

Сообщение о языке программирования Выхухоль

Соколова Т. А..

_____ 20__ г.

Оглавление

1	Язык	2
1.1	Лексемы и форма описания синтаксиса	2
1.1.1	Ключевые слова и идентификаторы	2
1.1.2	Числа	3
1.1.3	Литёры и строки	3
1.1.4	Знаки операций и разделители	3
1.2	Структура программы	4
1.3	Описания	4
1.3.1	Описание типов	4
1.3.1.1	Логические типы	5
1.3.1.2	Символьные типы	6
1.3.1.3	Строковые типы	7
1.3.1.4	Числовые типы	7
1.3.1.5	Пустой тип	8
1.3.1.6	Кортежи	8
1.3.1.7	Указатели	9
1.3.1.8	Ссылки	9
1.3.1.9	Типы указателей на функции	9
1.3.1.10	Массивы	9
1.3.1.11	Алгебраические типы данных	11
1.3.2	Описание переменных	11
1.3.3	Описание констант	11
1.3.4	Описание функций	11
1.4	Выражения	11
1.5	Операторы	12
1.5.1	Операторы присваивания	12
1.5.2	Условный оператор	12
1.5.3	Оператор выбора	12
1.5.4	Оператор разбора значения алгебраического типа	13
1.5.5	Убийца морали	13
1.5.6	Переводчик в десятичную систему	13
1.5.7	Операторы цикла	13
1.5.7.1	Оператор цикла с предусловием	13
1.5.7.2	Оператор цикла с постусловием	13
1.5.7.3	Оператор цикла „вечно повторяй“	14
1.5.7.4	Оператор цикла „для“	14
1.5.7.5	Оператор цикла „Iam“	14
1.5.8	Оператор выхода из цикла	15
1.5.9	Оператор выполнения следующей итерации цикла	15
1.5.10	Оператор возврата из функции	15
2	Стандартная библиотека	16
2.1	Математические функции	16
2.2	Ввод-вывод	16

Глава 1. Язык

1.1. Лексемы и форма описания синтаксиса

Синтаксис языка программирования Поро описан как словесно, так и с помощью расширенных формул Бэкуса–Наура (РБНФ). В РБНФ формула состоит из двух частей: первая часть содержит имя определяемого понятия (в используемой здесь версии РБНФ оно выделяется зелёным цветом), а затем, после метасимвола \rightarrow , идёт вторая часть, содержащая определение понятия. В данном описании кроме метасимвола \rightarrow будут также использоваться следующие метасимволы:

- | — означает „или“;
- { } — содержимое этих скобок может повторяться любое число раз, в том числе и ни разу;
- () — эти скобки группируют конструкции;
- [] — содержимое данных скобок является необязательным.

Текст программы на языке Рысь состоит из лексем. В языке имеется четыре класса лексем:

- 1) ключевые слова и идентификаторы;
- 2) числа;
- 3) знаки операций и разделители;
- 4) литеры и строки.

Никакая лексема не может разбиваться на части пробельными символами (т.е. пробелами, табуляциями и концами строк) или комментариями. Опишем каждый класс лексем.

1.1.1. Ключевые слова и идентификаторы

Идентификатор — это последовательность русских и латинских букв, десятичных цифр и знаков подчёркивания. Идентификатор должен начинаться с буквы или со знака подчёркивания. Прописные и строчные буквы считаются различными. Идентификатор не может совпадать ни с каким ключевым словом. Ключевые слова языка записываются строчными буквами и в настоящем описании выделяются жирным шрифтом. Ниже приведён список ключевых слов:

anl	false	propaganda10	unval
array	for	r	unval128
big	frv	r128	unval16
c	function	r32	unval32
c128	head	r64	unval64
c32	if	r80	unval8
c64	ins	rep	var
c80	kill	ret	while
cho	link	sel	whilet
clean	log	small	z
continue	log16	string	z128
const	log32	structure	z16
deliv	log64	sym	z32
else	log8	that	z64
empt	mod	true	z8
ext	num	type	Iam

Примеры идентификаторов: cat, Dog, _Cat121taC, Dog_Dog, Pumpkin.

1.1.2. Числа

Числа — это беззнаковые целые и вещественные константы, а также комплексные константы. Знаковые константы — это беззнаковые константы, к которым применена унарная операция смены знака (операция „—“). Запись в РБНФ:

```
число→целое|вещественное|комплексное
целое→десятичное|шестнадцатиричное|двоичное|восьмеричное
десятичное→десятичная_цифра{десятичная_цифра}
десятичная_цифра→0|1|2|3|4|5|6|7|8|9
шестнадцатиричное→0(x|X)шестн_цифра{шестн_цифра}
шестн_цифра→0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f
двоичное→0(b|B)двоичн_цифра{двоичн_цифра}
двоичн_цифра→0|1
восьмеричное→0(o|O)восьмеричн_цифра{восьмеричн_цифра}
восьмеричн_цифра→0|1|2|3|4|5|6|7
вещественное→целая_часть[.дробн_часть](E|e)[(+|−)порядок][точность]
целая_часть→десятичное
дробн_часть→десятичное
порядок→десятичное
точность→одинарная|двойная|расш|четырёхкр
одинарная→f
двойная→d
расш→x
четырёхкр→q
комплексное→вещественноеi
```

1.1.3. Литёры и строки

Литёра — это либо произвольный символ, заключённый в одинарные (') или двойные (") кавычки, либо выражение вида *\$целое*. Строка — это либо последовательность из ноль или более символов, заключённых в кавычки; либо последовательность литёр, заданных посредством кодов; либо чередование того и другого. Под литёрой, заданной посредством кода, понимается выражение вида *\$целое*. Открывающая кавычка должна совпадать с закрывающей. Если в строке требуется записать кавычку, совпадающую с открывающей, то кавычка должна быть продублирована.

Примеры строк:

'' — пустая строка

'Хорошо живёт на свете Винни-Пух!'\$13\$10

'У попа была собака, он её любил.

Она съела кусок мяса, -- он её убил.

В землю закопал и надпись написал: ...'

'Привет лунатикам!'\$0

1.1.4. Знаки операций и разделители

[.	%	{	%.		###	:=	%.=	**.=
]	:	\	}	..	&&	++<	=	=	(+)
(;		+	?.	<<	--<	&=	&&=	(-)
)	==	&	++	??	>>	+=	^=	^^:=	@->
<-	#	<	--	<=	##	-=	&&.	**=	tt
!	+	>	+. .	>=	=	*=	+. =	<<=	ttt
~	-	?	-. .	!=	@@	/=	-. =	>>=	<-@
^	*	{.	*.	**	~	%:=	*.=	~ =	/
@	/	}.}	/.	^^	~&	\:=	/.=	~&=	

Кроме лексем в любом месте программы могут встречаться комментарии. Комментарий — это последовательность любых символов, заключённых между скобками (* и *). Комментарии могут быть вложенными.

Приведём пример:

```
t:=sin(x)
(* Это комментарий первого уровня вложенности.
  (* Это - второго.
    (* А это - третьего. *)
  *)
*)
```

1.2. Структура программы

Структура программы на языке Gjlhj выглядит так:

```
модуль имя_модуля
{
  {описание}
}
```

Здесь

имя_модуля — идентификатор, являющийся именем данного модуля;

описание — описание типов, переменных, констант, алгоритмов и операций.

1.3. Описания

Область видимости объекта *x* (здесь под объектом понимается тип, переменная, константа, алгоритм или операция) текстуально распространяется от точки его описания до конца блока (модуля, тела составного оператора, тела подпрограммы), к которому принадлежит описание и по отношению к которому объект, таким образом, считается локальным. Из этой области исключаются области видимости объектов с таким же именем, описанных в блоках, вложенных в данный. Правила видимости таковы.

- 1) Идентификатор может обозначать только один объект в данной области видимости (т.е. никакой идентификатор не может быть объявлен в блоке дважды).
- 2) На объект можно сослаться только в его области видимости.
- 3) Описание типа *T*, содержащее ссылки на другой тип *T₁*, может стоять в точках, где *T₁* еще не известен. Описание типа *T₁* должно следовать далее в том же блоке, в котором локализован *T*.
- 4) Заголовок функции может быть приведён до того, как будет дано полное определение.

1.3.1. Описание типов

Описание типов выглядит так:

```
тип имя_типа=определение_типа{;имя_типа=определение_типа}
```

Здесь *имя_типа* — идентификатор, являющийся именем определяемого типа; *определение_типа* — либо простейшее определение типа, либо определение алгебраического типа данных.

Алгебраические типы данных будут подробно рассмотрены в разделе, посвящённом таким типам. Здесь же поясним понятие простейшего определения типа.

Простейшие определения типов есть двух категорий:

- 1) стандартные типы;
- 2) простейшие определения типов, задаваемые пользователем.

Стандартные типы можно разделить на четыре вида:

- 1) логические типы;
- 2) символьные типы;
- 3) числовые типы;
- 4) пустой тип.

В свой черёд, числовые типы могут быть следующих подвидов:

- 1) целочисленные типы;
- 2) вещественные типы;
- 3) комплексные типы.

К простейшим определениям типов, задаваемым пользователем, относятся:

- 1) имя типа;
- 2) определение типа-указателя;
- 3) определение типа указателя на функцию;
- 4) определение типа-массива.

Приведём пример определения типов:

```
тип A = small small z;
    B = small z;
    C = z;
    D = big z
```

Здесь типы A и **small small z** — взаимозаменяемы.

Для каждого типа данных можно узнать размер переменной этого типа, для чего перед именем типа или переменной этого типа нужно поставить знак операции **##**.

Для динамических массивов операция **##** даёт размер не самого этого значения, а размер служебной информации. Чтобы узнать размер самого значения динамического массива, нужно перед именем переменной поставить знак операции **###**.

1.3.1.1. Логические типы

Переменная логического типа может принимать только два значения: **истина** или **ложь**. Логический тип выглядит так:

(**{dig}**){small}log

Размер переменной типа **log** зависит от реализации, но не может превышать размера машинного слова. Также имеются логические типы конкретных размеров, а именно, типы **log8**, **log16**, **log32**, **log64**, переменные которых имеют размер в 1, 2, 4 и 8 байт соответственно.

Над логическими значениями определены следующие операции:

	логическое „или“ (сокращённое вычисление)
&&	логическое „и“ (сокращённое вычисление)
^^	логическое „исключающее или“
!	логическое „не“
==	равно
!=	не равно
(+)	есть ли true
(-)	есть ли false

Все логические типы попарно совместимы между собой. Термин „полное вычисление“ означает, что вычисляются все аргументы операции; а термин „сокращённое вычисление“ — что вычисляется лишь часть аргументов.

Приведём таблицы истинности логических операций.

x	y	$x \mid y$	$x \& y$	$x \wedge y$	$x(-)y$
false	false	false	false	false	true
false	true	true	false	true	true
true	false	true	false	true	true
true	true	true	true	false	false

x	y	$(x! \mid y) \equiv x \mid (!y)$	$(x! \& y) \equiv x \& (!y)$
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

x	!x
false	true
true	false

К операциям с логическими значениями тесно примыкают тернарные операции $?:$ и $?:.$. Поясним смысл операций $?:$ и $?:.$.

1) Операция $?:$. Выражение вида $S?A:B$ эквивалентно следующей последовательности действий:

- а) вычислить логическое выражение S , и выражения A и B ;
- б) если ($S=\text{true}$) то
 выдать A
 else
 выдать B
 всё

Замечание 1.3.1. Если в системе команд процессора имеется команда условной пересылки, то для вычислений в пункте б) **должна использоваться именно эта команда.**

2) Операция $?:.$. Выражение вида $S?.A:B$ эквивалентно следующей последовательности действий:

- а) вычислить логическое выражение S ;
- б) если ($S=\text{true}$) то
 вычислить выражение A и выдать полученное значение
 иначе
 вычислить выражение B и выдать полученное значение
 всё

Замечание 1.3.2. Здесь для вычислений в пункте б) **должна использоваться команда сравнения с последующим условным переходом.**

1.3.1.2. Символьные типы

Переменная символьного типа может хранить любой символ, доступный в конкретной реализации. Символьный тип выглядит так:

sym

Для символьных данных определены лишь операции отношения и операция присваивания. Ниже приведён список операций отношения:

< меньше
 > больше
 <= меньше или равно
 >= больше или равно
 == равно
 != не равно

1.3.1.3. Строковые типы

Переменная строкового типа хранит строковые значения. Символьный тип выглядит так:

string

Для строковых данных определены операции отношения, операция присваивания, и операция обращения к символу строки по его индексу. Также определена операция конкатенации (склейки) строк, обозначаемая знаком „+“.

1.3.1.4. Числовые типы

Целочисленные типы. Переменные целочисленных типов предназначены для хранения целых чисел. Целочисленный тип выглядит так:

(({{большое}}|{{small}})[unval|z|unvaln8|unval16|unval32|unval64|
 unval128|unval8|unval16|unval32|unval64|unval128])

Допустимые операции:

+	(бинарный)	целочисленное сложение
+	(унарный)	подтверждение знака
-	(бинарный)	целочисленное вычитание
-	(унарный)	изменение знака
++		следующее значение
--		предыдущее значение
*		целочисленное умножение
/		целочисленное деление
%		целочисленный остаток от деления
**		целочисленное возведение в степень
		поразрядное „или“
~		поразрядное „не-или“
&		поразрядное „и“
~&		поразрядное „не-и“
~		поразрядное „не“
^		поразрядное „исключающее или“
<<		сдвиг влево
>>		сдвиг вправо
<		меньше
>		больше
<=		меньше или равно
>=		больше или равно
==		равно
!=		не равно
tt		деление на 2
ttt		деление на 3

Вещественные типы. Переменные вещественных типов предназначены для хранения вещественных чисел. Вещественный тип выглядит так:

`({big}){small})r|r32|r64|r80|r128`

Допустимые операции:

<code>+</code>	сложение
<code>+</code> (унарный)	подтверждение знака
<code>-</code>	вычитание
<code>-</code> (унарный)	изменение знака
<code>*</code>	умножение
<code>/</code>	деление
<code>%</code>	вещественный остаток от деления
<code>**</code>	вещественное возведение в степень
<code><</code>	меньше
<code>></code>	больше
<code><=</code>	меньше или равно
<code>>=</code>	больше или равно
<code>==</code>	равно
<code>!=</code>	не равно
<code>@-></code>	первая цифра из десятичного числа
<code><-@</code>	последняя цифра из десятичного числа

Комплексные типы. Переменные комплексных типов предназначены для хранения комплексных типов. Комплексный тип выглядит так:

`({big}){small})c|c32|c64|c80|c128`

Допустимые операции:

<code>+</code>	сложение
<code>+</code> (унарный)	подтверждение знака
<code>-</code>	вычитание
<code>-</code> (унарный)	изменение знака
<code>*</code>	умножение
<code>/</code>	деление
<code>==</code>	равно
<code>!=</code>	не равно

1.3.1.5. Пустой тип

Пустой тип — это тип **empt**. Тип **empt** может быть либо базовым типом указателя, либо типом значения функции. Ни в каких других целях тип **empt** применяться не может. При этом `##empt` = 0, т.е. размер типа **empt** равен нулю.

1.3.1.6. Кортежи

Кортеж — это упорядоченный набор конечного числа элементов, вообще говоря, разных типов. Тип-кортеж выглядит так:

`(:[тип_элемента{,тип_элемента}] :)`

Здесь *тип_элемента* — тип соответствующего элемента кортежа. Этот тип может быть либо именем типа, либо указателем, либо типом указателя на функцию, либо встроенным типом, либо кортежем.

Если для каждого элемента кортежа определена одна и та же операция отношения, то эта операция определена и для всего кортежа.

Кроме того, если x — значение-кортеж, то можно получить значения отдельных элементов этого кортежа. А именно, для получения значения элемента с номером i (элементы кортежа нумеруются слева направо, и нумерация начинается с нуля), нужно написать $x\#i$.

1.3.1.7. Указатели

Указатели содержат адреса ячеек памяти. Тип-указатель определяется так:

@простейшее_определение_типа

Указателю можно присвоить константу **empt**. В этом случае указатель перестаёт указывать на какую бы то ни было ячейку памяти. Указатель можно разыменовывать, то есть получать значение переменной, на которую он указывает. Для разыменования указателя нужно после имени указателя поставить знак @. Разыменовывать можно все указатели, кроме указателей типа **@empt**. Тип переменной, на которую указывает указатель, называется базовым типом указателя.

Указатели можно сравнивать на равенство и неравенство.

Указателю типа **@empt** можно присваивать значение указателя любого типа.

Пример 1.3.1.

```
перем x : z;
      y : @z
      ...
      x := @y + 1;
      ...
```

1.3.1.8. Ссылки

Тип-ссылка выглядит так:

(link|const link)простейшее_определение_типа

1.3.1.9. Типы указателей на функции

Переменные таких типов предназначены для хранения указателей на функции. Тип указателя на функцию выглядит так:

function *сигнатура*

Здесь *сигнатура* определяется следующими формулами в РБНФ:

сигнатура → (*группа_параметров* { ; *группа_параметров* }) : *тип_значения*
группа_параметров → *имя_параметра* { , *имя_параметра* } : *тип_параметра*
имя_параметра → *идентификатор*

1.3.1.10. Массивы

Тип-массив имеет следующий вид:

array [*[выражение]* { , *[выражение]* }] *простейшее_определение_типа*

Если какое-либо из выражений опущено, то по этому измерению массив считается динамическим. Каждое из выражений указывает, сколько значений может принимать соответствующий индекс массива. Каждое выражение должно быть таким, чтобы его можно было вычислить на этапе компиляции. Наименьшее значение каждого индекса равно нулю, а массивы хранятся по строкам.

При этом записи

array[N_0, \dots, N_{m-1}] **array**[N_m, \dots, N_{m+p-1}] T

и

array[$N_0, \dots, N_{m-1}, N_m, \dots, N_{m+p-1}$] T

считаются эквивалентными.

Далее, если тип T определён как

array[N_m, \dots, N_{m+p-1}] T'

то запись

`array[N0, . . . , Nm-1] T`

считается эквивалентной записи

`array[N0, . . . , Nm-1, Nm, . . . , Nm+p-1] T'`

Кроме того, любой тип вида

`array[N0, . . . , Nm-1] T`

где тип `T` эквивалентен типу `empt`, сам эквивалентен типу `empt`.

Все эти преобразования производятся на этапе компиляции.

Пример 1.3.2. Пусть сделаны определения

`const N : z = 128`

`type float = small z;`

`T = array[N] float`

Тогда следующие записи эквивалентны:

- 1) `array[N, N]float`
- 2) `array[N] array[N] float`
- 3) `array[N] T`

Для обращения к элементу массива надо после имени массива в квадратных скобках перечислить индексы нужного элемента.

Пример 1.3.3. Пусть сделаны определения

`const N : z = 128;`

`M : z = 256`

`type float = small z;`

`T1 = array[M] float;`

`T2 = array[N] T1`

`var A : T1;`

`B : T2`

Тогда к элементу массива *A* с индексом 200 нужно обращаться как *A*[200], а к имеющему индекс 107 элементу массива *B* — *B*[107]. Поскольку, в силу сделанных определений, элемент *B*[107] сам является массивом, то для обращения к имеющему индекс 91 элементу массива *B*[107] нужно писать *B*[107][91]. Последняя запись эквивалентна записи *B*[107, 91]. Аналогичные правила действуют и для массивов большей размерности.

Тип элемента массива называется базовым типом массива.

Для массивов определена инфиксная бинарная операция `#`, первым (левым) операндом которой служит имя массива, а вторым (правым) — номер индекса массива, считая слева. Самый левый индекс имеет номер ноль. В результате вычисления данной операции будет получено количество возможных значений указанного вторым операндом индекса. Так происходит, если второй аргумент неотрицателен и меньше количества индексов (с учётом преобразований этапа компиляции). Если же второй аргумент операции `#` либо отрицателен, либо не меньше количества индексов массива, то результат будет равен нулю.

Пример 1.3.4. Пусть сделаны определения

`var A : array[16]r;`

`B : array[9, 11]r;`

`B : array[17, 8, 19]r`

Тогда *A*#0 = 16, *A*#(-3) = 0, *B*#0 = 9, *B*#(-5) = 0, *B*#1 = 11, *B*#2 = 0, *B*#1000 = 0, *A*#1 = 0, *B*[3]#0 = 11, *C*#0 = 17, *C*#1 = 8, *C*#2 = 19, *C*[5]#0 = 8, *C*[5]#1 = 19, *C*[5, 4]#0 = 19.

1.3.1.11. Алгебраические типы данных

Определение алгебраического типа данных имеет следующий вид:

```
опр_алгебр_типа → компонента { . | .компонента }
компонента → опр_структуры | опр_перечисления
опр_структуры → структура имя_структуры { тело_структуры }
опр_перечисления → перечисление имя_перечисления { тело_перечисления }
тело_структуры → [ группа_полей { ; группа_полей } ]
группа_полей → имя_поля { , имя_поля } : тип_поля
тело_перечисления → имя_значения { , имя_значения }
```

1.3.2. Описание переменных

Синтаксис описания переменных:

```
описание_переменных → перем группа_переменных : простейшее_определение_типа
{ ; группа_переменных : простейшее_определение_типа }
группа_переменных → переменная { , переменная }
переменная → имя_переменной
имя_переменной → идентификатор
```

Необязательная звёздочка после имени переменной означает, что переменная доступна из других модулей. Отсутствие звёздочки означает недоступность переменной из других модулей.

1.3.3. Описание констант

Синтаксис описания констант:

```
описание_констант →
const имя_константы : простейшее_определение_типа = значение_константы
{ ; имя_константы : простейшее_определение_типа = значение_константы }
значение_константы → выражение [ значение_константы { , значение_константы } ]
имя_константы → идентификатор
```

1.3.4. Описание функций

Описание алгоритма имеет следующую структуру:

```
описание_функции → [ head | clean ] function имя_функции сигнатура (реализация | ; )
реализация → { { описание | операторы } }
операторы → оператор { ; оператор }
имя_функции → идентификатор
```

Необязательное ключевое слово **head** означает, что выполнение модуля начинается с этой функции. Функций с атрибутом **head** в модуле может быть не более одной.

Необязательное ключевое слово **clean** означает, что функция не имеет побочных эффектов.

1.4. Выражения

Синтаксис выражений с помощью РБНФ можно записать так:

```
выражение → выражение0 [ операция_присваивания выражение ]
выражение0 → выражение1 [ ( ? | ? . ) выражение1 : выражение1 ]
выражение1 → выражение2 { ( | | | | , | ! | | | | , | ^ ) выражение2 }
выражение2 → выражение3 { ( & & | & & . | ! & & | ! & & . ) выражение3 }
выражение3 → { ! } выражение4
выражение4 → выражение5 { ( < | > | < = | > = | = ) выражение5 }
выражение5 → выражение6 { ( | ~ | ^ ) выражение6 }
выражение6 → выражение7 { ( & | ~ & | < < | > > ) выражение7 }
выражение7 → { ~ } выражение8
выражение8 → выражение9 { ( + | + . | - | - . ) выражение9 }
выражение9 → выражение10 { ( * | * . | / | / . | % | % . ) выражение10 }
```

$выражение_{10} \rightarrow выражение_{11} [(** | ** .) выражение_{10}]$
 $выражение_{11} \rightarrow выражение_{12} [\# выражение_{12}]$
 $выражение_{12} \rightarrow \{ (++ | -- | ++ < | -- <) \} выражение_{13}$
 $выражение_{13} \rightarrow [\#] выражение_{14}$
 $выражение_{14} \rightarrow [(+ | -)] выражение_{15}$
 $выражение_{15} \rightarrow [@ | @ @ | \# | \# \# | \# \# \#] выражение_{16}$
 $выражение_{16} \rightarrow (sel | deliv) (имя \{ , выражение \}) | литёра | строка | целое |$
 $r | c | true | false | empty | имя (выражение)$
 $имя_модуля \rightarrow идентификатор$
 $имя \rightarrow идентификатор \{ . идентификатор | @ [выражение \{ , выражение \}] |$
 $([выражение \{ , выражение \}]) \}$

1.5. Операторы

1.5.1. Операторы присваивания

Синтаксис оператора присваивания:

$имя (= | := | | = | | . = | ! | = | ! | | . = | \& \& = | \& \& . = | ! \& \& = | ! \& \& . = | ^ = | | = | \& = | \sim = | \sim \& = | ^ = |$
 $< < = | > > = | + = | - = | * = | / = | \% = | ** = | + . = | - . = | * . = | / . = | \% . = | ** . =) выражение$

Все эти операторы можно разделить на три группы: простой оператор присваивания (=), оператор копирования (:=) и все прочие операторы присваивания. Отличие оператора копирования от оператора присваивания состоит в поведении для динамических массивов: в этом случае оператор присваивания копирует ссылки на значения (точнее, служебные сведения, в которые входят эти ссылки), а оператор копирования копирует сами значения.

1.5.2. Условный оператор

Синтаксис условного оператора:

$if (условие) that \{ \{ описание | операторы \} \}$
 $\{ ins (условие) that \{ \{ описание | операторы \} \} \}$
 $[else \{ \{ описание | операторы \} \}]$

Здесь *условие* — это логическое выражение, а $ins (условие) that \{ \{ описание | операторы \} \}$ является сокращённой формой для $else \{ if (условие) that \{ \{ описание | операторы \} \} \}$.

1.5.3. Оператор выбора

Синтаксис оператора выбора:

$оператор_выбора \rightarrow cho (S) из \{$
 $список_значений_для_ветви : \{ ветвь \}$
 $\{ список_значений_для_ветви : \{ ветвь \} \}$
 $[иначе \{ ветвь_иначе \}] \}$
 $список_значений_для_ветви \rightarrow значение_для_ветви \{ , значение_для_ветви \}$
 $значение_для_ветви \rightarrow выражение [. . выражение]$
 $ветвь \rightarrow \{ описание | операторы \}$
 $ветвь_иначе \rightarrow \{ описание | операторы \}$
 $S \rightarrow выражение$

Оператор выбора определяет выбор и выполнение операторов на основе значения выражения *S*, которое должно быть выражением целочисленного, перечислимого, или символьного типа.

Оператор выполняется так. Сначала вычисляется *S*, а затем выполняется та ветвь (т.е. последовательность операторов), соответствующий которой список значений содержит значение выражения *S*. При этом можно указывать диапазоны значений, а именно, следующим образом: *A*..*B*, где *A* — минимальное значение в диапазоне, *B* — максимальное значение в диапазоне. Например, 1..2000, -7..7. Значения в списках значений должны быть константами, которые можно вычислить на этапе компиляции, и ни одно значение не должно употребляться более одного раза. Если значения выражения *S* нет в списке значений ни для какой ветви, то выполняются операторы *ветвь_иначе*,

если ключевое слово **else** присутствует. Если же ключевого слова **else** нет, то выполнение оператора выбора завершается.

1.5.4. Оператор разбора значения алгебраического типа

Синтаксис оператора разбора:

```
оператор_разбора → anl(S) {  
метка_разбора → {ветвь}  
{метка_разбора → {ветвь}}  
[иначе {ветвь_иначе}]  
метка_разбора → {ид::} ид {..  
ветвь → {описание | операторы}  
ветвь_иначе → {описание | операторы}  
S → выражение
```

1.5.5. Убийца морали

убийца морали → kill(*S*)

Обнуляет объект. Например, если используется на вещественной переменной, то её присваивается значение "0" если же на массиве, то "0" присваивается всем элементам массива.

1.5.6. Переводчик в десятичную систему

переводчик в десятичную систему → propaganda10(*S*)

Переводит число в десятичную систему.

1.5.7. Операторы цикла

Операторы цикла организуют выполнение повторяющихся действий. Всего в языке есть четыре типа операторов цикла: оператор цикла с предусловием (оператор „while“), оператор цикла с постусловием (оператор „rep... while“), оператор „frv rep“, оператор „for“. Опишем каждый из этих операторов.

1.5.7.1. Оператор цикла с предусловием

Оператор цикла с предусловием выглядит так:

```
[+ | имя_цикла : ] while(условие) { {описание | операторы} }
```

Здесь *условие* — это логическое выражение, а *имя_цикла* — идентификатор, являющийся именем цикла. Данный идентификатор можно использовать только в операторе выхода из цикла. Оператор „while“ выполняет тело цикла, пока логическое выражение *условие* остаётся истинным. Истинность этого логического выражения проверяется перед каждым выполнением тела цикла (т.е. операторов *операторы*).

1.5.7.2. Оператор цикла с постусловием

Оператор цикла с постусловием выглядит так:

```
[+ | имя_цикла : ] rep { {описание | операторы} } while(условие)
```

Здесь *условие* — это логическое выражение, а *имя_цикла* — идентификатор, являющийся именем цикла. Данный идентификатор можно использовать только в операторе выхода из цикла. Оператор цикла „rep... while“ выполняет тело цикла, пока логическое выражение *условие* остаётся истинным. Истинность этого логического выражения проверяется после каждого выполнения тела цикла (т.е. операторов *операторы*).

1.5.7.3. Оператор цикла „вечно повторяй“

Оператор „повторяй. . . вечно“ выглядит так:

$[+ | \text{имя_цикла} :] \mathbf{frv\ rep} \{ \{ \text{описание} | \text{операторы} \} \}$

Здесь *имя_цикла* — идентификатор, являющийся именем цикла. Данный идентификатор можно использовать только в операторе выхода из цикла. Оператор „**frv rep**“ выполняется до тех пор, пока из него не будет совершён явный выход — либо с помощью оператора выхода из цикла, либо с помощью оператора возврата из подпрограммы.

1.5.7.4. Оператор цикла „для“

Оператор цикла „**for**“ выглядит так:

$[+ | \text{имя_цикла} :] \mathbf{for\ } v = \text{нач_знач}, \text{кон_зн}[, \text{шаг}] \{ \{ \text{описание} | \text{операторы} \} \}$

Здесь *v* — идентификатор, являющийся именем переменной цикла; *нач_знач* — начальное значение переменной цикла; *кон_знач* — конечное значение переменной цикла; *шаг* — шаг цикла. По умолчанию шаг равен единице. Величины *нач_знач*, *кон_знач* и *шаг* являются выражениями, вычисляемыми до начала цикла. Переменная цикла должна быть символьного, целочисленного или перечислимого типа. Выражения *нач_знач* и *кон_знач* должны иметь тип, совместимый с типом переменной *v*, а выражение *шаг* должно быть целочисленного типа. Менять в теле цикла значение переменной цикла нельзя.

Смысл оператора цикла „**for**“:

```
t1 := нач_знач;
t2 := кон_знач;
t3 := шаг;
if (t3 > 0) that
{
    v := t1;
    while (v <= t2)
    {
        {описание|операторы}
        увелич(v, t3)
    }
}
ins (t3 < 0) that
{
    v := t1;
    while (v >= t2)
    {
        {описание|операторы}
        увелич(v, t3)
    }
}
else
{
    v := t1;
    {описание|операторы}
    пока (t1 != t2)
    {
        {описание|операторы}
    }
}
}
```

1.5.7.5. Оператор цикла „**Iam**“

Оператор „**Iam**. . .“ выглядит так:

[+| *имя_цикла* :] **Iam**{ {*описание*|*операторы*} }

Оператор „**Iam**“ просто есть. Он ничего не делает, но продолжит существовать.

1.5.8. Оператор выхода из цикла

Синтаксис оператора выхода из цикла:

continue [в *имя_цикла*]

Этот оператор совершает выход из цикла с именем *имя_цикла*, если оно указано. Если же нет, то производится выход из текущего цикла.

1.5.9. Оператор выполнения следующей итерации цикла

Синтаксис оператора выполнения следующей итерации из цикла:

ext [из *имя_цикла*]

Оператор **continue** прерывает текущее выполнение цикла и переходит к выполнению следующего шага этого цикла. При этом, цикл **while** возвращается непосредственно к своему условию, а цикл **for** сначала вычисляет выражение инкремента, а затем возвращается к условию.

1.5.10. Оператор возврата из функции

Оператор возврата из подпрограммы совершает выход из функции. Если тип возвращаемого функцией значения — тип **empt**, то выход из подпрограммы выполняется с помощью оператора возврата, имеющего вид **ret**. Если же тип возвращаемого функцией значения не эквивалентен типу **empt**, то возврат выполняется с помощью оператора возврата, имеющего вид **ret** *выражение*, причём тип выражения должен быть совместим с типом возвращаемого функцией значения.