CPSC 390 – Programming Project #1 – A Lexical Analyzer
Due, Friday September 30, 2011 by Midnight.

A compiler has three basic interacting parts: a lexical analyzer (which breaks a source program – a string of characters – into *tokens*), a parser (which generates the derivation tree for the sequence of tokens), and a code generator (which uses the parse tree to generate code, often in some intermediate form before actually generating object code). Tokens represent meaningful sequences of characters in the given language. For example, the following C++ (or Java) statement

```
count  = count + 50;
```

would be broken down into the following tokens: an identifier (count), the assignment operator (=), an identifier (count), the addition operator (+), the integer literal (50), and the semicolon (;). Generally the grammar for the language is expressed in a form where the tokens are the terminals and the variables are groups of tokens. In the compilation process, the parser calls the lexical analyzer when it needs a new token. The lexical analyzer returns the type of the token (e.g., identifier or integer literal) which is usually some code (often an integer) and a pointer to other information about the token (generally this is a pointer to an entry in a symbol table that contains information about the token – for example, if the token is an identifier it would contain the sequence of characters that make up the identifier).

A lexical analyzer (often called a lexer) is basically the same for all languages and can be implemented as a table-driven finite state machine. The only thing that changes is the table that represents the transitions and actions.

For this assignment, you will write a lexical analyzer that breaks a C++ source file (containing only a small subset of C++) into tokens. The lexical analyzer should recognize the following tokens:

- identifiers consisting of a letter followed by 0 or more letters and digits
- integer literals
- string literals (a sequence of characters enclosed in double quotation marks – the quotation marks are delimiters not part of the token)
- floating point constants (a C++ floating point constant is a sequence of 0 or more digits followed by a decimal point followed by 0 or more digits where there must be at least one digit either before or after the decimal point OR a sequence of 0 or more digits followed by a decimal point followed by 0 or more digits (again at least one digit) followed by **e** followed by 1 or more digits OR a sequence of 1 or more digits followed by **e** followed by 1 or more digits)
- the following operators: =, +=, +, ++, *, *=
- left and right parentheses
- the semicolon

Your lexical analyzer MUST be table driven and MUST be written in C++. That is, it should use a transition table to move from state to state. The finite state machine should be implemented in a function *getToken* that finds a single token. It should find the token, its length, and its lexical type. It should be designed to return error information (perhaps through a special code for the lexical type) if an error occurs and to return information indicating the end of a line has been reached without finding a token. To implement the finite state machine the program cannot use a "pure" DFA. It must have the capability to look one character beyond the end of the token (otherwise it won't know it is at the end). This requires that the program also be capable of performing some actions based on what the character is (and what state it is in). For example, the next character may be part of another token so that character would need

to be pushed back into the input stream or it could be a blank space delimiter in which case it can be ignored.

So, to implement a finite state machine that recognizes (and classifies) each of the above tokens you need to do the following:

1. draw a transition graph to recognize the above tokens
2. create a transition table from your graph
3. decide what action goes with each transition (some typical ones are – add the character to the current token, push the current character back into the input stream and classify the current token, ignore the current character and classify the current character)

The outline of the algorithm is as follows:

```
Outline of getToken
currentState = initialState
while "not done" do
    get the appropriate action from the transition table – index into the table
                    based on the current state and current symbol
    perform the action
    set currentState equal to the next state (get this from the transition table)
end while
```

The outline of your overall program should be:

```
initialize the transition table
while not at the end of the source file do
     read in a line from the source file
     while not at the end of the input line do
          call getToken to get a token
          print the token, its length, and lexical type (or an error message)
     end while (not end of line)
end while (not end of file)
```

You MUST manage your own input buffer (this is no big deal!!!). That is read a line from the source file into an array (that's the buffer) and use a pointer (an integer index variable is all you need) to keep track of the current symbol (or next symbol – whichever you want to do) to be processed (thus, "pushing back" merely means decrementing your pointer).

REQUIREMENTS: Your program must use good programming techniques. Design appropriate classes with appropriate member functions and be sure to use constants to avoid "magic numbers". The program must be well documented. Each function (method) must clearly explain what each parameter represents and any assumptions about the state of each parameter or the state of the object (the PREcondition for the function) and must document all changes made by the function to the member data of the class (the POSTcondition) including any return values. The declaration of each variable or data member of a class should include a brief description of the role of the variable or data member in the program. A Makefile must be included with your program. Prompt the user for the source file name (don't make it a command line argument).

HAND IN: A **neat** drawing of the finite state machine your program is based on (one that accepts the tokens listed above). It should include the state numbers or names that you use in your program. Submit a tar or zip file containing all of your files to Inquire.