

# Сообщение о языке программирования Лиса

Сарыкова А.А.

16 марта 2019г.

# Оглавление

<b>1</b>	<b>Язык</b>	<b>2</b>
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 . . . . .	8
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	Множества . . . . .	10
1.3.1.12	Алгебраические типы данных . . . . .	12
1.3.2	Описание переменных . . . . .	12
1.3.3	Описание констант . . . . .	12
1.3.4	Описание функций . . . . .	13
1.4	Выражения . . . . .	13
1.5	Операторы . . . . .	13
1.5.1	Операторы присваивания . . . . .	13
1.5.2	Условный оператор . . . . .	13
1.5.3	Оператор выбора . . . . .	14
1.5.4	Оператор разбора значения алгебраического типа . . . . .	14
1.5.5	Операторы цикла . . . . .	14
1.5.5.1	Оператор цикла с предусловием . . . . .	14
1.5.5.2	Оператор цикла с постусловием . . . . .	15
1.5.5.3	Оператор цикла „бескон повт“ . . . . .	15
1.5.5.4	Оператор цикла „для“ . . . . .	15
1.5.6	Оператор выхода из цикла . . . . .	16
1.5.7	Оператор возврата из функции . . . . .	16
<b>2</b>	<b>Стандартная библиотека</b>	<b>17</b>
2.1	Математические функции . . . . .	17
2.2	Ввод-вывод . . . . .	17

# Глава 1. Язык

## 1.1. Лексемы и форма описания синтаксиса

Синтаксис языка программирования Лиса 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 Лиса 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:

<b>бескон</b>	<b>выдел</b>	<b>логик16</b>	<b>строка</b>
<b>бз</b>	<b>выход</b>	<b>логик32</b>	<b>структ</b>
<b>бз8</b>	<b>глав</b>	<b>логик64</b>	<b>ссылка</b>
<b>бз16</b>	<b>для</b>	<b>ложь</b>	<b>тип</b>
<b>бз32</b>	<b>если</b>	<b>мал</b>	<b>то</b>
<b>бз64</b>	<b>иначе</b>	<b>массив</b>	<b>фун</b>
<b>бз128</b>	<b>иначеесли</b>	<b>множество</b>	<b>чистая</b>
<b>бол</b>	<b>истина</b>	<b>модуль</b>	<b>цел</b>
<b>в</b>	<b>компл</b>	<b>ничего</b>	<b>цел8</b>
<b>вещ</b>	<b>компл32</b>	<b>очищ</b>	<b>цел16</b>
<b>вещ32</b>	<b>компл64</b>	<b>перем</b>	<b>цел32</b>
<b>вещ64</b>	<b>компл80</b>	<b>перечисление</b>	<b>цел64</b>
<b>вещ80</b>	<b>компл128</b>	<b>повт</b>	<b>цел128</b>
<b>вещ128</b>	<b>конст</b>	<b>пока</b>	
<b>возврат</b>	<b>логик</b>	<b>разбор</b>	
<b>выбор</b>	<b>логик8</b>	<b>симв</b>	

Примеры идентификаторов: крокодил, Гена, \_s1024, Баба\_Яга, Чебурашка.

### 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

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

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 Лиса 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 *T<sub>1</sub>*, может стоять в точках, где *T<sub>1</sub>* еще не известен. Описание типа *T<sub>1</sub>* 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.
- 5) определение типа-множества.

Приведём пример определения типов:

```
тип  А = мал мал цел;
      В = мал цел;
      С = цел;
      D = бол цел
```

Здесь типы **А** и **мал мал цел** — 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:

({**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	$x    y$	$x \& y$	$x \wedge y$
ложь	ложь	ложь	ложь	ложь
ложь	истина	истина	ложь	истина
истина	ложь	истина	ложь	истина
истина	истина	истина	истина	ложь

x	y	$(x !    y) \equiv x    (!y)$	$(x ! \& y) \equiv x \& (!y)$
ложь	ложь	истина	ложь
ложь	истина	ложь	ложь
истина	ложь	истина	истина
истина	истина	ложь	ложь

x	!x
ложь	истина
истина	ложь

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 ( $S$ =**истина**) 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 ( $S$ =**истина**) 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:

**симв**

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:

**строка**

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}))|бз|цел|бз8|бз16|бз32|бз64|  
бз128|цел8|цел16|цел32|цел64|цел128)

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}))|вещ|вещ32|вещ64|вещ80|вещ128

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})компл|компл32|компл64|компл80|компл128`

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 **ничего** 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 @ничего можно присваивать значение указателя любого типа.

#### Пример 1.3.1.

```
перем x : цел;
    у : @цел
...
x := @у + 1;
...
```

#### 1.3.1.8. Ссылки

Тип-ссылка выглядит так:

(ссылка|конст ссылка)*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:

фун *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

массив[N<sub>0</sub>, ..., N<sub>m-1</sub>] массив[N<sub>m</sub>, ..., N<sub>m+p-1</sub>] T

и

массив[N<sub>0</sub>, ..., N<sub>m-1</sub>, N<sub>m</sub>, ..., N<sub>m+p-1</sub>] T

считаются эквивалентными.

Далее, если тип T определён как

массив[N<sub>m</sub>, ..., N<sub>m+p-1</sub>] T'

то запись

массив[N<sub>0</sub>, ..., N<sub>m-1</sub>] T

считается эквивалентной записи

массив[N<sub>0</sub>, ..., N<sub>m-1</sub>, N<sub>m</sub>, ..., N<sub>m+p-1</sub>] T'

Кроме того, любой тип вида

**массив**[ $N_0, \dots, N_{m-1}$ ] **Т**

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

Все эти преобразования производятся на этапе компиляции.

**Пример 1.3.2.** Пусть сделаны определения

**конст** **N** : **цел** = 128

**тип** **float** = **мал вещь**;

**Т** = **массив**[**N**] **float**

Тогда следующие записи эквивалентны:

1) **массив**[**N**, **N**] **float**

2) **массив**[**N**] **массив**[**N**] **float**

3) **массив**[**N**] **Т**

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

**Пример 1.3.3.** Пусть сделаны определения

**конст** **N** : **цел** = 128;

**М** : **цел** = 256

**тип** **float** = **мал вещь**;

**Т1** = **массив**[**M**] **float**;

**Т2** = **массив**[**N**] **Т1**

**перем** **A** : **Т1**;

**В** : **Т2**

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] **вещ**;

**В** : **массив**[9, 11] **вещ**;

**С** : **массив**[17, 8, 19] **вещ**

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. Множества

Множеством назовём совокупность объектов, которое может состоять из произвольного числа элементов одного типа. Тип-множество имеет следующий вид:

**множество**( (*элемент множества*) { (*элемент множества*) } ) *простейшее\_определение\_типа*

Каждый элемент множества должен быть таким, чтобы его можно было вычислить на этапе компиляции. В множестве не должно быть повторяющихся элементов.

Число элементов множества не может превышать число значений, входящих в простейшее определение типа.

**Присваивание значений.** Объектами множественного типа являются конкретные множества. Обращаться к ним можно с помощью переменных, вводимых обычным путём.

**Пример 1.3.5.**

**тип** A = множество(jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);

**перем** m : A;

В данном случае значение переменной m может быть произвольная совокупность названий месяца.

Множество задаётся в виде списка элементов, заключённого в круглые скобки. В скобках может не быть элементов, может стоять только один элемент или может присутствовать несколько элементов, разделённых запятыми. В качестве элемента может выступать константа, переменная, выражение, значение которого принадлежит простейшему определению типа, а также пара элементов, разделённых двумя точками, т.е. интервал значений.

Следовательно, можно использовать инструкции присваивания следующих видов:

m := (); (\* пустое множество \*)

m := (jan); (\* множество включает только один элемент \*)

m := (jan .. mar, aug, nov .. dec); (\* в множество входят элементы JAN, FEB, MAR, AUG, NOV, DEC \*)

**Операции над множествами.** Средства, предназначенные для создания множеств и присваивания им значений, не позволяют в полной мере использовать возможности данных множественного типа. В связи с этим введены такие операции:

1) **Объединение**

В объединение множеств  $L + R$  входят элементы, которые являются членами хотя бы одного из множеств L или R.

2) **Пересечение**

В пересечение множеств  $L * R$  входят элементы, принадлежащие и множеству L, и множеству R.

3) **Разность**

В разность множеств  $L - R$  входят элементы, которые являются членами множества L и не являются членами множества R.

**Пример 1.3.6.** Пусть сделаны определения

**тип** C = множество(anna, elena, conya, gelya, liza, mary, sveta, sasha);

**перем** A, B : C;

*и две конструкции присваивания:*

A := (anna, conya, mary .. sasha);

B := (anna, liza, mary, sveta);

В результате выполнения двум множествам A и B (в множество A включены имена спортивных девушек, а в множество B – имена девушек, которые занимаются рисованием) присваиваются конкретные значения.

Тогда результатом  $A + B$  является множество имён тех девушек, которые или занимаются спортом, или хороши в искусстве, или и спортивные, и рисуют. В это множество войдут **anna, conya, liza, mary, sveta, sasha.**

Результатом  $A * B$  является множество, в которое входят имена спортивных и творческих девушек, т.е. **anna, mary, sveta.**

Наконец, результатом  $A - B$  является множество имён девушек, которые спортивные, но не занимаются рисованием. Это **conya и sasha.**

Оператор **в** позволяет определить, принадлежит элемент множеству или нет. Первым операндом, расположенным слева от слова **в**, является значением простейшего определения типа, а вторым операндом, стоящим справа, имя множества. Результатом являются значения типа boolean: true – если значение является элементом множества; false – если не является. Следовательно,

liza в B  
даёт в результате true, а  
gelya в (A + B)  
даёт false

При работе с множествами можно использовать операторы сравнения. Операторы = и < > позволяют проверить, равны два множества или нет. Операторы <= и >= называются операторами включения. С их помощью легко определить, является ли одно множество подмножеством другого. Например, отношение

(anna, sveta) <= (A + B)

истинно, поскольку все элементы, стоящие слева, содержатся в множестве, расположенном справа.

Расположим операторы, предназначенные для работы с множествами, в порядке убывания приоритета:

- 1) \*
- 2) +, -
- 3) в, =, < >, <=, >=

В группы объединены операторы равного приоритета. Последовательность выполнения операторов одного приоритета определяется порядком их появления в выражении. Для изменения порядка выполнения операторов используются круглые скобки.

#### 1.3.1.12. Алгебраические типы данных

Определение алгебраического типа данных имеет следующий вид:

```
опр_алгебр_типа → компонента { . | .компонента }
компонента → опр_структуры | опр_перечисления
опр_структуры → структ 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 →
конст 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 → [глав | чистая] фун имя_функции сигнатура(реализация|);  
реализация → { {описание | операторы} }  
операторы → оператор { ; оператор }  
имя_функции → идентификатор
```

Необязательное ключевое слово **глав** означает, что выполнение модуля начинается с этой функции. Функций с атрибутом **глав** 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 → 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  
выражение10 → выражение11 [ ( ** | **. )} выражение10  
выражение11 → выражение12 [ #} выражение12  
выражение12 → { ( ++ | -- | ++< | --< )} выражение13  
выражение13 → [ #} выражение14  
выражение14 → [ ( + | - )} выражение15  
выражение15 → [ @ | @@ | ## | ### } выражение16  
выражение16 → ( выдел | очис ) ( имя { , выражение } ) | литёра | строка | целое |  
вещественное | комплексное | истина | ложь | 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} }
{иначеесли(условие)то{ {описание|операторы} }}
[иначе{ {описание|операторы} }]

```

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

### 1.5.3. Оператор выбора

Синтаксис оператора выбора:

```

оператор_выбора→выбор(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. 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 „повт. . . пока“), оператор „бескон повт“, оператор „для“. Опишем каждый из этих операторов.

#### 1.5.5.1. 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*).

#### 1.5.5.2. Оператор цикла с постусловием

Оператор цикла с постусловием выглядит так:

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

#### 1.5.5.3. Оператор цикла „бескон повт“

Оператор „повт. . . бескон“ выглядит так:

[+| *имя\_цикла* :] бескон повт{ {*описание*|*операторы*} }

Здесь *имя\_цикла* — идентификатор, являющийся именем цикла. Данный идентификатор можно использовать только в операторе выхода из цикла. Оператор „бескон повт“ 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.

#### 1.5.5.4. 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)
    }
}
иначеесли (t3 < 0) то
{
    v := t1;
    пока (v >= t2)
    {
        {описание|операторы}
        увелич(v, t3)
    }
}
```



```

}
иначе
{
     $v := t_1;$ 
    {описание|операторы}
    пока( $t_1 \neq t_2$ )
    {
        {описание|операторы}
    }
}
}

```

### 1.5.6. Оператор выхода из цикла

Синтаксис оператора выхода из цикла:

**выход** [из *имя\_цикла*]

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

### 1.5.7. 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 **возврат**. Если же тип возвращаемого функцией значения не эквивалентен типу **ничего**, то возврат выполняется с помощью оператора возврата, имеющего вид **возврат** *выражение*, причём тип выражения должен быть совместим с типом возвращаемого функцией значения.

## Глава 2. Стандартная библиотека

### 2.1. Математические функции

### 2.2. Ввод-вывод