+D`), «закрывающую» этот поток.

Опишем переменные

```
var x: integer; y: real; z: char;
```

Пусть в потоке `input` содержится строка (стрелочкой ↑ помечен текущий символ для ввода, пробел обозначен «видимым» символом ␣):

```
-123␣45.6␣A789…
↑
```

Рассмотрим семантику работы процедуры ввода

```
read(x,y,z)
```

Ввод значения каждой переменной из списка ввода проходит в три этапа. Сначала у нас в списке ввода стоит *целочисленная* переменная, поэтому надо ввести из потока `input` целое значение в переменную `x`.

1 этап. Процедура начинает обращаться к «кладовщику» получая от него символ за символом. Её цель – собрать из вводимых символов правильную лексему целого числа. В данном примере процедуре это удаётся сделать, введена лексема `-123` (введённый последним пробел является границей лексемы). В потоке остаётся

```
45.6␣A789…
↑
```

Отметим, что лексема может кончаться не только пробелом, но и служебными символами табуляции и конца строки,[1] в этом случае при запросе ещё одного символа «кладовщик» выдаст первый символ следующей строки. Учтите, что строки могут быть и пустыми, тогда «кладовщик» просто переходит к следующей строке текста.

Когда хорошую лексему нужного типа получить не удалось, поведение программы в стандарте Паскаля не определено. В языке Free Pascal поведение программы зависит от заданного программистом режима ввода. При вводе с контролем (он задаётся директивой `{$I+}`, есть «длинный» синоним `{$IOChecks ON}`, этот режим установлен по умолчанию), возникает исключительная ситуация. При вводе без контроля `{$I-}` или более понятно `{$IOChecks OFF}` процедура ввода просто заканчивает свою работу, и исполнитель переходит к следующему оператору программы (без правильного ввода!).

2 этап. Полученная лексема переводится во внутреннее машинное представление целого числа. Это закодированное (битовое) представление. Когда полученное целое число выходит за границы диапазона представимых целых чисел, то в стандарте Паскаля результат ввода не определён. В языке Free Pascal действия зависит от режима работы. При режиме с контролем `{$I+}` возникает исключительная ситуация «Exitcode=106. Invalid numeric format», а при режиме без контроля `{$I-}` процедура ввода заканчивает свою работу, и исполнитель переходит к следующему оператору программы. Вводимая переменная (и все следующие за ней в списке ввода) при этом получает неправильное значение и программа продолжит выполняться «как ни в чём не бывало». Подробно об этом будем говорить далее.

3 этап. Полученное на втором этапе значение присваивается переменной (поймите, что они совместимы по присваиванию). При выходе введённого значения за допустимый диапазон переменной из списка ввода, в стандарте Паскаля результат не определён. В языке Free Pascal, как мы уже знаем, этот результат зависит от режима работы. В режиме `{$R-}` переменная получает неправильное значение (просто усекается), а в режиме `{$R+}` возникает исключительная ситуация «Exitcode=201. Range check error». Видите, как сложно работает наша на первый взгляд простая операция ввода… 😕.

Далее по такой же схеме вводится значение вещественной переменной `y`:

лексема `45.6` ⇒ машинное представление ⇒ присваивание `y`

В языке Free Pascal для кодировки вещественного числа при вводе используется самый большой диапазон `extended`.[2] Отметим, что при вводе значения вещественной переменной правильной считается и лексема *целого* числа, при этом на этапе присваивания введённое целое значение автоматически преобразуется в вещественное. В потоке остаётся

```
A789…
↑
```

Проще всего вводится символьная переменная, её лексема всегда один символ `'A'`, в потоке остаётся

```
789…
↑
```

---

[1] В разных операционных системах и разных устройствах конец строки оформляется по разному. В Windows для клавиатуры это служебный символ с номером 13 (CR – Enter), а для текстового файла это два служебных символа 13 (CR) и 10 (LF), в Unix и Android это один символ 10 (LF).

[2] Так происходит на процессорах, совместимых с процессорами так называемой Интеловской архитектуры. На других типах процессоров используется тип `double`.

Когда список ввода исчерпан, то выполнение оператора процедуры ввода завершается, и производится переход к следующему оператору. Итак, на каждом из трёх этапах возможны ошибки. На первом этапе это лексические ошибки, например, строки `789A` и `1-2` никак не могут быть лексемами целого числа. На втором этапе это выход значения лексемы за (максимально) допустимый диапазон, а на третьем – ошибки присваивания переменной значения, выходящего за её диапазон. В языке Free Pascal ошибки первых двух этапов, как уже говорилось, управляются директивой `{$I±}`, а на третьем – директивой `{$R±}`.

Таков механизм ввода данных на так называемом внешнем (или логическом) уровне. К сожалению, при программировании некоторых задач этого недостаточно, и надо знать механизм ввода/вывода на внутреннем уровне. На этом уровне надо более глубоко понять взаимодействие процедуры ввода с «кладовщиком» и «кладовщика» с потоком. Итак, соберясь с духом, приступим …

Внутренний уровень. Во-первых, процедура `read` не знает, куда конкретно в данный момент подключён поток `input`. Это может быть физическая и экранная клавиатура, текстовый файл, особые каналы, связывающие параллельно работающие программы (процессы), так называемые сокеты, обеспечивающие сетевой ввод/вывод, или даже что-то экзотическое, вроде перчатки виртуальной реальности.

В некоторых языках (PascalABC.NET и др.) понятие «поток» обобщается на понятие «последовательность». Для входной последовательности порции данных могут поступать, например, от некоторой функции, которая генерирует эти данные (функция-генератор будет вызываться при каждом чтении из потока). Для выходной последовательности порции данных передаются как параметры в некоторую подпрограмму.

Итак, поток подключается к некоторому устройству, хранилищу данных или порождающей функции. Разумеется, «кладовщик» уже точно знает, к какому устройству сейчас подключён поток `input`. Именно туда (к драйверу конкретного устройства) он посылает своего представителя (будем образно называть его «экспедитором») за входными данными.[1]

Далее, оказывается, что и сам поток `input` может работать в нескольких режимах. Познакомимся с этими режимами, каждый из них двоичный, т.е. может быть *включён* или *выключен*.

1. Режим с буферизацией/без буферизации. В режиме с буферизацией «кладовщик» посылает «экспедитора» на грузовике в поток `input` с такой инструкцией: «Поезжай и жди там символы, по мере их поступления грузи на грузовик, когда поступит служебный символ конца строки, вези строку ко мне на склад». Получив строку, «кладовщик» раскладывает привезённые символы по ячейкам своего склада и выдаёт их потом по одному процедуре ввода. В режиме без буферизации «экспедитор» получает другую инструкцию: «Поезжай и жди там первый символ, потом сразу вези его мне на склад». Как видим, в этом случае как такового склада нет, символы поступают в процедуру ввода сразу, как только они появляются в потоке `input`. Понятна прагматика этого режима, например, в компьютерной игре надо быстро получать вводимые символы (скажем, команды «стреляй»), если ждать конца строки, то игрока, конечно, успеют убить 🙂.

2. Режим с контролем/без контроля.[2] В режиме с контролем (cooked mode) «экспедитор», направляясь в поток `input`, получает от «кладовщика» список особых, служебных символов. Около каждого из этих символов сказано, что делать, когда он поступит из потока `input`. Например, для символа `BS` надо удалить из грузовика последний погруженный туда символ (а если грузовик пуст, то ничего не делать), для символа `ESC` очистить грузовик ото всех символов и т.д. Таким образом, эти служебные символы «обрабатываются на месте» и в процедуру ввода никогда поступить не могут. Конечно, в основном эти служебные символы применяются при вводе данных с клавиатуры, это позволяет редактировать ввод. Таким образом, в режиме с буферизацией и контролем, пока пользователь не нажал `Enter`, он может как угодно изменять вводимую строку. В режиме без контроля (raw mode) в списке остаётся совсем мало служебных символов, обычно это `Enter` (конец строки для режима с буферизацией) и символ `Ctrl+D` (он говорит, что поток `input` закрыт, и больше из него символов ждать не стоит).

---

[1] Можно было бы считать, что «кладовщик» и есть этот драйвер, но так будет менее понятно.

[2] У английского глагола to control главное значение не «контролировать», а «управлять», поэтому более правильно было бы говорить с управлением/без управления, но, как переведено, так уж и переведено…

3. Режим с эхом/без эха. В режиме с эхом (echo input) «экспедитор», направляясь в поток `input`, обязательно берёт с собой коллегу, экспедитора потока `output`. Далее, получив из потока `input` очередной символ, он снимает с него копию и отдаёт коллеге из потока `output` (копия снимается до проверки на служебные символы). Так как поток `output` по умолчанию подключён к экрану консоли, то так пользователь видит, что он набирает на клавиатуре. Поймите, что при подключении к экрану поток `output` работает в режиме *без буферизации*, так что набираемые символы сразу появляются на экране. Прагматика режима работы без эха очевидна, так, например, вводятся пароли, чтобы символы пароля не появлялись на экране. В этом случае программа проверки пароля, получая эго символы в режиме без эха, сама выводит в поток `output`, скажем, символы * (или не выводит ничего).

В учебниках по Паскалю эта тема часто не описывается. Дело в том, что в стандарте Паскаля поток `input` всегда работает в режимах с буферизацией, с контролем и с эхом. В языке Free Pascal добавляется возможность работать в режимах без буферизации, без контроля и без эха, для этого надо использовать не процедуру ввода `read`, а функцию с именем `ReadKey`, например, `c:=ReadKey`.

Как программист может узнать, что в режиме с буферизацией на складе закончились все символы очередной строки, и при запросе процедурой ввода нового символа будет вводится следующая строка? Для этого в Паскале предназначена функция `eoln`, которая возвращает значение **`true`**, если все символы очередной строки уже введены (склад пуст).

В языке Free Pascal на «пустом» складе находятся служебные символы (признаки конца строки). Кроме того, отметим, что есть ещё полезная функция `SeekEoln`, она удаляет из текущей строки все символы пробелов и символы табуляции (до первого отличного от них символа). Функция возвращает значение **`true`**, если после этого осталась пустая строка.

Процедура `readln` имеет дополнительное назначение, она управляет буфером ввода. После обработки своего списка ввода она посылает «кладовщику» команду очистки буфера: склад становится пустым и очередной символ будет поступать из начала *следующей* строки. Таким образом, процедура `readln` без параметров просто очистит буфер ввода.

Теперь рассмотрим стандартные процедуры вывода. Стандартные процедуры вывода (как и ввода), не знают, к какому конкретному устройству подключён поток вывода. Их синтаксис:

```
<процедуры вывода> ::= Write([output,]<список вывода>)|
                       Writeln[([output,](<список вывода>)]
<список вывода> ::= <элемент>{,< элемент>}…
<элемент> ::= <выражение>[:<спецификатор>[:<спецификатор>]]
<спецификатор> ::= <целочисленное выражение>
```

Список вывода является списком *выражений*, которые надо сначала вычислить и затем вывести вычисленные значения. Дополнительно семантическое условие накладывает ограничение на тип выражений, он может быть только одним из следующих:

1). Целый (и ограниченный целый, помним, что они эквивалентны).
2). Вещественный.
3). Символьный и ограниченный символьный.
4). Логический и ограниченный логический.
5). Строковый тип. С этим типом мы познакомимся при изучении массивов. Пока мы знаем только константы этого типа, это (не пустые) строки в апострофах.

Каждое выражение в списке вывода обрабатывается в два этапа. Сначала выражение вычисляется (здесь может возникнуть исключительная ситуация, например, деление на ноль или переполнение). Затем полученное значение преобразуется в *лексему* соответствующего типа и выводится в поток `output`. Подключённый к экрану, поток `output` обычно работает в режимах без буферизации, с контролём и без эха. Например:

```
var x: integer; y: real;
begin x:=10; y:=-123.4;
   Write(-10*x+3, y+2*x, x>y, succ('2'))
```

Первое целочисленное выражение из списка вывода равно `-97` и имеет лексему из трёх символов, выводимых в поток `output`. Вещественное выражение равно `-103.4`, его преобразование в лексему неоднозначно, это могут быть строки `-103.4`, `-1.034E+2` или `-1034e-3`. Кроме того, точки `-103.4` на вещественной оси, скорее всего, нет, а есть, скажем, `-103.39999`. Для решения этих проблем есть стандартное представление лексемы вещественного числа в виде

```
[-]x.xxxxxxxxxxxxxxE±xxx
```

у нас будет

```
-1.03400000000000E±002
```

Такой вид чаще всего неудобен, и программист может *управлять* видом лексемы вещественного числа, задавая у выводимого выражения спецификаторы (это называется форматированием). У выражения можно задать один или два спецификатора, они являются целочисленными выражениями (в простейшем случае просто целыми константами).

Первый спецификатор задаёт ширину поля вывода, т.е. число символов в которые преобразуется лексема, например:

```
Write(-1.034:11)  {-1.034E+002  Всего 11 позиций}
Write(-1.034:9)   {-1.0E+002  Всего 9, округление }
Write(-1.034:6)   {-1.0E+002  6 → 9, меньше нельзя! }
Write(1234:6)     {␣␣1234  Всего 6, впереди два пробела }
Write(1234:1)     {1234          1 → 4, меньше нельзя! }
Write('*':4)      {␣␣␣*  Три пробела и * }
```

Как видно, при маленьком поле его длина в Паскале автоматически расширяется до минимально необходимого. А вот когда длина поля *больше* длины значения, по впереди добавляется нужное число пробелов, т.е. выводимое значение прижимается к правой границе поля.

В разных языках программирования принимаются свои решения, как выводить значение в поле, длина которого для этого недостаточна. Например, в Фортране оператор форматного вывода целого значения `Print (*,'I3'),1234` напечатает `***`. Здесь надо вывести четыре цифры, но задано поле вывода целого типа `I3` из трёх позиций, в Фортране принято решение вывести три звёздочки. Заметим, что, в отличие от Паскаля, такой вывод не искажает вид числовых таблиц, когда под каждую колонку отводится определённое число позиций, звёздочки предупреждают, что число не поместилось в отведённое поле столбца таблицы.

Два спецификатора используются только при выводе вещественных значений, при этом выбирается вид лексемы без порядка (без букв `E` и `e`), а второй спецификатор задаёт число цифр в дробной части числа, например:

```
Write(12.345:6:2)  {␣12.34  Всего 6, 2 дробных, округление }
Write(12.345:4:1)  {12.3    Всего 4, 1 дробная, округление }
Write(12.345:4:0)  {␣␣12    Всего 4, 0 дробных, в целое }
Write(12.345:2:1)  {12.3          2 → 4, меньше нельзя }
```

Рассмотрим теперь вывод логических и символьных значений

```
Write(x>y, succ('2'):2)
```

У нас `x=10`, `y=-123.4`, тогда `x>y` трактуется Паскалем как `real(x)>y`, таким образом `10.0>-123.4` это истина. Логическое значение истина выводится лексемой `TRUE`, соответственно, ложное значение выводится у нас лексемой `FALSE`. И, наконец, символьное выражение `succ('2')='3'`, выводится просто символом `'3'` в двух позициях, итого будет `FALSE␣3`. Отдельно отметим вывод символьных строк в апострофах

```
Write('Значение x=',x)  { Значение x=10 }
```

Во всех языках программирования есть свои средства для форматирование значений перед выводом. Некоторые языки (например, Фортран и C) имеют развитые возможности форматирования, позволяющие в широких пределах управлять внешним видом выводимых данных. В языке Free Pascal для тонкого форматирования предназначена функция с именем `Format` из модуля `SysUtils`, она предоставляет такие же развитые средства, как и указанные выше языки.

Вывод русского текста в некоторых операционных системах может представлять определённую трудность. Например, для Windows текст программы, как правило, набирается в каком-нибудь текстовом редакторе в кодировке Windows (CP1251), а консольное окно по умолчанию «считает», что текст в него выводится в кодировке DOS (CP866). Латинские буквы в этих кодировках совпадают, а вот русские нет. В языке Free Pascal простейшим выходом будет применение простой функции перекодировки (она находится в модуле `Windows`):

```
uses Windows;
  function Rus(x: Ansisting): Pchar;
    var a: Pchar;
  begin a:=Pchar(x); OemToChar(a,a); Rus:=a end;
  var   s: string='Любой русский текст';
begin Write(Rus('Любой русский текст'),Rus(s));
```

О типа данных `Ansisting` и `Pchar` будет рассказано в главе, посвящённой работе со строками.

Вопросы и упражнения

Трудность работы с программистом заключается в том, что вы не можете понять, что он делает, до тех пор, пока не стало слишком поздно.

Сеймур Крей
конструктор первой супер ЭВМ

1. Как определить, сколько операторов в разделе операторов?
2. Когда программа на Паскале завершает свою работу?
3. Чем простые операторы отличаются от сложных?
4. Зачем нужен пустой оператор?
5. Что такое простое выражение?
6. Правильное ли выражение `a<b<c` для логических переменных `a`, `b` и `c` ?
7. Чем отличаются бинарные операции от унарных?
8. Чем отличаются лево ассоциативные операции от право ассоциативных?
9. Как управлять видом лексемы выводимого значения?
10. Сколько уровней старшинства операций в Паскале?
11. Что такое сокращённое вычисление логических выражений?
12. Что такое соответствие по присваиванию?
13. Что такое поток ввода/вывода?
14. Чем отличаются стандартные потоки от не стандартных?
15. На каких этапах ввода значения в переменную возможны ошибки?
16. Как определить, что текстовая строка потока `input` введена полностью?
17. Что такое режим работы потока?
18. Для чего нужна директива контроля ввода/вывода `{$I-}` в языке Free Pascal?
19. Что происходит, если выводимое значение не помещается в отведённое ему поле для вывода?