Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчёт

по лабораторной работе №1

**Лексический анализатор**

Выполнил:

Семашко Г. В.

Проверил:

Шиманский В. В.

Минск 2016

# Цель

Освоить работу с существующими лексическими анализаторами. Разработать собственный лексический анализатор языка/подмножества языка программирования для чего определить лексические правила и выполнить перевод потока символов в поток лексем (токенов).

Написать программу, выполняющую анализ код на языке java. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами. На вход программы подается текстовый файл (TextData.txt), содержащий строки символов анализируемой программы.

# Общие теоретические сведения

### *Введение.*

В информатике лексический анализ — процесс аналитического разбора входной последовательности символов (например, такой как исходный код на одном из языков программирования) с целью получения на выходе последовательности символов, называемых «токенами» (подобно группировке букв в словах). Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Как правило, лексический анализ производится с точки зрения определённого формального языка 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.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

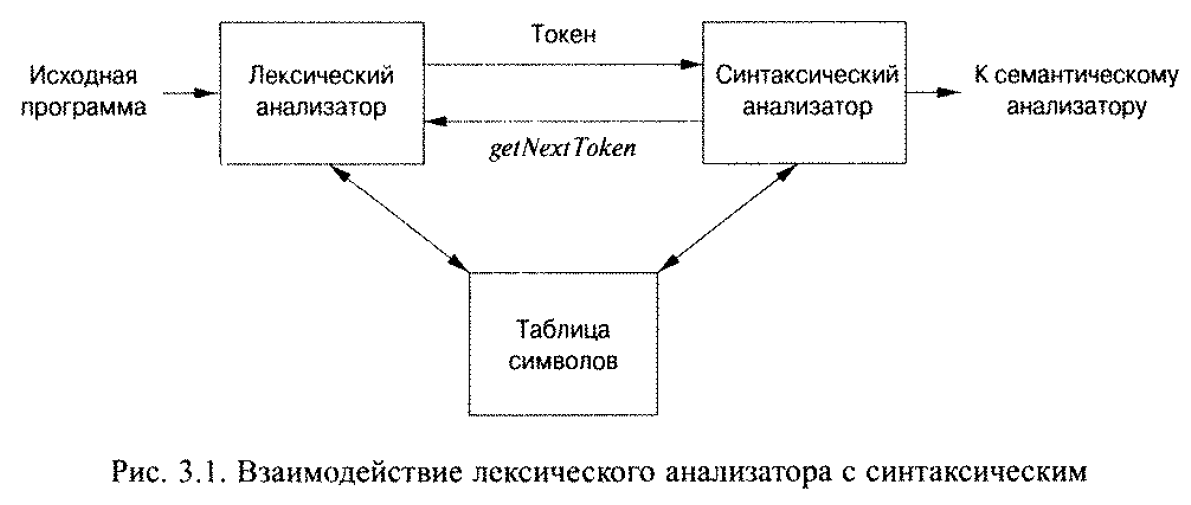
Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы, например, для грамматического анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

### *Взаимодействие лексического анализатора с синтаксическим.*

Лексический анализатор представляет собой первую фазу компиляции, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен.

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и некоторые другие действия, помимо идентификации лексем. Одной из таких задач является отбрасывание комментариев и пробельных символов.

Взаимодействие проиллюстрировано на рисунке 3.1.



# Выполнение программы

Программа выполняет анализ код на языке java. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами.

На вход программы подается текстовый файл (TextData.txt), содержащий строки символов анализируемой программы.

На выходе получаем лексемы с их идентификаторами. Виды идентификаторов: RESERVED\_WORD для зарезервированных слов, SEMICOLON для точки с запятой, LEFT\_PARENTHESES для открывающейся скобки, RIGHT\_PARENTHESES для закрывающейся скобки, OPEN\_CURLY\_BRACKET для открывающейся фигурной скобки, CLOSED\_CURLY\_BRACKET для закрывающейся фигурной скобки, MATH\_OPERATORS для математических операторов, NUMBER для чисел, WORD для имен переменных.

Список зарезервированных слов:

List<String> reservedWords = **new** ArrayList<>();  
reservedWords.add(**"abstract"**);  
reservedWords.add(**"assert"**);  
reservedWords.add(**"boolean"**);  
reservedWords.add(**"break"**);  
reservedWords.add(**"byte"**);  
reservedWords.add(**"case"**);  
reservedWords.add(**"catch"**);  
reservedWords.add(**"char"**);  
reservedWords.add(**"class"**);  
reservedWords.add(**"const"**);  
reservedWords.add(**"continue"**);  
reservedWords.add(**"default"**);  
reservedWords.add(**"do"**);  
reservedWords.add(**"double"**);  
reservedWords.add(**"else"**);  
reservedWords.add(**"enum"**);  
reservedWords.add(**"extends"**);  
reservedWords.add(**"false"**);  
reservedWords.add(**"final"**);  
reservedWords.add(**"finally"**);  
reservedWords.add(**"float"**);  
reservedWords.add(**"for"**);  
reservedWords.add(**"goto"**);  
reservedWords.add(**"if"**);  
reservedWords.add(**"implements"**);  
reservedWords.add(**"import"**);  
reservedWords.add(**"instanceof"**);  
reservedWords.add(**"int"**);  
reservedWords.add(**"interface"**);  
reservedWords.add(**"long"**);  
reservedWords.add(**"native"**);  
reservedWords.add(**"new"**);  
reservedWords.add(**"null"**);  
reservedWords.add(**"package"**);  
reservedWords.add(**"private"**);  
reservedWords.add(**"protected"**);  
reservedWords.add(**"public"**);  
reservedWords.add(**"return"**);  
reservedWords.add(**"short"**);  
reservedWords.add(**"static"**);  
reservedWords.add(**"super"**);  
reservedWords.add(**"switch"**);  
reservedWords.add(**"synchronized"**);  
reservedWords.add(**"this"**);  
reservedWords.add(**"throw"**);  
reservedWords.add(**"throws"**);  
reservedWords.add(**"transient"**);  
reservedWords.add(**"true"**);  
reservedWords.add(**"try"**);  
reservedWords.add(**"void"**);  
reservedWords.add(**"volatile"**);  
reservedWords.add(**"while"**);

Текст для анализа, находящийся в файле TextData.txt:

int n = 5;  
int m = 6;  
boolean greater;  
if( n > m )  
 greater = true;

Демонстрация выполнения программы представлена на рисунке 1.

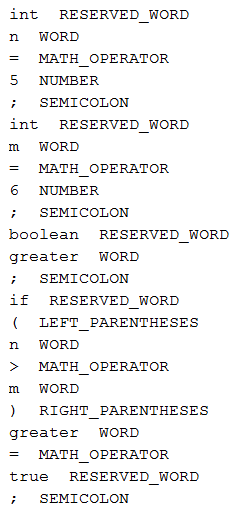


Рис.1

# Вывод

В ходе выполнения лабораторной работы была освоена работа с существующими лексическими анализаторами. Был разработан собственный лексический анализатор языка java, для чего определялись лексические правила и выполнялся перевод потока символов в поток лексем (токенов).

На вход программы подается текстовый файл (TextData.txt), содержащий строки символов анализируемой программы.

На выходе получаем лексемы с их идентификаторами. Виды идентификаторов: RESERVED\_WORD для зарезервированных слов, SEMICOLON для точки с запятой, LEFT\_PARENTHESES для открывающейся скобки, RIGHT\_PARENTHESES для закрывающейся скобки, OPEN\_CURLY\_BRACKET для открывающейся фигурной скобки, CLOSED\_CURLY\_BRACKET для закрывающейся фигурной скобки, MATH\_OPERATORS для математических операторов, NUMBER для чисел, WORD для имен переменных.

Приложение 1.

Код программы.

**package** mt.controller;  
  
  
**import** mt.action.TextReader;  
**import** mt.chain.BasicBreaker;  
**import** mt.chain.TextBreaker;  
**import** mt.composite.Component;  
  
**public class** Main {  
 **private static final** String ***FILE\_NAME*** = **"./TextData.txt"**;  
  
 **public static void** main(String[] args) {  
 TextReader textReader = **new** TextReader();  
 String content = textReader.readTextFromFile(***FILE\_NAME***);  
  
 BasicBreaker breaker = **new** TextBreaker();  
 Component text = breaker.breakText(content);  
 System.***out***.println(text);  
  
 }  
}

**package** mt.action;  
  
**import** org.apache.logging.log4j.LogManager;  
**import** org.apache.logging.log4j.Logger;  
  
**import** java.io.IOException;  
**import** java.nio.file.Files;  
**import** java.nio.file.NoSuchFileException;  
**import** java.nio.file.Paths;  
**import** java.util.stream.Collectors;  
  
**public class** TextReader {  
 **private static final** Logger ***logger*** = LogManager.*getLogger*();  
  
 **public** String readTextFromFile(String filepath) {  
 String content = **null**;  
 **try** {  
 content = Files.*lines*(Paths.*get*(filepath)).collect(Collectors.*joining*());  
 } **catch** (NoSuchFileException e) {  
 ***logger***.fatal(e);  
 **throw new** RuntimeException();  
 } **catch** (IOException e) {  
 ***logger***.error(e);  
 }  
 **return** content;  
 }  
}

**package** mt.chain;  
  
  
**import** mt.composite.Component;  
  
**public abstract class** BasicBreaker {  
 BasicBreaker **successor**;  
  
 **void** setSuccessor(BasicBreaker successor) {  
 **this**.**successor** = successor;  
 }  
  
 **public abstract** Component breakText(String data);  
}

**package** mt.chain;  
  
  
**import** mt.composite.Component;  
**import** mt.composite.TextComposite;  
  
**import static** mt.composite.ComponentType.***TEXT***;  
  
**public class** TextBreaker **extends** BasicBreaker {  
 **private static final** String ***NEW\_LINE*** = **"(?=[\\t])"**;  
  
 **public** TextBreaker() {  
 **this**.setSuccessor(**new** ParagraphBreaker());  
 }  
  
 **public** Component breakText(String text) {  
 Component textComposite = **new** TextComposite(***TEXT***);  
 **for** (String part : text.split(***NEW\_LINE***)) {  
 textComposite.add(**successor**.breakText(part));  
 }  
 **return** textComposite;  
 }  
}

**package** mt.chain;  
  
  
**import** mt.composite.Component;  
**import** mt.composite.TextComposite;  
  
**import static** mt.composite.ComponentType.***PARAGRAPH***;  
  
**public class** ParagraphBreaker **extends** BasicBreaker {  
 **private static final** String ***SEMICOLON*** = **"(?<=[;])|(?=[;])"**;  
  
 **public** ParagraphBreaker() {  
 **this**.setSuccessor(**new** SentenceBreaker());  
 }  
  
 **public** Component breakText(String paragraph) {  
 Component textComposite = **new** TextComposite(***PARAGRAPH***);  
 **for** (String part : paragraph.split(***SEMICOLON***)) {  
 textComposite.add(**successor**.breakText(part));  
 }  
 **return** textComposite;  
 }  
}

**package** mt.chain;  
  
  
**import** mt.composite.Component;  
**import** mt.composite.TextComposite;  
**import** mt.composite.TextLeaf;  
**import** mt.constant.ReservedIdentifiers;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.regex.Pattern;  
  
**import static** mt.composite.ComponentType.\*;  
**import static** mt.constant.Constant.***NUMBER\_PATTERN***;  
**import static** mt.constant.ReservedIdentifiers.*getMathOperators*;  
**import static** mt.constant.ReservedIdentifiers.*getReservedCharacters*;  
**import static** mt.constant.ReservedIdentifiers.*getReservedWords*;  
  
**public class** SentenceBreaker **extends** BasicBreaker {  
*// private static final String WORD\_DELIMITER = "(?<=[—, '.!?])|(?=[—, '.!?])";* **private** List<String> **reservedWords** = *getReservedWords*();  
 **private** List<String> **reservedCharacters** = *getReservedCharacters*();  
 **private** List<String> **mathOperators** = *getMathOperators*();  
  
 **public** Component breakText(String sentence) {  
 Component textComposite = **new** TextComposite(***SENTENCE***);  
 **for** (String part : sentence.split(**" "**)) {  
 part = part.trim();  
 **if** (!part.isEmpty()) {  
 **if** (***NUMBER\_PATTERN***.matcher(part).find()) {  
 textComposite.add(**new** TextLeaf(part, ***NUMBER***));  
 } **else if** (**reservedWords**.contains(part)) {  
 textComposite.add(**new** TextLeaf(part, ***RESERVED\_WORD***));  
 } **else if** (**reservedCharacters**.contains(part)) {  
 textComposite.add(**new** TextLeaf(part, ***RESERVED\_CHARACTER***));  
 } **else if** (**mathOperators**.contains(part)) {  
 textComposite.add(**new** TextLeaf(part, ***MATH\_OPERATOR***));  
 } **else** {  
 textComposite.add(**new** TextLeaf(part, ***WORD***));  
 }  
 }  
 }  
 **return** textComposite;  
 }  
}

**package** mt.composite;  
  
**import** java.util.List;  
  
**public abstract class** Component {  
 ComponentType **type**;  
  
 **public abstract void** add(Component component);  
  
 **public abstract void** remove(Component component);  
  
 **public abstract** List<Component> getComponents();  
  
 **public abstract void** setComponents(List<Component> components);  
  
 **public** ComponentType getType() {  
 **return type**;  
 }  
}

**package** mt.composite;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** TextComposite **extends** Component {  
 **private** List<Component> **components** = **new** ArrayList<>();  
  
 **public** TextComposite() {  
 }  
  
 **public** TextComposite(ComponentType type) {  
 **this**.**type** = type;  
 }  
  
 **public void** add(Component component) {  
 **components**.add(component);  
 }  
  
 **public void** remove(Component component) {  
 **components**.remove(component);  
 }  
  
 @Override  
 **public** String toString() {  
 StringBuilder stringBuilder = **new** StringBuilder();  
 **for** (Component component : **components**) {  
 stringBuilder.append(component.toString());  
 stringBuilder.append(**'\n'**);  
 }  
 **return** stringBuilder.toString();  
 }  
  
 **public** List<Component> getComponents() {  
 **return components**;  
 }  
  
 **public void** setComponents(List<Component> components) {  
 **this**.**components** = components;  
 }  
}  
  
**package** mt.composite;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** TextLeaf **extends** Component {  
 **private** String **content**;  
  
 **public** TextLeaf() {  
 }  
  
 **public** TextLeaf(String content, ComponentType type) {  
 **this**.**content** = content;  
 **this**.**type** = type;  
 }  
  
 **public void** add(Component component) {  
 }  
  
 **public void** remove(Component component) {  
 }  
  
 @Override  
 **public** String toString() {  
 **return content** + **" "** + **type**;  
 }  
  
 **public** List<Component> getComponents() {  
 List<Component> singleComponent = **new** ArrayList<>();  
 singleComponent.add(**this**);  
 **return** singleComponent;  
 }  
  
 **public void** setComponents(List<Component> components) {  
 }  
}

**package** mt.constant;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** ReservedIdentifiers {  
 **public static** List<String> getReservedWords() {  
 List<String> reservedWords = **new** ArrayList<>();  
 reservedWords.add(**"abstract"**);  
 reservedWords.add(**"assert"**);  
 reservedWords.add(**"boolean"**);  
 reservedWords.add(**"break"**);  
 reservedWords.add(**"byte"**);  
 reservedWords.add(**"case"**);  
 reservedWords.add(**"catch"**);  
 reservedWords.add(**"char"**);  
 reservedWords.add(**"class"**);  
 reservedWords.add(**"const"**);  
 reservedWords.add(**"continue"**);  
 reservedWords.add(**"default"**);  
 reservedWords.add(**"do"**);  
 reservedWords.add(**"double"**);  
 reservedWords.add(**"else"**);  
 reservedWords.add(**"enum"**);  
 reservedWords.add(**"extends"**);  
 reservedWords.add(**"false"**);  
 reservedWords.add(**"final"**);  
 reservedWords.add(**"finally"**);  
 reservedWords.add(**"float"**);  
 reservedWords.add(**"for"**);  
 reservedWords.add(**"goto"**);  
 reservedWords.add(**"if"**);  
 reservedWords.add(**"implements"**);  
 reservedWords.add(**"import"**);  
 reservedWords.add(**"instanceof"**);  
 reservedWords.add(**"int"**);  
 reservedWords.add(**"interface"**);  
 reservedWords.add(**"long"**);  
 reservedWords.add(**"native"**);  
 reservedWords.add(**"new"**);  
 reservedWords.add(**"null"**);  
 reservedWords.add(**"package"**);  
 reservedWords.add(**"private"**);  
 reservedWords.add(**"protected"**);  
 reservedWords.add(**"public"**);  
 reservedWords.add(**"return"**);  
 reservedWords.add(**"short"**);  
 reservedWords.add(**"static"**);  
 reservedWords.add(**"super"**);  
 reservedWords.add(**"switch"**);  
 reservedWords.add(**"synchronized"**);  
 reservedWords.add(**"this"**);  
 reservedWords.add(**"throw"**);  
 reservedWords.add(**"throws"**);  
 reservedWords.add(**"transient"**);  
 reservedWords.add(**"true"**);  
 reservedWords.add(**"try"**);  
 reservedWords.add(**"void"**);  
 reservedWords.add(**"volatile"**);  
 reservedWords.add(**"while"**);  
 **return** reservedWords;  
 }  
  
 **public static** List<String> getReservedCharacters() {  
 List<String> reservedCharacters = **new** ArrayList<>();  
 reservedCharacters.add(**"{"**);  
 reservedCharacters.add(**"}"**);  
 reservedCharacters.add(**"("**);  
 reservedCharacters.add(**")"**);  
 reservedCharacters.add(**";"**);  
 **return** reservedCharacters;  
 }  
  
 **public static** List<String> getMathOperators() {  
 List<String> mathOperators = **new** ArrayList<>();  
 mathOperators.add(**">"**);  
 mathOperators.add(**"<"**);  
 mathOperators.add(**"="**);  
 mathOperators.add(**"+"**);  
 mathOperators.add(**"-"**);  
 mathOperators.add(**"--"**);  
 mathOperators.add(**"++"**);  
 mathOperators.add(**">="**);  
 mathOperators.add(**"<="**);  
 mathOperators.add(**"\*"**);  
 mathOperators.add(**"/"**);  
 mathOperators.add(**"&&"**);  
 mathOperators.add(**"||"**);  
 mathOperators.add(**"=="**);  
 mathOperators.add(**"!="**);  
 **return** mathOperators;  
 }  
}