CS 352 – Spring 2011
Project 1

Handed out: Jan. 17 Due: Feb. 3 (11:30pm)

All projects should be done individually instead of in groups.

In Project 1, you manually implement a lexical analyzer for the MiniJava language specified in the Appendix of the textbook. As explained in the class, you use a while-loop to scan the source code. The loop body is mainly a table lookup. The table implements the finite automata you build for different types of tokens.

You can use some techniques to keep the table small. Here are a few hints. All the key words can be handled separately by a switch statement without having to loop up the table. Characters which have the same effect can be lumped together, e.g. all English letters. Thus you do not need a column in the table for each letter.

1. The output of your code:

(1) First, print out a list of identifiers which appear in the source code. Each identifier should be printed exactly once, in the first-appearance order. Each identifier should be followed by the line number(s) in which it appears. The line numbers should be put in parentheses. For clarity of the result, please start a new line for each identifier printed.

In order to remember the identifiers, you need to implement a symbol table in your lexical analyzer. Whenever an identifier is recognized, the symbol table is checked to see whether it is already there. If not, the ID should be inserted. The line numbers are stored in the symbol table as well. Note that keywords such as "System", "out", and "println" should NOT be treated as IDs.

(2) Next, print out a list of integer constants found in the source code. Constants of the same value should be printed once only. You do not need to print out the line numbers for this. To implement this, you need to maintain a “literal table” similar to the symbol table.

(3) Print the total number of all binary operators appearing in the source code.

(4) Lastly, report invalid tokens in the source code: any use of symbols not included in the specification (eg:- @ $ % /) should be recognized as invalid tokens.

(4) Important note about comments: For the form of comments which starts with /* and Ends with */, we assume the comments are **not** nested. (Nested comments will need techniques more powerful than finite automata to process, as will be explained in Chapter 3.)

2. How your code should be organized

(a) Your source code should all be in one file called “Lexer.java”.
(b) Your program should take in one command-line argument that is the name of the input text file that you are tokenizing.
(c) Your program should be usable as follows:

TA runs: `javac Lexer.java`
(this creates “Lexer.class”)
TA runs: `java Lexer inputTextFile.txt`
(this runs your program on “inputTextFile.txt”)

3. What to submit

Only your “Lexer.java” file

4. How to submit your work

You should turn in the Lexer.java file using the ***turnin*** command available on CS Unix machines.

To submit: turnin -c cs352 -p p1 Lexer.java

To verify: turnin -v -c cs352

5. Grading Criteria:

For this project, we will create a set of test cases and then compare the output of your solution to the key. Two sets of sample inputs and outputs are provided for your convenience. No partial credit will be given on failed test-cases. The only exception is if failure of one test-case triggers failure of other test cases (test-cases can't all be independent, but we'll try as hard as we can).

Any doubts concerning the project requirements should be raised by sending email to lib@cs.purdue.edu. The TA may then discuss things with the instructor if necessary.