Comp 333 Project #2 (Lexical Analyzer Project)

**Due: March 1**

**Points: 30**

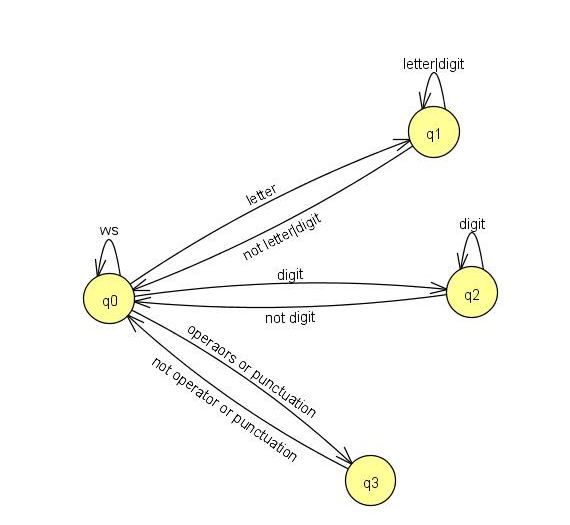
**You may work in teams of 2 or 3 or you may do an individual project. Each team should submit only one project. If you form a team, complete an index card distributed by your instructor that lists the team members.**

1. In this project you will create a lexical analyzer for the following grammar using a finite state automata methodology.

P 🡪 begin <stmt\_list> end   
 <stmt\_list> 🡪 <stmt> | <stmt> ; <stmt\_list>  
 <stmt> 🡪 <assign\_stmt>   
 <assign\_stmt> 🡪 <id> := E   
 <write\_stmt> 🡪 write <id>

<read\_stmt> 🡪 read <id>  
 E🡪 E + T | E - T | T  
 T 🡪T \* F | T / F | F  
 F 🡪 ( E ) | <id> | <int>   
 <id> 🡪 <letter> { <letter> | <digit> }\*  
 <int> 🡪<digit> { <digit>}\*  
   
Your lexical analyzer should find the tokens in an input string and return a token list that that labels the tokens as to their type. The exact format of the input and output is listed below in item 3c. If your program finds an unidentified character it should print a useful message (character and position number of the char) and exit the program. The input string ends with a ‘$” so you tell where the program ends.

1. Use the following finite state automata to identify the tokens. The structure of your code must reflect this finite state automata. We will discuss how to do this at length in class.



1. A token object has two fields (1) kind and (2) the value of the token if the token kind is an “int” or an “id”

For example: [“int”, “897” ] [“id”, “book”] [“begin”, “”] [“end”, “”] [“read”, “”] [“write”, “”] [“assign”, “”] [“plus”, “”] [“minus”, “”] [“times”, “”] [“divide”, “”] [“semicolon”, “”] [“lparen”, “”] [“rparen”, “”] . There are no other kinds of tokens.

1. Use object oriented programming to develop your program;
   1. Use the attached Token class that stores tokens. Do not modify the Token class.
   2. Complete the attached Lexer class. You may need to add additional private methods and fields.
   3. Add useful error messages.
   4. A typical test case that has no errors will look like this:

String program = " begin read a12b4c ; book := 62 ; write book end $";  
 Lexer lex = new Lexer( program);  
 System.out.println("Input to Lexer: " + program);  
 ArrayList<Token> tokenList = lex.scan();  
 System.out.println("Output tokenlist: " + tokenList);

OUTPUT  
Input to Lexer: begin read a12b4c ; book :=62 ; write book end $

Output tokenlist: [begin, read, id("a12b4c"), semicolon, id("book"), assign, int(62), semicolon, write, id("book"), end]  
Ï

1. Test your program thoroughly. Test good and bad strings, not just the examples below. You do not need to turn in these test cases. Your instructor will run your program on a set of test cases which will not be known to you. At least 50% of your grade will be based on how well your program runs on the instructor test cases. [Suggestion: Appoint someone on your team to thoroughly test your program.]
2. **Submit your Lexer.java program to Moodle on March 1 by 7:30 am**. All team members must be listed at the top of the source code . Your program must be named Lexer.java and meet the conditions of the project.
3. **Hand in in class on March 1: (stapled together in this order)**
   1. Title Page with course , project, date and team member names.
   2. Lexer.java source code
   3. Results for your program run on these test cases. Label test cases.

// Test Case 1–" begin read a12b4c; book := 62 ; write book end $"

//Test Case 2 – ″ a :=b+c 5009\*6 := c ; read 80 end $″

//Test Case 3 – ″ a := b & c 5 \* 6 = c end $″ 🡨 bad one

//Test Case 4 - ″ a := b + (c + 1); apple end $”

//Test Case 5 - ″ a : = b\*( a + s) $″ 🡨 bad one

//Test Case 6 - ″ a := b\*( a \* ) + s) ″ 🡨 bad one

// Use Token class as is. Do not make any changes to it

public class Token

{

String kind;

String value ;

public Token( String t, String val)

{

kind = t;

value = val;

}

public Token( String t)

{

kind = t;

value = "";

}

public String getKind()

{

return kind;

}

public String getValue()

{

return value;

}

public String toString() //prints one token

{

String s = "" + kind;

if( kind == "id")

s = s + "(" + "\"" + value + "\"" + ")";

else if (kind == "int")

s = s + "(" + value + ")";

return s;

}

}

import java.util.\*;

//Complete the Lexer class. You can add more private methods and fields. //Do not change the signatures of the public classes

import java.util.\*;

import java.io.\*;

public class Lexer

{

private String progText; //holds the input string

private int nextCharPtr;

private char nextChar;

private ArrayList<Token> tokenList; //holds list of tokens found

public Lexer( String p)

{

}

private char getNextChar()

{

}

private void pushBackChar()

{

//optional but might be useful

}

public ArrayList<Token> scan()

{

//This is where the finite state machine algorithm goes

//See next page

}

public static void main( String[] args)

{

//Typical Test Case –

String program = " begin read a; book := 62 ; write book end $";

Lexer lex = new Lexer( program);

System.out.println("Input to Lexer: " + program);

ArrayList<Token> tokenList = lex.scan();

System.out.println("Output tokenlist: " + tokenList);

}

}

//PARTIALLY COMPLETED scan() method that implements to finite state automata

public ArrayList<Token> scan()

{

System.out.println("Start Scanner");

int state = 0;

while( nextChar != '$')

{

switch(state)

{

**case 0: // state 0**

{

char x = getNextChar();

while( Character.isWhitespace(x) )

x = getNextChar();

if( Character.isLetter(x)) state = 1;

else if(Character.isDigit(x)) state = 2;

else state = 3;

break;

}

**case 1: //state 1**

{}

**case 2: //state 2**

{}

**case 3: //state 3**

{}

} //end switch statement

}//end while

System.out.println("Scan completed");

return tokenList;

}