Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS4308, Concepts of Programming Languages, Section W01

Project Deliverable 1 Report

Joe Potchen & Daemin Lee

[jpotchen@students.kennesaw.edu](mailto:jpotchen@students.kennesaw.edu)

[dlee80@students.kennesaw.edu](mailto:dlee80@students.kennesaw.edu)

2/26/2018

Problem Statement:

Develop a complete scanner. Write a short report describing the work performed. Include the source program, input and output. You must show the execution of this program by using several relevant source lines as input, the program must show a list of the tokens scanned.

Summary of Program:

The scanner program was developed while watching the project lecture video from Dr. Gaylor, so the base code is very similar to his. There is a TokenType Enum class that contains all the different token names, as well as a Token CRUD class (Although a better description for it would be CR since it only creates and reads lexemes) that stores a lexeme with their respective token as well as their line number and column number.

Most of the program logic is done in the LexicalAnalyzer class, here the class is given a file, it then goes through each line of the program, finding all the lexemes on the line. It does this by searching for whitespace, then all the characters between the whitespace become a lexeme. However, this won’t work when reading in strings since they can’t contain whitespace, so the program finds the first quotation mark for the string and then iterates through the string until it finds the closing quotation mark. It then sets everything from the first quotation mark to the second quotation mark as the string lexeme.

The program determines what token a lexeme falls under using a large switch statement. Each case is an operator, and then the switch statement defaults to an if statement that checks if the first character of the lexeme is a digit, a letter, or a quotation mark. If it’s a digit it makes sure the entire lexeme is digits and then assigns it to the integer token, if it’s a letter it determines if it’s an ID or a loop/print/if statement, and if it’s a quotation mark it automatically knows that it’s a string.

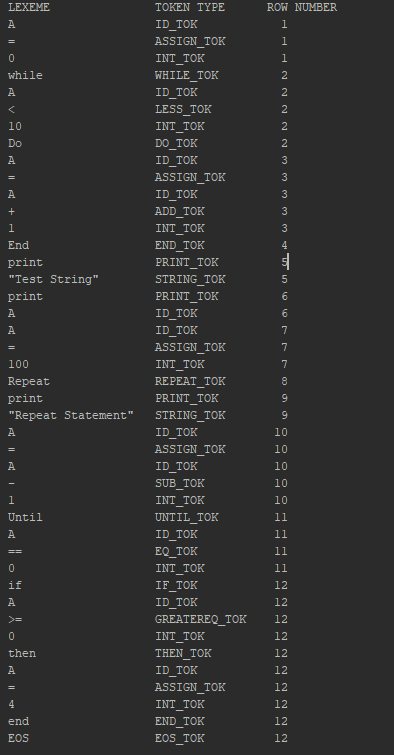
A lexical analyzer object is initialized in the Main class and given a LUA file that is from a java argument. The main class then calls the LexicalAnalyzer method LexicalAnalyzer.printTokens(). This method prints out each lexeme and its respective token and row number.

The input file is a LUA file that has an example of almost every token in it. It didn’t seem necessary to repeat things like \* and / since + and – both work, and they use the same logic in the LexicalAnalyzer class.

Sample.lua:



Program Output:



Source Code:

Main.java



Token.java



LexicalAnalyzer.java

package com.Potchen;  
  
import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
import java.util.Scanner;  
  
/\*  
  
  
  
 \*/  
  
  
  
*/\*\*  
 \* Created by Joe on 2/22/2018.  
 \*/*public class LexicalAnalyzer {  
 private List<Token> tokens;  
  
 public LexicalAnalyzer(String fileName) throws FileNotFoundException, LexicalException{  
 assert (fileName != null);  
 tokens = new ArrayList<Token>();  
 Scanner sourceCode = new Scanner(new File(fileName));  
 int lineNumber = 0;  
 while(sourceCode.hasNext()){  
 String line = sourceCode.nextLine();  
 processLine(line, lineNumber);  
 lineNumber++;  
 }  
 tokens.add(new Token(lineNumber, 1, "EOS", TokenType.*EOS\_TOK*));  
 sourceCode.close();  
  
 }  
  
  
 */\*\*  
 \* Prints a formatted version of the tokens List.  
 \*/* public void printTokens(){  
 System.*out*.printf("%-20s %-15s %3s\n", "LEXEME", "TOKEN TYPE", "ROW NUMBER");  
 for (int i = 0; i < tokens.size(); i++) {  
 System.*out*.printf("%-20s %-15s %3s\n", tokens.get(i).getLexeme(), tokens.get(i).getTokenType(), tokens.get(i).getRowNumber());  
 }  
 }  
  
  
 */\*\*  
 \* This goes through a line and finds a lexeme using the whitespace that surrounds it.  
 \* It then calls getTokenType to determine the token of the lexeme before moving onto the next lexeme.  
 \** ***@param*** *line  
 \** ***@param*** *lineNumber  
 \** ***@throws*** *LexicalException  
 \*/* private void processLine(String line, int lineNumber) throws LexicalException{  
 assert (line != null && lineNumber >= 1);  
 int index = 0;  
 index = skipWhiteSpace(line, index);  
 while(index < line.length()){  
 String lexeme = getLexeme(line, lineNumber, index);  
 TokenType tokType = getTokenType (lexeme, lineNumber, index);  
 tokens.add(new Token(lineNumber + 1, index +1, lexeme, tokType));  
 index += lexeme.length();  
 index = skipWhiteSpace(line, index);  
 }  
 }  
  
 */\*\*  
 \* Determines the tokenType of a given lexeme.  
 \** ***@param*** *lexeme  
 \** ***@param*** *lineNumber  
 \** ***@param*** *columnNumber  
 \** ***@return*** *\** ***@throws*** *LexicalException  
 \*/* private TokenType getTokenType(String lexeme, int lineNumber, int columnNumber) throws LexicalException{  
 assert (lexeme != null && lineNumber >= 1 && columnNumber >= 1);  
 TokenType tokType = null;  
  
  
 */\*\*  
 \* Determines what the token is based on the lexeme  
 \*/* switch (lexeme){  
 case "=":  
 return TokenType.*ASSIGN\_TOK*;  
 case "==":  
 return TokenType.*EQ\_TOK*;  
 case "<":  
 return TokenType.*LESS\_TOK*;  
 case "<=":  
 return TokenType.*LESSEQ\_TOK*;  
 case ">":  
 return TokenType.*GREATER\_TOK*;  
 case ">=":  
 return TokenType.*GREATEREQ\_TOK*;  
 case "~=":  
 return TokenType.*NOTEQ\_TOK*;  
 case "+":  
 return TokenType.*ADD\_TOK*;  
 case "-":  
 return TokenType.*SUB\_TOK*;  
 case "\*":  
 return TokenType.*MULT\_TOK*;  
 case "/":  
 return TokenType.*DIV\_TOK*;  
 default:  
  
 */\*\*  
 \* Checks to see if the first character is a letter. If it is it then determines  
 \* if it's an ID or a statement token.  
 \*/* if (Character.*isLetter*(lexeme.charAt(0))){  
 if(lexeme.length() == 1){  
 tokType = TokenType.*ID\_TOK*;  
 } else{  
  
  
 if(lexeme.equalsIgnoreCase("While")){  
 return TokenType.*WHILE\_TOK*;  
  
 }else if (lexeme.equalsIgnoreCase("Print")){  
 return TokenType.*PRINT\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("If")){  
 return TokenType.*IF\_TOK*;  
 }else if(lexeme.equalsIgnoreCase("Repeat")){  
 return TokenType.*REPEAT\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("Do")){  
 return TokenType.*DO\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("Until")){  
 return TokenType.*UNTIL\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("End")){  
 return TokenType.*END\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("Then")){  
 return TokenType.*THEN\_TOK*;  
 }else if (lexeme.equalsIgnoreCase("Else")){  
 return TokenType.*ELSE\_TOK*;  
 } else {  
 throw new LexicalException("invalid lexeme at row number " + (lineNumber + 1) + "and column" + columnNumber + 1);  
 }  
  
 }  
 }  
  
 */\*\*  
 \* Checks to see if the first character is a digit. It then checks the entire lexeme  
 \* to make sure there are no non-digit characters.  
 \*/* else if(Character.*isDigit*(lexeme.charAt(0))){  
 if(allDigits(lexeme))  
 return TokenType.*INT\_TOK*;  
 else  
 throw new LexicalException("Invalid lexeme at row number " + (lineNumber + 1) + " and column " + (columnNumber + 1));  
 }  
  
 */\*\*  
 \* Checks to see if the first character is a ", if it is it then assigns the lexeme to a String.  
 \*/* else if(lexeme.charAt(0) == '"'){  
 return TokenType.*STRING\_TOK*;  
 }  
  
 else {  
 throw new LexicalException("Invalid lexeme at row number " + (lineNumber + 1) + " and column " + (columnNumber + 1));  
 }  
 }  
  
 return tokType;  
 }  
  
 */\*\*  
 \* Checks a lexeme to determine whether it is a valid integer.  
 \** ***@param*** *s  
 \** ***@return*** *true if all digits  
 \*/* private boolean allDigits(String s){  
 assert (s != null);  
 int i = 0;  
 while (i < s.length() && Character.*isDigit*(s.charAt(i)))  
 i++;  
 return i == s.length();  
 }  
  
 */\*\*  
 \* Finds a lexeme by using whitespace around unbroken characters.  
 \* If lexeme starts with " it then goes through the string until it  
 \* finds the next " and assigns the lexeme from the first " to the next.  
 \** ***@param*** *line  
 \** ***@param*** *lineNumber  
 \** ***@param*** *index  
 \** ***@return*** *\*/* private String getLexeme(String line, int lineNumber, int index) {  
 assert (line != null && lineNumber >= 1 && index >= 0);  
 int i = index;  
 if (line.charAt(index) == '"'){  
 i++;  
 while(line.charAt(i) != '"'){  
 i++;  
 }  
 i++;  
 }else {  
 while (i < line.length() && !Character.*isWhitespace*(line.charAt(i)))  
 i++;  
 }  
  
 return line.substring(index, i);  
 }  
  
  
 */\*\*  
 \* Skips over unnecessary whitespace.  
 \** ***@param*** *line  
 \** ***@param*** *index  
 \** ***@return*** *\*/* private int skipWhiteSpace(String line, int index) {  
 assert (line != null && index >= 0);  
 while (index < line.length() && Character.*isWhitespace*(line.charAt(index)))  
 index++;  
 return index;  
 }  
}

LexicalException.java

package com.Potchen;  
  
*/\*\*  
 \* Created by Joe on 2/22/2018.  
 \*/*public class LexicalException extends Exception{  
  
 LexicalException(String string){  
 super(string);  
 }  
  
  
}

TokenType.java

package com.Potchen;  
  
*/\*\*  
 \* Created by Joe on 2/22/2018.  
 \*/*public enum TokenType {  
 *IF\_TOK*, *ASSIGN\_TOK*, *WHILE\_TOK*, *PRINT\_TOK*, *LESSEQ\_TOK*, *LESS\_TOK*, *GREATEREQ\_TOK*, *GREATER\_TOK*, *EQ\_TOK*, *NOTEQ\_TOK*, *ADD\_TOK*, *SUB\_TOK*, *MULT\_TOK*, *DIV\_TOK*, *ID\_TOK*, *INT\_TOK*, *STRING\_TOK*, *REPEAT\_TOK*, *UNTIL\_TOK*, *DO\_TOK*, *END\_TOK*,  
 *THEN\_TOK*, *ELSE\_TOK*, *EOS\_TOK*}