|  |  |
| --- | --- |
|  | ы |
| МИНОБРНАУКИ РОССИИ |  |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |  |

**Институт информационных технологий**

КАФЕДРА ИНСТРУМЕТНАЛЬНОГО И ПРИКЛАДНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ (ИППО)

ПРАКТИЧЕСКАЯ РАБОТА № 3

по дисциплине «Теория автоматов и формальных языков»

«ЗНАКОМСТВО С ЛЕКСИЧЕСКИМ АНАЛИЗОМ»

Выполнил студент группы ИКБО-12-18 Копотов М.А.

Преподаватель: Алпатов А.Н.

Практическая работа выполнена «13» октября 2020г.

«Зачтено» «\_\_\_»\_\_\_\_\_\_\_2020г.

Москва 2020

Цель работы

Знакомство с лексическим анализом, построение собственного лексического анализатора.

**Выполнение работы**

**Задание.** Вариант 13. Входной язык содержит логические выражения, разделённые символом ; (точка с запятой). Логические выражения состоят из идентификаторов, символьных констант 'T' и 'F', знака присваивания (:=), операций or, xor, and, not и круглых скобок.

*Листинг А — Lexer*

|  |
| --- |
| package com.lab3.mike;  import java.text.ParseException; import java.util.\*; import java.util.regex.Matcher; import java.util.regex.Pattern; /\*\*  \* Лексический анализатор \*/ public class Lexer {  /\*\*  \* Входная строка  \*/  private final String str;  /\*\*  \* Текущая позиция во входной строке  \*/  private int index = 0;   public Lexer(String str) {  this.str = str;  }   /\*\*  \* Попытка сопоставить текст, начиная с текущей позиции index, с  \* регулярным выражением. \*  \*  \* @param pattern регулярное выражение  \* @return -1, если если регулярное выражение не удалось найти в  \* текущей позиции; значение >= 0 - индекс первого символа,  \* следующего после найденной лексемы, соответствующей  \* регулярному выражению  \*/  private int match(Pattern pattern) {  Matcher matcher = pattern.matcher(str);  // Устанавливаем регион поиска - начиная с текущей позиции:  matcher.region(index, str.length());  if (matcher.lookingAt()) {  // Да, в текущей позиции найдено регулярное выражение - возвращаем индекс символа \_после\_ найденной лексемы  return matcher.end();  } else {  // Не найдено совпадения - возвращаем -1  return -1;  }  }   private Token matchNumber() {  Pattern numberPattern = Pattern.compile("[0-9]+([abcdef]\*)?");  int matched = match(numberPattern);  if (matched < 0)  return null;  String numberText = str.substring(index, matched);  return new Token(TokenType.NUMBER, numberText, index, matched);  }   private final Map<String, TokenType> SYMBOL\_MAP = new LinkedHashMap<>();  {  SYMBOL\_MAP.put("+", TokenType.ADD);  SYMBOL\_MAP.put("-", TokenType.SUB);  SYMBOL\_MAP.put("\*", TokenType.MUL);  SYMBOL\_MAP.put("/", TokenType.DIV);  SYMBOL\_MAP.put("(", TokenType.LPAR);  SYMBOL\_MAP.put(")", TokenType.RPAR);  SYMBOL\_MAP.put(":=", TokenType.ASSIGN);  SYMBOL\_MAP.put(";", TokenType.END);  }   private Token matchAnySymbol() {  for (Map.Entry<String, TokenType> entry : SYMBOL\_MAP.entrySet()) {  String key = entry.getKey();  TokenType value = entry.getValue();  Pattern symbolPattern = Pattern.compile(Pattern.quote(key));  int matched = match(symbolPattern);  if (matched < 0) continue;  String symbolText = str.substring(index, matched);  return new Token(value, symbolText, index, matched);  }  return null;  }   private Token matchSpaces() {  int i = index;  while (i < str.length()) {  char ch = str.charAt(i);  if (ch <= ' ') {  i++;  } else {  break;  }  }  if (i > index) {  String spaces = str.substring(index, i);  return new Token(TokenType.SPACES, spaces, index, i);  } else {  return null;  }  }   private Token matchVariable() {  Pattern varPattern = Pattern.compile("[A-Za-z]+[\\w]\*");  int matched = match(varPattern);  if (matched < 0)  return null;  String varText = str.substring(index, matched);  return new Token(TokenType.VAR, varText, index, matched);  }   /\*\*  \* Получение лексемы, стоящей в текущей позиции. \*  \*  \* @return null, если в строке больше нет лексем  \*/  private Token matchAnyToken() throws ParseException {  // Мы стоим в конце строки - больше нет лексем:  if (index >= str.length())  return null;  // Перебираем все возможные типы лексем:   Token spacesToken = matchSpaces();  if (spacesToken != null)  return spacesToken;  Token numberToken = matchNumber();  if (numberToken != null)  return numberToken;  Token symbolToken = matchAnySymbol();  if (symbolToken != null)  return symbolToken;  Token varToken = matchVariable();  if (varToken != null)  return varToken; // Символ в текущей позиции не подходит ни к одной из возможных лексем - ошибка:  throw new ParseException(  "Unexpected character '" + str.charAt(index) + "'", index);  }   /\*\*  \* Получение лексемы, стоящей в текущей позиции и перемещение текущей  \* позиции дальше. \*  \*  \* @return null, если в строке больше нет лексем  \*/  public Token nextToken() throws ParseException {  while (true) {  Token token = matchAnyToken();  if (token == null) {// Строка закончилась, больше нет лексем:  return null;  }  // Перемещаем текущую позицию после найденной лексемы:  index = token.to;  if (token.type != TokenType.SPACES) {// Непробельную лексему возвращаем:  return token;  }  }  }   public List<Token> getAllTokens() throws ParseException {  List<Token> allTokens = new ArrayList<>();  while (true) {  Token token = nextToken();  if (token == null)  break;  allTokens.add(token);  }  return allTokens;  } } |

Листинг B — Main

|  |
| --- |
| package com.lab4.tanya;  import com.lab3.mike.Lexer; import com.lab3.mike.Token;  import java.text.ParseException; import java.util.List; /\*  Реализовать разбор заданной грамматики в соответствии с вариантом задания.   выражение ::= слагаемое (('+'|'-') слагаемое)\*  слагаемое ::= множитель (('\*'|'/') множитель)\*  множитель ::= ('-')? ЧИСЛО | '(' выражение ')'   Число - любое целое число (лексмы взяты не из варианта прошлого задания, а из методички)  \*/  public class Main {   /\*\*  \* Проверка грамматического разбора выражения  \*/  public static void main(String[] args) {  Lexer lexer;  List<Token> allTokens;  Parser parser;   try {  lexer = new Lexer("89 + 4 - 56 \* -44 \* 2 + 3 - 5 \* 412");  allTokens = lexer.getAllTokens();  for(int i = 0; i < allTokens.size(); i++)  System.out.println(i +")\t"+ allTokens.get(i).str +"\t"+ allTokens.get(i).type);  }catch (ParseException pe){  System.out.println("Error In Lexer: " + pe.getMessage()+ " at " + pe.getErrorOffset());  return;  }   try {  parser = new Parser(allTokens);  parser.matchExpression();  }catch (ParseException pe){  System.out.println("Error In Parser: " + pe.getMessage() + " at " + pe.getErrorOffset());  return;  }  System.out.println("No Errors");  } } |

*Листинг C — Token*

|  |
| --- |
| package com.lab3.mike;  public class Token {  public TokenType type;  public String str;  public int from;  public int to;   public Token(TokenType type, String str, int from, int to) {  this.type = type;  this.str = str;  this.from = from;  this.to = to;  } } |

*Листинг D — TokenType*

|  |
| --- |
| package com.lab3.mike;  public enum TokenType {  /\*\*  \* Пробелы  \*/  SPACES,  /\*\*  \* Целое число  \*/  NUMBER,  /\*\*  \* Символ '+'  \*/  ADD,  /\*\*  \* Символ '-'  \*/  SUB,  /\*\*  \* Символ '\*'  \*/  MUL,  /\*\*  \* Символ '/'  \*/  DIV,  /\*\*  \* Символ '('  \*/  LPAR,  /\*\*  \* Символ ')'  \*/  RPAR,  /\*\*  \* Переменная  \*/  VAR,  /\*\*  \* Приравнивание :=  \*/  ASSIGN,  /\*\*  \* Конец выражения ;  \*/  END } |

**Изображение выглядит как текст

Автоматически созданное описание**

**Рисунок 1 – Пример работы программы**

**Вывод**

В результате выполнения практической работы мы ознакомились с лексическими анализаторами и построили собсвенный.

**Список литературных источников**

1. Зорина Н.В, Соболев О.В. Теория автоматов и формальных языков: Методические указания по выполнению практических работ для студентов, М. 2017, - 108 с.
2. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб, 2010. – 400 с.