федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский УНИВЕРСИТЕТ информационных технологий, механики и оптики

**Факультет программной инженерии и компьютерной техники**

**ЛАБОРАТОРНАЯ РАБОТА № 2**

**ПО ДИСЦИПЛИНЕ «ОСНОВЫ РАЗРАБОТКИ КОМПИЛЯТОРОВ»**

Выполнил: Чудаков М.И.

Группа: P3319

Преподаватели: Лаздин А.В.

Логинов И.П.

Санкт-Петербург 2018/2019

**Вариант 19**

**Цель**

Изучение лексического и синтаксического анализа, используемого в компиляции исходного кода программы.

**Задачи**

1. По варианту задания определить, какие классы лексем будут в вашем языке.
2. Составить контрольные примеры на реализуемом языке.
3. Запрограммировать и отладить модуль сканирования. Выполнить тестирование на контрольных примерах. Результатом работы должна быть последовательность лексем, содержащая лексемы и признаки их классов.
4. По полученной последовательности лексем необходимо построить синтаксическое дерево.

**БНФ реализуемого языка**

<Программа> ::= <Объявление переменных> <Описание вычислений> .

<Описание вычислений> ::= Begin < Список операторов > End

<Объявление переменных> ::= Var <Список переменных>

<Список переменных> ::= <Идент> | <Идент> , <Список переменных>

<Список операторов> ::= <Оператор> | <Оператор> <Список операторов>  
<Оператор>::=<Присваивание> |<Сложный оператор>

<Присваивание> ::= <Идент> **:=** <Выражение>

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд> |

<Подвыражение> <Бин.оп.> <Подвыражение>

<Ун.оп.> ::= "- "

<Бин.оп.> ::= "-" | "+" | "\*" | "/" | ">>" | "<<" | ">" | "<" | "="

<Операнд> ::= <Идент> | <Const>

<Сложный оператор> ::= IF "("< Выражение> ")" Оператор|

IF "(" <Выражение> ")" <Оператор> ELSE <Оператор>|<Составной оператор>  
<Составной оператор>::= Begin < Список операторов > End

<Идент> ::= <Буква> <Идент> | <Буква>

<Const> ::= <Цифра> <Const> | <Цифра>

**Классы лексем**

* Ключевое слово
* Идентификатор
* Числовая константа
* Строковая константа
* Разделитель
* Операция

**Лексический анализ**

Лексический анализ - процесс аналитического разбора входной последовательности символов на распознанные группы — *лексемы*, с целью получения на выходе идентифицированных последовательностей, называемых «[токенами](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BA%D0%B5%D0%BD_(%D0%BB%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7))» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т. п.). Лексический анализ используется в [компиляторах](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) и [интерпретаторах](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) [языков программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), и в различных [парсерах](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D1%81%D0%B5%D1%80) [слов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%BE) [естественных языков](https://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA).

Как правило, лексический анализ производится с точки зрения определённого [формального языка](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса. Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

**class** Tokenizer {  
 **private val finiteAutomates**: MutableMap<Tag, (str: String) -> AutomateState> = *mutableMapOf*()  
  
 **private val isSeparator** = Regex(**"(begin|end|\\(|\\)|,)"**)  
 **private val isId** = Regex(**"[a-zA-Z]+\\w\*"**)  
 …

**init** { **finiteAutomates**[Tag.**ELSE**] = **{** str: String **->  
 when** {  
 **isElse**.matches(str) -> AutomateState.**FINAL  
 "else"**.*startsWith*(str) -> AutomateState.**ACTIVE  
 else** -> AutomateState.**ERROR** }  
 **}  
 finiteAutomates**[Tag.**ID**] = **{** str: String **->  
 when** {  
 **isId**.matches(str) -> AutomateState.**FINAL  
 else** -> AutomateState.**ERROR** }  
 **}  
 …** }  
  
 **fun** getTokens(expression: String): List<Token> {  
 **val** result = *mutableListOf*<Token>()  
 **val** chars = expression.*toCharArray*() **while** (index < chars.**size**) {  
 **for** (it **in finiteAutomates**) {  
 **while** (index < chars.**size** && chars[index].*isSpace*()) {  
 index++  
 }  
  
 **while** (thisState != AutomateState.**ERROR**) {  
 lastState = thisState  
 index++  
 **if** (index == chars.**size**) **break** str += chars[index]  
 thisState = it.**value**(str)  
 }  
  
 **if** (lastState == AutomateState.**FINAL**) {  
 **val** attribute = **if** (index < chars.**size**) str.*substring*(0, str.**length** - 1) **else** str  
 result.add(Token(it.**key**, attribute))  
 isCorrectToken = **true  
 break** } **else** {  
 index -= (str.**length** - 1)  
 }  
 }

…  
 }  
  
 **return** result  
 }

**Синтаксический анализ**

Синтаксический анализ - процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

**Метод рекурсивного спуска**

Алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора.

**lass** SyntaxAnalyzer {  
  
 **fun** parse(tokens: List<Token>): Symbol? {  
 **if** (tokens.*any* **{ it**.**name** == Tag.**ERROR }**) **return null  
  
 return** getProgram(tokens)  
 }  
  
 **private fun** getProgram(tokens: List<Token>): Symbol? {  
 **val** result = Node(**"program"**)  
  
 **val** varToken = tokens.*firstOrNull* **{ it**.**name** == Tag.**VAR }** ?: **return null  
 val** beginToken = tokens.*firstOrNull* **{ it**.**attribute** == **"begin" }** ?: **return null  
 val** endToken = tokens.*lastOrNull* **{ it**.**attribute** == **"end" }** ?: **return null  
  
 val** declaration = getDeclarationOfVariables(tokens.subList(tokens.indexOf(varToken), tokens.indexOf(beginToken)))  
 ?: **return null  
 val** computing = getComputing(tokens.subList(tokens.indexOf(beginToken), tokens.lastIndexOf(endToken) + 1))  
 ?: **return null** result.**nodes**.add(declaration)  
 result.**nodes**.add(computing)  
  
 **return** result  
 }  
  
 **private fun** getDeclarationOfVariables(tokens: List<Token>): Symbol? {  
 **val** result = Node(**"declaration of variables"**)  
  
 **val** varToken = tokens.*firstOrNull* **{ it**.**name** == Tag.**VAR }** ?: **return null  
  
 val** variables = getVariables(tokens.*takeLastWhile* **{ it** != varToken **}**) ?: **return null** result.**nodes**.add(Leaf(varToken, **"key word"**))  
 result.**nodes**.add(variables)  
  
 **return** result  
 }  
  
 **private fun** getVariables(tokens: List<Token>): Symbol? {  
 **val** result = Node(**"list of variables"**)  
  
 tokens.*forEach* **{  
 when** {  
 **it**.**name** == Tag.**ID** -> {  
 **val** variable = getVariable(**it**) ?: **return null** result.**nodes**.add(variable)  
 }  
 **it**.**name** == Tag.**SEPARATOR** && **it**.**attribute** == **","** -> {  
 }  
 **else** -> **return null** }  
 **}  
  
 return** result  
 }

…

**Результат работы программы**

Пример входной последовательности символов

var x, y  
  
begin  
 x:= y << 1  
 y:=3  
 if (x>0) y := 1  
 else  
 begin  
 y := 0  
 x := 3  
 end  
end

Результат работы лексического анализатора:

Token: VAR; attribute: var

Token: ID; attribute: x

Token: SEPARATOR; attribute: ,

Token: ID; attribute: y

Token: SEPARATOR; attribute: begin

Token: ID; attribute: x

Token: OPERATION; attribute: :=

Token: ID; attribute: y

Token: OPERATION; attribute: <<

Token: NUMBER; attribute: 1

Token: ID; attribute: y

Token: OPERATION; attribute: :=

Token: NUMBER; attribute: 3

Token: IF; attribute: if

Token: SEPARATOR; attribute: (

Token: ID; attribute: x

Token: OPERATION; attribute: >

Token: NUMBER; attribute: 0

Token: SEPARATOR; attribute: )

Token: ID; attribute: y

Token: OPERATION; attribute: :=

Token: NUMBER; attribute: 1

Token: ELSE; attribute: else

Token: SEPARATOR; attribute: begin

Token: ID; attribute: y

Token: OPERATION; attribute: :=

Token: NUMBER; attribute: 0

Token: ID; attribute: x

Token: OPERATION; attribute: :=

Token: NUMBER; attribute: 3

Token: SEPARATOR; attribute: end

Token: SEPARATOR; attribute: end

Результат работы синтаксического анализатора:

*<?***xml version="1.0" encoding="UTF-8" standalone="yes"***?>*<**node description="program"**>  
 <**node description="declaration of variables"**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>VAR</**name**>  
 <**attribute**>var</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="list of variables"**>  
 <**leaf description="variable"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>x</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="variable"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>y</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 </**node**>  
 <**node description="description of computing"**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>begin</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="list of operators"**>  
 <**node description="assigment"**>  
 <**leaf description="id"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>x</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>:=</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="subexpression"**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>y</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="binary operation"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>**&lt;&lt;**</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>1</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 </**node**>  
 <**node description="assigment"**>  
 <**leaf description="id"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>y</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>:=</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>3</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 <**node description="difficult operator"**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>IF</**name**>  
 <**attribute**>if</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="open bracket"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>(</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="subexpression"**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>x</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="binary operation"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>**&gt;**</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>0</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 <**leaf description="close bracket"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>)</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="assigment"**>  
 <**leaf description="id"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>y</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>:=</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>1</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>ELSE</**name**>  
 <**attribute**>else**&#xD;**</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="Composite operator"**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>begin</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**node description="list of operators"**>  
 <**node description="assigment"**>  
 <**leaf description="id"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>y</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>:=</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>0</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 <**node description="assigment"**>  
 <**leaf description="id"**>  
 <**token**>  
 <**name**>ID</**name**>  
 <**attribute**>x</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>OPERATION</**name**>  
 <**attribute**>:=</**attribute**>  
 </**token**>  
 </**leaf**>  
 <**leaf description="operand"**>  
 <**token**>  
 <**name**>NUMBER</**name**>  
 <**attribute**>3</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 </**node**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>end</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
 </**node**>  
 </**node**>  
 <**leaf description="key word"**>  
 <**token**>  
 <**name**>SEPARATOR</**name**>  
 <**attribute**>end</**attribute**>  
 </**token**>  
 </**leaf**>  
 </**node**>  
</**node**>

**Вывод**

В результате выполнения лабораторной работы мною были изучены и реализованы принципы построения лексического и синтаксического анализа. По БНФ реализуемого языка было выделено шесть классов лексем. В качестве языка реализации выбран Kotlin. На вход лексического анализатора подается последовательность символов, представляющая собой исходный код программы реализуемого языка. Алгоритм представляет собой последовательную передачу символов на вход конечным автоматам, до тех пока не будет найден токен. На выходе получаем последовательность лексем, которая передается синтаксическому анализатору, анализатор строит синтаксическое дерево по данной последовательности.