CSE 340 Principles of Programming Languages
Fall 2014

Programming Assignment 1
Due on **September 11, 2014** by 11:59 PM

Abstract

The goal of this project is to give you some hands-on experience with implementing a lexer (lexical analyzer). You will be reading a plain-text file, splitting it in lines, then splitting each line into strings, and identifying the corresponding token for each string. The lexer will create a vector of tokens as a final result. Lexical errors should be identified and reported.

1. Instructions

a) Review the code included in this assignment, files: `Lexer.java`, `Gui.class`, and `Token.class`. Execute `Gui.class` and get familiar with its functionality, the program is able to recognize strings that are well-formed binary numbers and classify them with the token BINARY token. Also, the program recognizes DELIMITERS and some OPERATORS (**operators == and != are not recognized yet).**

b) The class `Lexer.java` does the following:

1. Receives as an input a plain-text document, in a String variable.

2. Splits the text document in lines, using `System.lineSeparator` character as a delimiter of lines. The `System.lineSeparator` character is system-dependent; on UNIX systems, it is "\n"; on Microsoft Windows systems it is "\r\n".

3. For each line, it reads the line character by character and concatenates the characters in a STRING, the largest possible one, until a delimiter is found. Delimiters include the traditional ones (such as ; : ,) and also white spaces, operators, end of line (`System.lineSeparator`), and some special characters (such as quotation marks).

4. For each STRING, the lexer should identify if it is correct (i.e., it is a WORD) or if it is an error. If it is correct then, a token should be related to it. A Token object is created for each string. The Token object contains information about the number of the line that contains the string, the token associated with the string, and the string itself. All Token objects are stored in a Vector. The name of the tokens should be in UPPERCASE letters. The label “ERROR” should be used, as the token name, for any string that does not match with any lexical rule.

5. The method **isDelimiter (String)** defines the full set of delimiters for our language. The method **isOperator (String)** defines all operators except == and !=. The methods

isQuotationMark (String) and **isWhiteSpace (String)** are used to search for single and double quotation marks and whitespace respectively.

c) Modify the class `Lexer.java`, be sure to use the DFA approach in order to make it able to recognize all of the following tokens:

TOKEN	
OPERATOR	`+, -, *, /, &, |, !, <, >, ==, !=`
DELIMITER	`: ; ( ) { } [ ] ,`
INTEGER	A base 10 literal, with digits consisting of the numbers from 0 through 9; for instance, 1975.
FLOAT	A floating-point literal. For instance, 2.5. They can also be expressed using E or e (for scientific notation). For instance, 1.245e2.
HEXADECIMAL	A base 16 literal, with digits consisting of the numbers from 0 through 9 and the letters from A through F. They start with the prefix 0x. For instance, 0x1A.
OCTAL	A base 8 literal, with digits consisting of the numbers from 0 through 7 preceded by a zero digit. For instance, 016.
BINARY	A base 2 literal, with digits consisting of the numbers 0 and 1. They start with the prefix 0b. For instance, 0b11010.
STRING	A series of characters inside double quotation marks. For instance, “This is a string”.
CHAR	A single printable character in a pair of single quote characters; for instance, ‘a’. There are some particular characters, such as tab, quotation mark, and single quote that are represented using an escape sequence. An escape sequence begins with a single backslash and then a character; for instance, ‘\t’, ‘\’’, or ‘\”’.
IDENTIFIER	An unlimited-length sequence of letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_". IDs are case-sensitive.
KEYWORD	if, else, while, switch, case, return, int, float, void, char, string, boolean, true, false, print.

The modifications in Lexer.java include:

1 Adding new constants. They are used to associate a column number with a word. Therefore, facilitate the access to the table of states. For instance, you will need to add constants for A_Z, 0_9, A_F, E, DOT, among others.

2 Adding new states to the table of states (stateTable). Define the rules for the tokens mentioned before using a DFA and then express the DFA as a transition table. That is the table that you should use here.

3 Updating the methods **run()**, **splitLine()**, and **calculateNextState()** as needed.

d) Test your lexical analyzer running the file Gui.class. Gui.class will show a graphical user interface as shown in the Fig 1. You can load an input text file or write your input in the Editor panel. Then choose run/lexer from the menu bar.

e) Examples of results that you should get are the followings:

Input

```
5.
"hello world this is an errror
0b00000000001111111111111;hello
00
0xJAVIER
CSE_340
$20
'''
'\X'
23.23E-23
23.23E-23.23:_
2014AUGUST
```

line	Token	String or Word
1	FLOAT	5.
2	ERROR	“hello world this is an errror
3	BINARY	0b00000000001111111111111
3	DELIMITER	;
3	IDENTIFIER	hello
4	OCTAL	00
5	ERROR	0xJAVIER
6	IDENTIFIER	CSE_340
7	IDENTIFIER	$20
8	CHARACTER	'''
9	CHARACTER	'\X'
10	FLOAT	23.23E-23
11	ERROR	23.23E-23.23
11	DELIMITER	:
11	IDENTIFIER	_
12	ERROR	2014AUGUST

c) Read and actively participate in the discussion board about common errors in lexical analysis and use those discussions to improve your implementation.

d) Create a zip file, using the following naming convention: **Firstname_Lastname_P1.zip.** This file should contain your source code for:

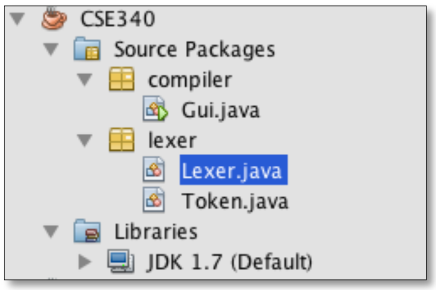
Lexer.java

And should include a compiled version of your application:

CSE340.jar.

2. Source Code

We are providing you with an initial version of the Lexer. The source code is structured as described in the following paragraph and shown in the figure below.



The application name is CSE340; therefore, it compiles in a JAR file called `CSE340.jar` and that is the file you should run to execute the application.

The application has two packages (compiler and lexer). The package compiler has one class (`Gui.java`) and the package lexer has two classes (`Lexer.java` and `Token.java`).

Your Lexer should be implemented in the class `Lexer.java`. **YOU DO NOT NEED TO MAKE ANY CHANGE IN THE CLASSES `GUI.JAVA` NOR IN THE CLASS `TOKEN.JAVA`. In case that you detect any bug in those files, report it on the discussion board on Blackboard**.

Details about the code are going to be described in lecture 5 (September 4). The graphical user interface of the project as well as a list of the code included in each file is included in the lecture 5 slides.

Set up your IDE to be able to work with those files; if you are not using an IDE then read the following tutorial to learn how to work with JAR files in Java.

http://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html

The code has been compiled and tested using Java 1.7.

3. Grading

The assignment will be graded in a scale of 0-100; each item in the following list gives you five points:

- Conventions in format, naming, and packaging are followed
- The application is able to compile and run, recognizing Binary numbers (that functionality is the one already implemented in the source code)
- Recognize Integer (e.g., 798)
- Recognize Integer Error (e.g., 7abc)
- Recognize Float (e.g., 1.98)
- Recognize Float error (e.g., 75.23.13)
- Recognize Hex (e.g., 0x28A) and Octal (e.g., 02)
- Recognize Hex error (e.g., 0x29FG) and Octal error (e.g., 09)
- Recognize String (e.g., “Hi World”)
- Recognize String error (e.g., “Hello)
- Recognize String with \ (e.g., “\”a”)
- Recognize String with \ error (e.g., “abc\”)
- Recognize Char (e.g., ‘a’)
- Recognize Char error (e.g., ‘hi’,’’, ‘a)
- Recognize Char with \ (e.g., ‘\’’, ‘\\’, ‘\t’)
- Recognize Char with \ error (e.g., ‘\’, ‘a\’)

- Recognize IDENTIFIER (e.g., x, foo)
- Recognize IDENTIFIER error (e.g., @foo)
- Recognize Keywords
- Recognize Delimiters and Operators

Be aware that:

- No credit will be given to programs that do not compile.

- No credit will be given if the project does not follow the format described above (input, output, number and structure of classes, attributes, and methods).

If you have any question, contact us (me or the TAs) to clarify them. And remember, actively participate in the discussion board.