Сообщение о языке программирования Lu

Кашневич А.А.

17 марта 2019г.

Оглавление

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:

area
Arh2
Arh4
Arh6
ast
ast1
ast3
ast5
ast7
better
big
define
Difficulty1
Difficulty3
Difficulty5
Difficulty7
else
example
false
full1
full3
full5
full7
good
Hello
If
negative
question
question2
question4
question4
small
sign
square
string
triangle
true
Wrong
union
Zero

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 индитификатор, 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 $\texttt{T}_1$, 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`.

5) 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 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;

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 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: **true** или **false**. Логический тип выглядит так:

[{**big**}/{**small**}]**ast**

Имеются логические типы конкретных размеров, а именно, типы **ast1**, **ast3**, **ast5**, **ast7**, 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 **full1**, **full3**, **full5**, **full7**.

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 **`@ничто`**. 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.

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

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

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

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

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→**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→[**головная**|**чистая**] **функция** *имя_ функции сигнатура*(*реализация*|;)
реализация→{ {*описание*|*операторы*} }
операторы→*оператор*{; *оператор*}
имя_ функции→*идентификатор*

Необязательное ключевое слово **головной** означает, что выполнение модуля начинается с этой функции. Функций с атрибутом **головной** 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$_{16}$→(**выделение**|**освобождение**)(*имя*{,*выражение*})|*литёра*|*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. 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. 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.

1.5.5.1. 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*:)**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*).

1.5.5.3. Оператор цикла „вечно повторяй“

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

(+|*имя_цикла*:)**вечно 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.

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

1.5.6. Оператор выхода из цикла

Синтаксис оператора выхода из цикла:

выйди (**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.

1.5.7. Оператор возврата из функции

Оператор возврата из подпрограммы совершает выход из функции. Если тип возвращаемого функцией значения — тип **ничто**, то выход из подпрограммы выполняется с помощью оператора

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