Сообщение о языке программирования Енот

Гуськова И.А.

16 марта 2019г.

Оглавление

Глава 1. Язык

1.1. Лексемы и форма описания синтаксиса

Синтаксис языка программирования Енот 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 →, идёт вторая часть, содержащая определение понятия. В данном описании кроме метасимвола → 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 Енот состоит из лексем. В языке имеется четыре класса лексем:

1) ключевые слова и идентификаторы;

2) числа;

3) знаки операций и разделители;

4) литёры 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:

беззнач	**выбор**	**логик8**	**символ**
беззнач8	**выдел**	**логик16**	**строчка**
беззнач16	**выход**	**логик32**	**структ**
беззнач32	**шапка**	**логик64**	**ссылочка**
беззнач64	**для**	**неправда**	**тип**
беззнач128	**если**	**мал**	**то**
бол	**иначе**	**массив**	**удали**
бесконечно	**инес**	**мод**	**фун**
R	**правда**	**max**	**чистая**
R32	**подсчет**	**min**	**целN**
R64	**рандом**	**ничего**	**целN8**
R80	**IC**	**очищение**	**целN16**
R128	**IC32**	**перемен**	**целN32**
вернуть	**IC64**	**перечисление**	**целN64**
вкл	**IC80**	**повтор**	**целN128**
время	**IC128**	**пока**	
вход	**пост**	**покуда**	
выкл	**логик**	**разбор**	

Примеры идентификаторов: `котенок`, `Гав`, `_s1994`, `Кощей_Бессмертный`, `Черепашка`.

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→*вещественное*i

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 надписал: ...'

'Привет лунатикам!'$0
```

1.1.4. Знаки операций и разделители

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

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

Приведём пример:

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

1.2. Структура программы

Структура программы на языке Рысь 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 $\texttt{T}_1$, может стоять в точках, где $\texttt{T}_1$ еще не известен. Описание типа $\texttt{T}_1$ 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 = мал мал N;
     B = мал N;
     C = N;
     D = бол N
```

Здесь типы `A` и **мал мал n** — 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 само́го значения динамического массива, нужно перед именем переменной поставить знак операции `###`.

1.3.1.1. Логические типы

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

({**бол**}|{**мал**})**логик**

Размер переменной типа **логик** зависит от реализации, но не может превышать размера машинного слова. Также имеются логические типы конкретных размеров, а именно, типы **логик8**, **логик16**, **логик32**, **логик64**, переменные которых имеют размер в 1, 2, 4 и 8 байт соответственно.

Над логическими значениями определены следующие операции:

`||`	логическое „или“ (сокращённое вычисление)
`||.`	логическое „или“ (полное вычисление)
`!||`	логическое „не–или“ (сокращённое вычисление)
`!||.`	логическое „не–или“ (полное вычисление)
`&&`	логическое „и“ (сокращённое вычисление)
`&&.`	логическое „и“ (полное вычисление)
`!&&`	логическое „не–и“ (сокращённое вычисление)
`!&&.`	логическое „не–и“ (полное вычисление)
`^^`	логическое „исключающее или“
`!`	логическое „не“
`==`	равно
`!=`	не равно

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||y	x&&y	x^^y
неправда	**неправда**	**неправда**	**неправда**	**неправда**
неправда	**правда**	**правда**	**неправда**	**правда**
правда	**неправда**	**правда**	**неправда**	**правда**
правда	**правда**	**правда**	**правда**	**неправда**

x	y	(x!||y)≡x||(!y)	(x!&&y)≡x&&(!y)
неправда	**неправда**	**правда**	**неправда**
неправда	**правда**	**неправда**	**неправда**
правда	**неправда**	**правда**	**правда**
правда	**правда**	**неправда**	**неправда**

x	!x
неправда	**правда**
правда	**неправда**

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`=**правда**) то
    выдать `A`
иначе
    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`=**правда**) 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:

символ

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:

строчка

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:

({**бол**}|{**мал**})|**беззнач**|**n**|**беззнач8**|**беззнач16**|**беззнач32**|**беззнач64**|
беззнач128|**целN8**|**целN16**|**целN32**|**целN64**|**целN128**)

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

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

({**бол**}|{**мал**})**R**|**R32**|**R64**|**R80**|**R128**

Допустимые операции:

`+.`	сложение
`+` (унарный)	подтверждение знака
`-.`	вычитание
`-` (унарный)	изменение знака
`*.`	умножение
`/.`	деление
`%.`	вещественный остаток от деления
`**.`	вещественное возведение в степень
`<`	меньше
`>`	больше
`<=`	меньше или равно
`>=`	больше или равно
`==`	равно
`!=`	не равно

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

({**бол**}|{**мал**})**IC**|**IC32**|**IC64**|**IC80**|**IC128**

Допустимые операции:

`+.`	сложение
`+` (унарный)	подтверждение знака
`-.`	вычитание
`-` (унарный)	изменение знака
`*.`	умножение
`/.`	деление
`==`	равно
`!=`	не равно

1.3.1.5. Пустой тип

Пустой тип — это тип **ничего**. Тип **ничего** может быть либо базовым типом указателя, либо типом значения функции. Ни в каких других целях тип **ничегоо** применяться не может. При этом **##ничего**= 0, т.е. размер типа **ничего** 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 **ничего**. 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 **@ничего**. Тип переменной, на которую указывает указатель, называется базовым типом указателя.

Указатели можно сравнивать на равенство и неравенство.

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

Пример 1.3.1.

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

1.3.1.8. Ссылки

Тип–ссылка выглядит так:

(**ссылочка**|**конст ссылочка**)*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:

функция *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:

массив[[*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

$$\textbf{массив}[\mathtt{N}_0,\ldots,\mathtt{N}_{m-1}]\ \textbf{массив}[\mathtt{N}_m,\ldots,\mathtt{N}_{m+p-1}]\ \mathtt{T}$$

и

$$\textbf{массив}[\mathtt{N}_0,\ldots,\mathtt{N}_{m-1},\mathtt{N}_m,\ldots,\mathtt{N}_{m+p-1}]\ \mathtt{T}$$

считаются эквивалентными.

Далее, если тип `T` определён как

$$\textbf{массив}[\mathtt{N}_m,\ldots,\mathtt{N}_{m+p-1}]\ \mathtt{T}'$$

то запись

$$\textbf{массив}[\mathtt{N}_0,\ldots,\mathtt{N}_{m-1}]\ \mathtt{T}$$

считается эквивалентной записи

$$\textbf{массив}[\mathtt{N}_0,\ldots,\mathtt{N}_{m-1},\mathtt{N}_m,\ldots,\mathtt{N}_{m+p-1}]\ \mathtt{T}'$$

Кроме того, любой тип вида

массив[N_0,...,N_{m-1}] T

где тип T эквивалентен типу **ничего**, сам эквивалентен типу **ничего**.

Все эти преобразования производятся на этапе компиляции.

Пример 1.3.2. *Пусть сделаны определения*

```
конст N : целN = 128
тип   float = мал вещR;
      T = массив[целN] float
```

Тогда следующие записи эквивалентны:

1) **массив**[N, N]float
2) **массив**[N] **массив**[N] float
3) **массив**[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*

```
пост N : целN = 128;
      M : целN = 256

тип   float = маленькое вещR;
      T1 = массив[M] float;
      T2 = массив[N] T1

перем 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*

```
перем A : массив[16]вещR;
      B : массив[9, 11]вещR;
      B : массив[17, 8, 19]вещR
```

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_струкm*|*опр_перечисления*
опр_струкm→**струкm** *имя_струкm* {*тело_струкm*}
опр_перечисления→**перечисление** *имя_перечисления* {*тело_перечисления*}
тело_струкm→[*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→
пост *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→[**шапка**|**чистая**] **фун** *имя_функции сигнатура*(*реализация*|;)

реализация→{ {*описание*|*операторы*} }

операторы→*оператор*{; *оператор*}

имя_функции→*идентификатор*

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

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

1.4. Выражения

Синтаксис выражений с помощью РБНФ можно записать так:

выражение→*выражение*$_0$[*операция_присваивания выражение*]

выражение$_0$ →*выражение*$_1$[(?|?.)*выражение*$_1$: *выражение*$_1$]

выражение$_1$→*выражение*$_2${(|||||.|!|||!||.|^^)*выражение*$_2$}

выражение$_2$→*выражение*$_3${(&&|&&.|!&&|!&&.)*выражение*$_3$}

выражение$_3$→{!}*выражение*$_4$

выражение$_4$→*выражение*$_5${(<|>|<=|>=|==|!=)*выражение*$_5$}

выражение$_5$→*выражение*$_6${(||~||^)*выражение*$_6$}

выражение$_6$→*выражение*$_7$6{(&|~&|<<|>>)*выражение*$_7$6}

выражение$_7$→{~}*выражение*$_8$

выражение$_8$→*выражение*$_9${(+|+.|-|-.)*выражение*$_9$}

выражение$_9$→*выражение*$_{10}$*{(*|*.|/|/.|%|%.)*выражение*$_{10}$}

выражение$_{10}$→*выражение*$_{11}$[(**|**.)*выражение*$_{10}$]

выражение$_{11}$→*выражение*$_{12}$[#*выражение*$_{12}$]

выражение$_{12}$→{(++|--|++<|--<)}*выражение*$_{13}$

выражение$_{13}$→[#]*выражение*$_{14}$

выражение$_{14}$→[(+|-)]*выражение*$_{15}$

выражение$_{15}$→[@|@@|##|###]*выражение*$_{16}$

выражение$_{16}$→(**выдел**|**очистить**)(*имя*{,*выражение*})|*литёра*|*строчка*|*целое*|*вещественное*|*комплексное*|**правда**|**неправда**|**ничего**|*имя*|(*выражение*)

имя_модуля→*идентификатор*

имя→*идентификатор*{.*идентификатор*|@|[*выражение*{,*выражение*}]|([*выражение*{,*выражение*}])}

1.5. Операторы

1.5.1. Операторы присваивания

Синтаксис оператора присваивания:

имя(=|:=|||=|||.=|!||=|!|||.=|&&=|&&.=|!&&=|!&&.=|^^=||=|&=|~|=|~&=|^=|
<<=|>>=|+=|-=|*=|/=|%=|**=|+.=|-.=|*.=|/.=|%.=|**.=)*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:
если(*условие*)**то**{ {*описание*|*операторы*} }
{**инес**(*условие*)**то**{ {*описание*|*операторы*} }}
[**иначе**{ {*описание*|*операторы*} }]

Здесь *условие* — это логическое выражение, а **инес**(*условие*)**то**{ {*описание*|*операторы*} } является сокращённой формой для **иначе**{ **если**(*условие*)**то**{ {*описание*|*операторы*} } }.

1.5.3. Оператор выбора

Синтаксис оператора выбора:

оператор_выбора→**выбери**(*S*)**из**{
список_значений_для_ветви : {*ветвь*}
{*список_значений_для_ветви* : {*ветвь*}}
[**иначе**{*ветвь_иначе*}]}

список_значений_для_ветви→*значение_для_ветви*{, *значение_для_ветви*}
значение_для_ветви→*выражение*[..*выражение*]
ветвь→{*описание*|*операторы*}
ветвь_иначе→{*описание*|*операторы*}
S→*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 **иначе** присутствует. Если же ключевого слова **иначе** нет, то выполнение оператора выбора завершается.

1.5.4. Оператор разбора значения алгебраического типа

Синтаксис оператора разбора:

оператор_разбора→**разбор**(*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. Оператор кто

Синтаксис оператора **„кто“**:

оператор_кто →**кто**(имена переменных | тип переменных)

Оператор **„кто“** возвращает переменные, созданные во время программы, с указанием их типов и значений. При введении имена переменных возвращает только типы и значения указанных переменных. При введении типа переменных, возвращает все переменные, указанного типа. При отсутствии условий возвращает все переменные, созданные во время работа программы. Используя связку **„кто подсчет “** к возращаемым переменным добавляется их колличество, а так же обьем памяти, занимаемый на диске.

1.5.6. Оператор удали

оператор_удали →**удали**(имена переменных | тип переменных)

Оператор **„удали“** удаляет переменные, созданные во время программы, с указанием их типов и значений. При введении имена переменных удаляет только типы и значения указанных переменных. При введении типа переменных, удаляет все переменные, указанного типа. При отсутствии условий все переменные, созданные во время работа программы.

1.5.7. Оператор время

оператор_время →**„время вкл“**
оператор_время →**„время выкл“**

Оператор **„время“** засекает время от начала работы „время вкл„ до конца „время выкл„ и возращает полученный результат в секундах.

1.5.8. Оператор рандом

оператор_рандом →**„рандом“**(промежуток)

Оператор **„рандом“** произвольно выбирает значение из указанного числового промежутка и возращает полученный результат. textcolorGreenпромежуток - упорядоченный по возрастанию целочисленный массив, где первому элементу соответсвует минимальное значение промежутка, а последнему максимальное.

1.5.9. Оператор min

Синтаксис оператора **„min“**:

оператор_min →**min**(имя массива)

Оператор **„min“** возращает минимальное значение, содержащеся в массиве числового типа, в противном случае выдает ошибку.

1.5.10. Оператор max

Синтаксис оператора **„max“**:

оператор_max →**max**(имя массива)

Оператор **„max“** возращает максимальное значение, содержащеся в массиве числового типа, в противном случае выдает ошибку.

1.5.11. Операторы цикла

Операторы цикла организуют выполнение повторяющихся действий. Всего в языке есть четыре типа операторов цикла: оператор цикла с предусловием (оператор **„пока“**), оператор цикла с постусловием (оператор **„повторяй. . . пока“**), оператор **„вечно повторяй“**, оператор **„для“**. Опишем каждый из этих операторов.

1.5.11.1. Оператор цикла с предусловием

Оператор цикла с предусловием выглядит так:

[+|*имя_цикла*:]**пока**(*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 **„пока“** выполняет тело цикла, пока логическое выражение *условие* остаётся истинным. Истинность этого логического выражения проверяется перед каждым выполнением тела цикла (т.е. операторов *операторы*).

1.5.11.2. Оператор цикла с постусловием

Оператор цикла с постусловием выглядит так:

[+|*имя_цикла*:]**повторяй**{ {*описание*|*операторы*} }**покуда**(*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 **„повторяй. . . пока“** выполняет тело цикла, пока логическое выражение *условие* остаётся истинным. Истинность этого логического выражения проверяется после каждого выполнения тела цикла (т.е. операторов *операторы*).

1.5.11.3. Оператор цикла „бесконечно повторяй“

Оператор **„повторяй. . . бесконечно“** выглядит так:

[+|*имя_цикла*:]**бесконеечно повторяй**{ {*описание*|*операторы*} }

Здесь *имя_цикла* — идентификатор, являющийся именем цикла. Данный идентификатор можно использовать только в операторе выхода из цикла. Оператор **„вечно повторяй“** 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.

1.5.11.4. Оператор цикла „для“

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

[+|*имя_цикла*:]**для** *v* = `нач_знач`, `кон_зн`[, `шаг`] { {*описание*|*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 **„для“**:

```
t1 := нач_знач;
t2 := кон_знач;
t3 := шаг;
если(t3 > 0)то
{
    v := t1;
    пока(v <= t2)
    {
        {описание|операторы}
        увелич(v, t3)
    }
}
```

```
инес(t3 < 0)то
{
    v := t1;
    пока(v >= t2)
    {
        {описание|операторы}
        увелич(v, t3)
    }
}
иначе
{
    v := t1;
    {описание|операторы}
    пока(t1 != t2)
    {
        {описание|операторы}
    }
}
}
```

1.5.12. Оператор выхода из цикла

Синтаксис оператора выхода из цикла:

выход [**из** *имя_цикла*]

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

1.5.13. Оператор возврата из функции

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

1.5.14. Оператор входа в цикл

Синтаксис оператора входа в цикл:

вход [**в** *имя_цикла*]

Этот оператор совершает вход из в цикла с именем *имя_цикла*.

Глава 2. Стандартная библиотека

2.1. Математические функции

2.2. Ввод-вывод