CSC 3310 Lab 1 CalcLex
Due: Wednesday, November 16, 2016 11:59pm

Design and implement a Visual Studio C++ "console" application CalcLex that implements a scanner function that recognizes the tokens in the Calculator language:

A Calculator program is a sequence of one or more statements.
A Calculator program is entered in a "free format" style, allowing a statement to be split across multiple physical lines, or multiple statements on a single line.
Comments must fit on a single line, and can appear between tokens on a line.
Identifiers are not declared, but just come into existence when they are first encountered.

- 3 statement types: Assignment (id := expression)
  read id
  write expression
- Expressions: normal infix expressions with nested parentheses, operators +, -, *, /
  identifiers (L (L | d | _)*), and numeric constants (d^+ or $d^+.d^+$)
- Comments: /* ….. */

Calculator Grammar

```
<program>       → <stmt_list>   $$
<stmt_list>     → <stmt>  <stmt_list>  |  ε
<stmt>          → id := <expr>  | read  id  |  write  <expr>
<expr>          → <term>  <term_tail>
<term_tail>     → <add_op>  <term>  <term_tail>  |   ε
<term>          → <factor>  <factor_tail>
<factor_tail>   → <mul_op>  <factor>  <factor_tail>  |   ε
<factor>        → ( <expr> )  |  id  |  number
<add_op>        → +  |  -
<mul_op>        → *  |  /
```

First, analyze the language/grammar and determine a list of all possible valid token types in the Calculator language (including a special end-of-file token). These will include the operators, punctuation symbols, keywords, identifiers (letter followed by letters&digits, where letter can be upper or lower case or the underscore character), and numeric constants (integer or fixed-point decimal value, e.g., "2", "2.0", "123.456"). Identifiers and keywords are not case-sensitive (e.g., "read", "Read", "READ" all represent the same token).

Create a list of symbolic Names, one for each type of Token. Assign a small integer value to each token type using #defines or an enumerated type, e.g.,

enum CALCTOKENS { EOFSY=0, LPAREN, …, ADDOP, … , ID, NUMCONST, READSY, … };

The scanner must ignore and skip over white space characters (' ', '\t', '\n') between tokens. Also, the input file may contain comments of the form: **/* … */**. However, any comment must fit on a single line (if \n is encountered before the */, then just terminate the comment). Be careful to properly recognize a comment such as **/****** My Comment ******/**.
Note that comments and white space are not "tokens"; the scanner should process and ignore them.

Your scanner must be implemented as a function (e.g., int gettoken(. . .), or yylex(. . .)) that reads through a specified input file, extracting out tokens one at a time. Each time the scanner function is called, it will read through the next portion of the input file until it recognizes and extracts a valid token, and then it returns the integer value of the next token value (one of the enumerated CALCTOKEN token type integer values) and a TokenString string variable containing the literal characters that comprise the recognized token value. Some tokens have obvious TokenString values (e.g., the + operator has string value "+", or the READSY keyword has string value "read"), but identifiers and integer constants have programmer-defined literal representations.

Also, since keywords and identifiers have string values that must be treated as case-insensitive, its OK to design the scanner to recognize letters as either upper or lower case, but to then immediately convert them to all-upper (or all-lower) case before returning the TokenString parameter value.

You can design the scanner function to use explicit parameters for the input open-file object and the output TokenString value, or you can implement these as global variables that are simply referenced as needed by both main() and the scanner function (oftentimes in real compilers, global variables are used to minimize the execution overhead incurred by parameter passing).

Design a console-application **`int main(int argc, char *argv[])`** main program that obtains an input filename character string as a <u>required</u> argument on the command line. main() initially opens the input file and then repeatedly calls the scanner function until all tokens have been processed. For each input token recognized and returned by the scanner, main() must output a line indicating the CALCTOKEN value and the associated TokenString value for the recognized token. At the end of the input, output a summary totaling the number of tokens recognized.

You'll find on Canvas the "EGrammar" notes with details and code examples.

<u>Turn in</u>:
Submit via Canvas a ZIP of your Visual Studio C++ CalcLex project directory.

➔ Before ZIP'ing the directory, please do a Build/Clean within Visual Studio, and manually delete the very large *.sdf file and the ipch directory within the project.

As an example, the input file t1.calc

```
/* t1.calc */
read A      /* Input the values */
read B

/*****Perform the computations*****/
sum := A + B
write sum
write sum/2.0
```

when executed from the commandline as

```
C:-> CalcLex   t1.calc
```

might produce the output report

```
tok = 10 READSY (read)
tok = 08 ID (A)
tok = 10 READSY (read)
tok = 08 ID (B)
tok = 08 ID (sum)
tok = 01 ASSIGNOP (:=)
tok = 08 ID (A)
tok = 04 ADDOP (+)
tok = 08 ID (B)
tok = 11 WRITESY (write)
tok = 08 ID (sum)
tok = 11 WRITESY (write)
tok = 08 ID (sum)
tok = 07 DIVOP (/)
tok = 09 NUMCONST (2.0)
tok = 00 EOFSY-$$ ()

Number of tokens = 15
```