

Сообщение о языке программирования Li

Кашневич А.А.

17 марта 2019г.

Оглавление

1	Язык	2
1.1	Лексемы и форма описания синтаксиса	2
1.1.1	Ключевые слова и идентификаторы	2
1.1.2	Числа	2
1.1.3	Литёры и строки	3
1.1.4	Знаки операций и разделители	3
1.2	Структура программы	3
1.3	Описания	3
1.3.1	Описание типов	4
1.3.1.1	Логические типы	4
1.3.1.2	Символьные типы	5
1.3.1.3	Строковые типы	5
1.3.1.4	Числовые типы	5
1.3.1.5	Кортежи	6
1.3.1.6	Указатели	6
1.3.1.7	Ссылки	6
1.3.1.8	Типы указателей на функции	6
1.3.1.9	Массивы	7
1.3.1.10	Алгебраические типы данных	7
1.3.2	Описание переменных	8
1.3.3	Описание констант	8
1.3.4	Описание функций	8
1.4	Выражения	8
1.5	Операторы	9
1.5.1	Операторы присваивания	9
1.5.2	Оператор очистки памяти delete	9
1.5.3	Условный оператор	9
1.5.4	Операторы цикла	9
1.5.4.1	Оператор цикла while	9
1.5.4.2	Оператор цикла for	9
1.5.5	Оператор выхода из цикла break	10

Глава 1. Язык

1.1. Лексемы и форма описания синтаксиса

Синтаксис языка программирования Lu с помощью формул, состоящих из двух частей: первая часть содержит имя определяемого понятия, а затем, после метасимвола `|`, идёт вторая часть, содержащая определение понятия. Также кроме метасимвола `|` будут также использоваться следующие метасимволы:

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

Текст программы на языке Lu состоит из лексем. В языке имеется три класса лексем:

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

Лексемы — это предопределённые константы, идентификаторы и знаки операций, формирующиеся из допустимых символов. В свою очередь, лексемы являются частью выражений, из которых в дальнейшем составляются операторы. Никакая лексема не может разбиваться на части пробелами или комментариями. Опишем каждый класс лексем.

1.1.1. Ключевые слова и идентификаторы

Идентификатор — это слово, в состав которого могут входить символы латинского алфавита, цифры и знак подчёркивания. Идентификатор должен начинаться с буквы. Прописные и строчные буквы считаются различными. Идентификатор не может совпадать ни с каким ключевым словом. Ключевые слова языка записываются как строчными, так и заглавными буквами. Ниже приведён список ключевых слов:

big	full32	perem
break	full64	small
const	full128	sign
clean	good	string
delete	If	time
else	log	type
example	log8	then
false	log16	true
for	log32	usl
full8	log64	while
full16	main	

Примеры идентификаторов: `Sun_52`, `car`, `f567`.

1.1.2. Числа

Числа — это беззнаковые целые числа. Знаковые константы — это беззнаковые константы, к которым применена унарная операция смены знака (операция „—“). Запись:

число|целое

1.1.3. Литёры и строки

Литёра — это либо произвольный символ, заключённый в двойные (") кавычки. Строка — это либо последовательность из ноль или более символов, заключённых в кавычки. Открывающая кавычка должна совпадать с закрывающей. Если в строке требуется записать кавычку, совпадающую с открывающей, то кавычка должна быть продублирована.

Примеры строк:

" — пустая строка

"Good day!"

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 }.

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 аналогичным именем (вне зависимости с заглавной или строчной буквы начинается имя), 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) Аналогичный идентификатор, начинающийся с заглавной (или прописной) буквы также не может быть объявлен в блоке дважды.
- 3) На объект можно сослаться только в его области видимости.
- 4) Описание типа *T*, содержащее ссылки на другой тип *T*₁, может стоять в точках, где *T*₁ еще не известен. Описание типа *T*₁ должно следовать далее в том же блоке, в котором локализован *T*.
- 5) Заголовок функции может быть приведён до того, как будет дано полное определение.

1.3.1. Описание типов

Описание типов выглядит так:

type *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 позже.

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) перечисляемый тип;

К простейшим определениям типов, задаваемым пользователем, относятся:

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

Чтобы узнать размер переменной любого из типов необходимо перед именем типа или переменной этого типа 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: **true** или **false**. Логический тип выглядит так:

[{big}/{small}]log

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“
&&	логическое „и“
!//	логическое „не-или“
!&&	логическое „не-и“
!	логическое „не“
==	равно
!=	не равно

Все логические типы попарно совместимы между собой.

Приведём таблицы истинности логических операций.

x	y	x//y	x&& y
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	true

x	y	(x!//y)	(x!&&y)
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

x	!x
false	true
true	false

1.3.1.2. Символьные типы

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

sign

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 объединения строк, обозначаемая знаком „+“.

1.3.1.4. Числовые типы

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

[{big}/{small}]full

Имеются логические типы конкретных размеров, а именно, типы **full8**, **full16**, **full32**, **full64**, **full128**.

Допустимые операции:

- + целочисленное сложение
- целочисленное вычитание
- ++ 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“
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
==	равно
!=	не равно

1.3.1.5. 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-кортежа, 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 $x[i]$.

1.3.1.6. Указатели

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

```
@простейшее_определение_типа
```

Указателю можно присвоить константу **ничто**. 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 **Сничто**. Тип переменной, на которую указывает указатель, называется базовым типом указателя.

Указатели можно сравнивать на равенство и неравенство.

Указателю типа **Сничто** можно присваивать значение указателя любого типа.

1.3.1.7. Ссылки

Тип-ссылка выглядит так:

```
[ссылка/конст ссылка]простейшее_определение_типа
```

1.3.1.8. 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:

сигнатура[(*группа_параметров*{;*группа_параметров*})]:*тип_значения*
группа_параметров|*имя_параметра*{,*имя_параметра*}:*тип_параметра*
имя_параметра|*идентификатор*

1.3.1.9. 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_0, \dots, N_{m-1}] **массив**[N_m, \dots, N_{m+p-1}] T

и

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

считаются эквивалентными.

Далее, если тип T определён как

массив[N_m, \dots, N_{m+p-1}] T'

то запись

массив[N_0, \dots, N_{m-1}] T

считается эквивалентной записи

массив[$N_0, \dots, N_{m-1}, N_m, \dots, N_{m+p-1}$] T'

Кроме того, любой тип вида

массив[N_0, \dots, N_{m-1}] T

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

Все эти преобразования производятся на этапе компиляции.

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

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.

1.3.1.10. 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 | perem группа_переменных : простоее_определение_типа  
{; группа_переменных : простоее_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 : простоее_определение_типа = значение_константы  
{; имя_константы : простоее_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 | (main/clean) функция имя_функции сигнатура [реализация];  
реализация | { {описание/операторы} }  
операторы | оператор {; оператор}  
имя_функции | идентификатор
```

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

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

1.4. Выражения

Синтаксис выражений 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 → (выделение | освобождение) (имя {, выражение} ) | литерал | строка | целое |  
вещественное | комплексное | истина | ложь | ничто | 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. Оператор очистки памяти delete

Синтаксис оператор очистки памяти delete:

```
delete[имя массива/имя переменной]
```

1.5.3. Условный оператор

Синтаксис условного оператора:

```
if[условие]then{ {описание/операторы} }  
{usl[условие]then{ {описание/операторы} }}  
(else{ {описание/операторы} })
```

Здесь *условие* — это логическое выражение, а `usl(условие)then{ {описание|операторы} }` является сокращённой формой для `else{ if(условие)then{ {описание|операторы} } }`.

1.5.4. Операторы цикла

Операторы цикла организуют выполнение повторяющихся действий. Всего в языке есть два типа операторов цикла: `while` и `for`. Опишем каждый из этих операторов.

1.5.4.1. Оператор цикла while

Оператор цикла `while` выглядит так:

```
while[условие]{ {описание/операторы} }
```

Здесь *условие* — это логическое выражение. 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*).

1.5.4.2. Оператор цикла for

Оператор цикла `for` выглядит так:

```
for[a = нач_знач, a = кон_зн, шаг]{ {описание/операторы} }
```

Здесь *a* — 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 *a*, а выражение *шаг* должно быть целочисленного типа. Менять в теле цикла значение переменной цикла нельзя.

1.5.5. Оператор выхода из цикла break

Синтаксис оператора выхода из цикла break:

[Цикл]{ break }

Когда оператор break выполняется в цикле, то досрочно прерывается исполнение оператора цикла, и управление передаётся следующему оператору после цикла.