Національний Технічний Університет України

“Київский Політехнічний Інститут”

Факультет Інформатики та Обчислювальної Техніки

Кафедра обчислювальної техніки

**Лабораторна робота №2.4**

по курсу «Системне програмування»

Виконав:

студент 3 курсу

ФІОТ гр. ІО-92

Петрук Вадим

Киев 2011

ПОБУДОВА І ВИКОРИСТАННЯ АВТОМАТІВ ЛЕКСИЧНОГО АНАЛІЗУ ТА МЕХАНІЗМІВ УПРАВЛІННЯ СТАНОМ

Мета роботи: Вивчення схеми табличного подання авто­мат­ної граматики лексичного аналізу. Використання об’єктів стану графів автоматів для формування лексем у форматі внутрішнього подання вузлів графів розбору.

**Лістинг програми:**

**package** lab2;

/\*\*

\* Testing class for the Lexical Analyzer.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*

\*/

**public** **class** **TestLexicalAnalyzer** {

/\*\*

\* Main testing method.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

String inputToken = "if a-c=0 then b:=(a-c)\*2\*a;";

LexicalAnalyzer analyzer = **new** LexicalAnalyzer();

**try** {

TokenTable table = analyzer.analyzeLine(inputToken);

table.print();

} **catch** (LineParseException e) {

e.printStackTrace();

}

}

}

**package** lab2;

/\*\*

\* Class analyze line and divide it into numbers, symbols, comparators,

\* separators, operations. Spaces and tabs are ignored.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*

\*/

**public** **class** **LexicalAnalyzer** {

/\*\*

\* Table with tokens from input line.

\*/

**private** TokenTable tokenTable = **new** TokenTable();

/\*\*

\* State of traversal tree.

\*/

**private** **int** state = 0;

/\*\*

\* Buffer variable of tokens.It accumulates tokens characters.

\*/

**private** String buffer = "";

/\*\*

\* Validate keywords of this token analyzer.

\*/

**private** String[] keyWords = { "if", "then", "else", "while", "until", "do" };

/\*\*

\* All possible types of symbols of input line.

\*/

**private** **enum** TypesOfSymbols {

*number*, *symbol*, *comparator*, *separator*, *operation*, *space*, *other*

};

/\*\*

\* Divide input line into symbols and delegate the analys of them to the

\* private method analyzeSymbol(char symbol).

\*

\* **@throws** LineParseException

\* **@param** inputLine

\* - analyzing line

\*/

**public** TokenTable analyzeLine(String inputLine) **throws** LineParseException {

**char** currentSymbol;

**for** (**int** i = 0; i < inputLine.length(); i++) {

currentSymbol = inputLine.charAt(i);

**boolean** symbolIsToken = **true**;

**while** (symbolIsToken) {

symbolIsToken = **this**.analyzeSymbol(currentSymbol);

}

}

**return** tokenTable;

}

/\*\*

\* Analyze symbols of input line and passes to the state of the tree

\* traversal.

\*

\* **@throws** ForbiddenSymbolException

\* , NumberFormatException, LineParseException

\* **@param** analyzing

\* symbol

\*/

**private** **boolean** analyzeSymbol(**char** symbol) **throws** ForbiddenSymbolException,

NumberFormatException, LineParseException {

TypesOfSymbols charType = determineCharType(symbol);

// System.out.println("curState = "+state+" smb = "+symbol+" ; charType = "+charType);

**switch** (state) {

**case** 0:

**switch** (charType) {

**case** *number*:

buffer += symbol;

state = 1;

**return** **false**;

**case** *symbol*:

buffer += symbol;

state = 7;

**return** **false**;

**case** *comparator*:

**if** (symbol == '<') {

buffer += symbol;

state = 8;

**return** **false**;

}

**if** (symbol == '=') {

buffer += symbol;

state = 0;

**this**.addTokenToTable(TokenName.*COMPARATOR*);

**return** **false**;

}

**if** (symbol == '>') {

buffer += symbol;

state = 9;

**return** **false**;

}

**case** *separator*:

buffer += symbol;

state = 0;

**this**.addTokenToTable(TokenName.*SEPARATOR*);

**return** **false**;

**case** *operation*:

**if** (symbol == ':') {

buffer += symbol;

state = 11;

**return** **false**;

} **else** {

buffer += symbol;

state = 0;

**this**.addTokenToTable(TokenName.*OPERATION*);

**return** **false**;

}

**case** *space*:

state = 10;

**return** **false**;

**case** *other*:

**throw** **new** ForbiddenSymbolException("Symbol '" + symbol

+ "' is forbidden for using!");

}

**case** 1:

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

**return** **false**;

} **else** {

**if** (symbol == '.') {

buffer += '.';

state = 2;

**return** **false**;

} **else** {

**if** ((symbol == 'E')) {

buffer += 'E';

state = 4;

**return** **false**;

} **else** {

**if** (charType.equals(TypesOfSymbols.*symbol*)) {

**throw** **new** NumberFormatException(

"Wrong format of number!");

}

**this**.addTokenToTable(TokenName.*NUMBER*);

state = 0;

**return** **true**;

}

}

}

**case** 2:

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

state = 3;

**return** **false**;

} **else** {

**throw** **new** NumberFormatException(

"Wrong format of number! Extra . in number");

}

**case** 3:

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

**return** **false**;

} **else** {

**if** (symbol == 'E') {

buffer += symbol;

state = 4;

**return** **false**;

} **else** {

**if** (charType.equals(TypesOfSymbols.*symbol*)) {

**throw** **new** NumberFormatException(

"Wrong format of number!");

}

**this**.addTokenToTable(TokenName.*NUMBER*);

state = 0;

**return** **true**;

}

}

**case** 4:

**if** ((symbol == '+') || (symbol == '-')) {

buffer += symbol;

state = 5;

**return** **false**;

} **else** {

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

state = 6;

**return** **false**;

} **else** {

**throw** **new** NumberFormatException(

"Wrong format of number!No digit after power.");

}

}

**case** 5:

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

state = 6;

**return** **false**;

} **else** {

**throw** **new** NumberFormatException(

"Wrong format of number! No digit after sign in a power");

}

**case** 6:

**if** (charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

state = 6;

**return** **false**;

} **else** {

**if** (charType.equals(TypesOfSymbols.*symbol*)) {

**throw** **new** NumberFormatException("Wrong format of number!");

}

**this**.addTokenToTable(TokenName.*NUMBER*);

state = 0;

**return** **true**;

}

**case** 7:

**if** (charType.equals(TypesOfSymbols.*symbol*)

|| charType.equals(TypesOfSymbols.*number*)) {

buffer += symbol;

**return** **false**;

} **else** {

**if** (**this**.isKeyWords(buffer)) {

// System.out.println("asrgrehgaqe");

**this**.addTokenToTable(TokenName.*KEYWORD*);

state = 0;

**return** **true**;

} **else** {

**this**.addTokenToTable(TokenName.*VARIABLE*);

state = 0;

**return** **true**;

}

}

**case** 8:

**if** (symbol == '=') {

buffer += symbol;

**this**.addTokenToTable(TokenName.*COMPARATOR*);

state = 0;

**return** **false**;

} **else** {

**if** (symbol == '>') {

buffer += symbol;

**this**.addTokenToTable(TokenName.*COMPARATOR*);

state = 0;

**return** **false**;

} **else** {

**this**.addTokenToTable(TokenName.*COMPARATOR*);

state = 0;

**return** **true**;

}

}

**case** 9:

**if** (symbol == '=') {

buffer += symbol;

**this**.addTokenToTable(TokenName.*COMPARATOR*);

state = 0;

**return** **false**;

} **else** {

**this**.addTokenToTable(TokenName.*COMPARATOR*);

state = 0;

**return** **true**;

}

**case** 10:

**if** (charType.equals(TypesOfSymbols.*space*)) {

**return** **false**;

} **else** {

state = 0;

**return** **true**;

}

**case** 11:

**if** (symbol == '=') {

buffer += symbol;

state = 0;

**this**.addTokenToTable(TokenName.*OPERATION*);

**return** **false**;

} **else** {

**throw** **new** LineParseException("Wrong format of oeration!");

}

**default**:

**throw** **new** ForbiddenSymbolException("Symbol '" + symbol

+ "' is forbidden for using!");

}

}

/\*\*

\* Determine type of input symbol from private enum type TypesOfSymbols.

\*

\* **@param** symbol

\*/

**private** TypesOfSymbols determineCharType(**char** symbol)

**throws** ForbiddenSymbolException {

String validSymbols = "0123456789";

String stringSymbol = "" + symbol;

**if** (validSymbols.contains(stringSymbol)) {

**return** TypesOfSymbols.*number*;

}

validSymbols = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\_";

**if** (validSymbols.contains(stringSymbol)) {

**return** TypesOfSymbols.*symbol*;

}

validSymbols = "><=!";

**if** (validSymbols.contains(stringSymbol)) {

**return** TypesOfSymbols.*comparator*;

}

validSymbols = "(){}[];";

**if** (validSymbols.contains(stringSymbol)) {

**return** TypesOfSymbols.*separator*;

}

validSymbols = "\*/+-:";

**if** (validSymbols.contains(stringSymbol)) {

**return** TypesOfSymbols.*operation*;

}

**if** ((symbol == ' ') || (symbol == ' ')) {

**return** TypesOfSymbols.*space*;

}

**return** TypesOfSymbols.*other*;

}

/\*\*

\* Create new token and add it to the table of tokens.

\*

\* **@param** tokenName

\* - type of token from enum class TokenName.

\*/

**private** **void** addTokenToTable(TokenName tokenName) {

Token token = **new** Token(tokenName, buffer);

tokenTable.addToken(token);

buffer = "";

}

/\*\*

\* Check out the word it belongs to the key words.

\*

\* **@param** mKeyWord

\* - checking word

\* **@return** is input word a key word?

\*/

**private** **boolean** isKeyWords(String mKeyWord) {

**for** (String keyWord : keyWords) {

**if** (keyWord.equals(mKeyWord)) {

**return** **true**;

}

}

**return** **false**;

}

}

**package** lab2;

**import** java.util.ArrayList;

/\*\*

\* Save tokens of input to lexical analyzer line.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **TokenTable** {

/\*\*

\* Analyzed tokens

\*/

ArrayList<Token> tokens = **new** ArrayList<Token>();

/\*\*

\* Add analyzed token to the table.

\*

\* **@param** token

\* - added token

\*/

**public** **void** addToken(Token token) {

**this**.tokens.add(token);

}

/\*\*

\* Getter for the table of tokens.

\*

\* **@return** the tokens - table of tokens

\*/

**public** ArrayList<Token> getTokens() {

**return** tokens;

}

/\*\*

\* Get one token from the table from the specified position.

\*

\* **@param** specified

\* position of token in table

\* **@return** the token

\*/

**public** Token getToken(**int** position) {

**return** tokens.get(position);

}

/\*\*

\* Print table of tokens to the console.

\*/

**public** **void** print() {

**for** (**int** i = 0; i < **this**.tokens.size(); i++) {

System.*out*.println(tokens.get(i).getTokenName() + "; value = "

+ tokens.get(i).getValue());

}

}

}

**package** lab2;

/\*\*

\* Describes the tokens of lexical analyzer.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **Token** {

**private** TokenName tokenName;

**private** String value;

/\*\*

\* Create token with defined parameters.

\*

\* **@param** tokenName

\* - type of token from enum type TokenName.

\* **@param** value

\* - value of token.

\*/

**public** Token(TokenName tokenName, String value) {

**this**.tokenName = tokenName;

**this**.value = value;

}

/\*\*

\* Getter for the private field tokenName.

\*

\* **@return** the tokenName

\*/

**public** TokenName getTokenName() {

**return** tokenName;

}

/\*\*

\* Getter for the private field value.

\*

\* **@return** the value

\*/

**public** String getValue() {

**return** value;

}

}

**package** lab2;

/\*\*

\* Describes all possible types of tokens of the lexical analyzer.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **enum** **TokenName** {

*NUMBER*, *VARIABLE*, *OPERATION*, *COMPARATOR*, *SEPARATOR*, *KEYWORD*

}

**package** lab2;

/\*\*

\* Signals about exception during parsing the line to the tokens.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **LineParseException** **extends** Exception {

LineParseException() {

**super**();

}

LineParseException(String massage) {

**super**(massage);

}

}

**package** lab2;

/\*\*

\* Signals about wrong format of numbers in the parsing by lexical analyzer line

\* .

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **NumberFormatException** **extends** LineParseException {

NumberFormatException() {

**super**();

}

NumberFormatException(String massage) {

**super**(massage);

}

}

**package** lab2;

/\*\*

\* Signals about unknown symbol in input line to the lexical analyzer.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **ForbiddenSymbolException** **extends** LineParseException {

ForbiddenSymbolException() {

**super**();

}

ForbiddenSymbolException(String massage) {

**super**(massage);

}

}

**package** lab2;

**import** junit.framework.TestCase;

/\*\*

\* Test Lexical Analyzer for the the correctness of the output result.

\*

\* **@author** Petruk V., e-mail: vadpetruk@mail.ru

\*/

**public** **class** **JTestLexicalAnalyzer** **extends** TestCase {

/\*\*

\* Compare the correct table with result of working of Lexical Analyzer.

\*/

**public** **void** testLexicalAnalyzer() {

Token[] tokens = **new** Token[19];

tokens[0] = **new** Token(TokenName.*KEYWORD*, "if");

tokens[1] = **new** Token(TokenName.*VARIABLE*, "a");

tokens[2] = **new** Token(TokenName.*OPERATION*, "-");

tokens[3] = **new** Token(TokenName.*VARIABLE*, "c");

tokens[4] = **new** Token(TokenName.*COMPARATOR*, "=");

tokens[5] = **new** Token(TokenName.*NUMBER*, "0");

tokens[6] = **new** Token(TokenName.*KEYWORD*, "then");

tokens[7] = **new** Token(TokenName.*VARIABLE*, "b");

tokens[8] = **new** Token(TokenName.*OPERATION*, ":=");

tokens[9] = **new** Token(TokenName.*SEPARATOR*, "(");

tokens[10] = **new** Token(TokenName.*VARIABLE*, "a");

tokens[11] = **new** Token(TokenName.*OPERATION*, "-");

tokens[12] = **new** Token(TokenName.*VARIABLE*, "c");

tokens[13] = **new** Token(TokenName.*SEPARATOR*, ")");

tokens[14] = **new** Token(TokenName.*OPERATION*, "\*");

tokens[15] = **new** Token(TokenName.*NUMBER*, "2");

tokens[16] = **new** Token(TokenName.*OPERATION*, "\*");

tokens[17] = **new** Token(TokenName.*VARIABLE*, "a");

tokens[18] = **new** Token(TokenName.*SEPARATOR*, ";");

TokenTable testTokenTable = **new** TokenTable();

**for** (Token token : tokens) {

testTokenTable.addToken(token);

}

String inputToken = "if a-c=0 then b:=(a-c)\*2\*a;";

LexicalAnalyzer analyzer = **new** LexicalAnalyzer();

TokenTable table = **null**;

**try** {

table = analyzer.analyzeLine(inputToken);

table.print();

} **catch** (LineParseException e) {

e.printStackTrace();

}

**for** (**int** i = 0; i < 2; i++) {

*assertEquals*(tokens[i].getTokenName(), table.getToken(i)

.getTokenName());

*assertEquals*(tokens[i].getValue(), table.getToken(i).getValue());

}

}

}