Computer Science 426 --- Fall 2014
Stage 2 --- The Scanner

1. Administrative Matters

Make a copy of your “stage1YourLastName” folder and then rename the copy as “stage2YourLastName”. Then complete the project in the stage2YourLastName folder. This project is due at 5pm on Wednesday 9/24. When you are ready to turn in your project, upload it to Moodle and send me an email to confirm that it is done.

2. Underlying Assumption

You may assume that the input to your compiler is being prepared on a terminal that can't generate lines with more than eighty characters in them. So you do not need to worry about what happens if you feed your compiler a file with a line of length greater than eighty.

3. Designing the FSA

This part of the homework should be done with one or two partners. If you need help finding a partner let me know.

Everyone in the group should understand the entire solution you turn in, and you should turn in one solution on behalf of the entire group.

Your FSA is due as a homework (it will count as a homework grade) at 3pm on Tuesday 9/9. This part of the assignment will count as a homework grade. You can hand-draw the assignment, or you can draw it in DyKnow or with some other tool. If you do the work electronically you can email it to me as an attachment.

Your specific task is to design an FSA that accepts the language of all valid YASL lexemes. The FSA should treat YASL keywords as identifiers (they will be distinguished in the next step of this assignment.) The FSA does not need to detect errors due to identifiers, integer constants, or string constants that are too long as we will save that for the next step of this assignment. However it should deal with all the other errors and points described in the Lexical Information Section of the YASL Language Definition Document. This includes dealing with compiler directives, comments, and lexical errors other than those specifically excluded earlier in this paragraph.

Represent your FSA as a transition diagram (finite state machine). You will find it helpful to place the start state in the center of the page and to let arcs leave it in all directions as this will give you more room on the page. Use a double-circle to indicate final states and use one final state for each token. Number each of the states, using 0 for the start state.

For reasons that will become clear later, it will be **helpful to use the highest possible numbers for the final states, and perhaps even to skip at least twenty values between the last value you use for a non-final state**, and the first one you use for a final state. One way to do this would be to use consecutive integers starting at 0 for non-final states, and integers starting with 100 for final states. You may label the arcs in your diagram with one or more of the characters from the YASL alphabet. Additionally you may use the following abbreviations if you like:

ws = white space (space, tab)
cr = carriage return
l = upper/lower case letter,
d = digit,
EOF = end of file marker,
other = any character not specified on another arc leaving that state.

Traditionally, the abbreviation **ws** means space, tab, or carriage return, but you will find it useful to have a separate category for carriage return as shown above. The reason is that in some situations (such as in the middle of a string constant) a "cr" is treated differently than other types of whitespace.

As part of your FSA design, make a list of each state in your diagram. Next to each state write the action(s) (there may be more than one) that must take place when that state is entered. Possible actions (there may be others you want to use) include the list below:

Accept: TYPE = RELOP_T, SUBTYPE = EQUAL_ST
Accept: TYPE = COMMA_T, SUBTYPE = NONE_ST
Accept: TYPE = IDENTIFIER_T, SUBTYPE = NONE_ST
PushBack (push one symbol back into input buffer)
Error-Exit-Program:
    Message = Illegal symbol <symbol> encountered.
    Use printCurrentLine to print line # and line contents.
React-To-Compiler-Directive: Turn printing on.

From past semesters I know students sometimes worry about specifying the actions (see above) "correctly". If this happens to you, relax -- there is really nothing special that I am looking for. The actions are really notes that will help you when you do the C++ part of this project. The format above is a reasonable format to use but there are no rigid rules. It is much more important to draw the FSA correctly than it is to specify the actions in some particular way.

4. Implementing a Lexer (Scanner) to perform the Lexical Analysis stage of your compiler:

This is the part of the assignment that your project grade will be based on. However, if there are mistakes the FSA design then there will also be mistakes in the parser, so indirectly the FSA will be part of this grade.

Since you may be coming out of a group experience from the first part of this assignment, I want to remind you that I expect each student to write his/her own compiler. Doing side by side programming where you continuously ask each other "what line should we write next" then you implement that line, and then say "what should we do next" is not acceptable. You are welcome to discuss general ideas and you are welcome to develop high level plans together. You are also welcome to answer questions for each other (including helping each other to find bugs, discussing code fragments, etc). However, this is different from actually designing and writing the programs together line-by-line. If the distinction is not clear please come and see me. No two people should be turning in essentially the same solutions to these projects.

Begin by revising the main function so it can be used to test your scanner. A reasonable test program would be based on the following:

```
int main()
{ scannerClass   scanner;     //Note that we  no longer include the file manager!
  tokenClass theToken;       //You can define the tokenClass in scanner.h
  do
  { theToken = scanner.getToken();
    theToken.display();
  } while (theToken.getType() != EOF_T);
  scanner.closeSourceProgram();
  //add code here to print the number of lines compiled by YASLC-XY
  return (0);
}
```

You will need a switch statement (or sequence of if statements) to do the printing in the display() function for the token class, so that you can print lines such as:

```
>      RELOP_T    GREAT_ST
cat    IDENT_T    NONE_ST
```

rather than just printing the numeric codes for RELOP_T etc. You might want to write a function that takes a numeric code as a parameter and prints the correct text in response. This will be useful to you later in the project for debugging. Note that it is not acceptable to simply print the numeric codes! Don't worry about getting the columns to line up in the output - that is not crucial.

Next add files named scanner.h and scanner.cpp to your project. These should be the primary files that contain your work for this stage; however, if you find it helpful to add other files to the project that is fine. You are free to organize your work as you see fit.

This file scanner.h should define a class named **TokenClass** and then one named **ScannerClass**.

The token class will have private data elements for the token type (an int), subtype (an int), and a string lexeme value.

The scanner class will have a private data field which is an object of type **fileManagerClass** from project 1. Notice how we have added a level of abstraction to the main() program. The file manager is now hidden from the main program.

When writing the scanner class your main task will be to implement the getToken() method which is called in main as shown above. In order to do this, the scanner class will need to have additional private instance fields to store the lexeme. It will need a constructor that initializes the lexeme to be empty and perhaps does other things. The class will also need a method that can be called to close the file when the driver program is about to terminate (this new member function will simply call the appropriate member function from the fileManagerClass).

As you implement the method named scanner.getToken(theToken) I suggest that the basic flow of getToken might be as shown below. Of course, **getToken should be modular** and may call other methods.

```
state = 0
do
{       get an input character (use routine from file manager)
        state = table[state, input]
        switch (i)
                0  : call routine to perform the actions associated with state 0
                     break;
                1:   <so on for other states>
        end of switch
} while (1);    //Note this is an infinite loop... there will be return statements in
                //some of the cases of the switch statement however which will
                //terminate the loop.
```

You may find it helpful to start this stage by only worrying about a few tokens. For example, you might limit yourself to =, ==, <, >, and integer constants. Then you can test your work using an input file that contains only those tokens before moving on to finish the implementation.

Additional Details and Suggestions
a) Some additional hints and suggestions for issues to be aware of will be made during class. Scan your notebook and flag all of those issues that you think you need to keep in mind.

b) If a lexical error is encountered, your program should print an appropriate message. The message should consist of an explanation of the error (i.e. "String constant terminated incorrectly." followed by the line which triggered the error (the file manager already has a method that can be used to print the line). Be sure to account for all of the lexical errors described in the preliminary YASL definition.

There is one lexical error that is described in the YASL definition but which should never happen in practice. This error is an embedded EOF in a string constant. The reason this error won't come up, is because the file manager adds a CR to the end of each line. Therefore, there will always be a CR in the input stream before the EOF. Since the CR in the middle of the string constant will (correctly) trigger a lexical error, the EOF error will never be triggered. So, don't be concerned if you can't get this error message to appear when you are testing your code - although for safety reasons (in case the file manager is changed in the future) it would be best to account for the error in your code. Note that the lexical error caused by an EOF in the middle of a comment can occur. This is because a CR in a comment does not raise a lexical error.

c) You can define any constants you wish for the token types and token subtypes but I suggest that you use some convention like having the token type constants end in _T such as RELOP_T (or RELOPT if you don't like typing underscores) and ASSIGN_T while the subtypes end in _ST such as GREATER_ST   (GREATERST) and LESS_ST. A reasonable place to define these for now would be in scanner.h. It would also be reasonable to define a separate .h file which contains these definitions.

d) The easiest way to distinguish keyword from identifiers is to add code to the Identifier state to do this. Before announcing an identifier, compare the identifier to the valid YASL keywords. If you find a match, return the appropriate values instead of the identifier token.

e) Your lexical analyzer should react to the single compiler directive which was defined in the preliminary language definition for YASL. If the source file contains compiler directives, the compiler should compiler should perform the appropriate action. Be sure you deal with the lexical warning that can be raised by a compiler directive with an invalid character specifier. In that case, print an appropriate warning message ("Invalid compiler directive"), print the line that contains the problem, and then continue with the scan.