CS39003 Compiler Lab
Assignment: 4

Implementation of a Lexical Analyzer

Submission Deadline: 31st August 2013 **Marks: 40(15+15+10)**

Design a lexical analyzer in C to accept the language mentioned below. The analyzer should remove whitespaces from the input code, ignore the text within comments (anything enclosed within /* */) and recognize the tokens. The description of the lexeme patterns are the following

- Identifier: A string starting with an underscore or a letter and followed by any number of underscores, letters and digits.
  Identifiers with two leading underscores(__) are disallowed
- Keywords: **short sizeofint float double bool char signed unsigned for while do return structconst void switch break case continue goto long static union default**
- Signed and unsigned Integer constants: 12, 0, 3456, +56, -234 etc.
- Signed and unsigned Floating-point constant: 1.2, 4.25, -0.35 etc.
- Arithmetic operators: +, -, *, /, %, ++, --
- Assignment operators: =, +=, -=, *=, /=
- Relational operators: <,>, <=, >=, ==
- Special symbols: ; ( ) ,(comma) [ ] { }

This assignment is composed of the following three different parts (a), (b), and (c); each of them will be evaluated individually.The modules to be submitted are the following:

a. **FSM** - Hard code the finite state machine description functions for each of the abovespecified lexeme patterns. Write one function for each pattern which will recognize the corresponding lexeme.Same machine should recognize the keyword and identifier. However, the keywords must be handled separately. A priori, the keywords are to be entered in a symbol table and stored in the file "**symbol_table_1.out**". The symbol table must contain two fields which are used to store (a) the lexeme itself and (b) the marker (keyword or identifier). Keyword detection is to be done by referring this symbol table.

   Each of the functions, implementing FSM, reads the next token and does the following:

   i. **Return the token ID-**Token ID is a unique number assigned to each token and is specified in the header file **def.h**attached with the assignment. Remember to include this header file as is, in your code.
   ii. **Fill up the attributes of the token** - The definition of the attribute structure is as mentioned in the header file. This contains the exact lexeme of the token, in case the token is an Integer or Floating-point constant or an identifier (string). (A pointer to this structure is an input to this function and its contents are to be updated accordingly).
   iii. **Update the symbol table** – Once recognized, an identifier needs to be inserted in the symbol table (mentioned above) if it is not already present. Make sure that you are able to distinguish between keywords and identifiers in the symbol table. The symbol_table_1.out file should be updated after each such insert.

   A sample input file "**input.in**" is attached with the assignment. Your code should take it as input [redirect input from command line – no need to use file pointers]. It is preferred that you read the input into a buffer string and carry out the FSM simulation using this buffer (using Lexeme_Begin and Forward pointers). Simulate each of these FSMs on this input.

If all of these FSMs enter a halt state (without accept), then return the error code. Otherwise, return the token which has the longest prefix match with the input.

The output tokens should be written to a file "**a4_1.out**" in the following format:

- Each token's description is written in a new line in the same order as in the input file. On every line, there is a token ID (as mentioned in the header file), followed by a space, and then the attribute's value

  Examples:

▪ 300 value	//identifier with attribute "value"
▪ 301 234	// integer constant with attribute 234
▪ 302 23.4	// floating constant with attribute 23.4
▪ 303	//% symbol

The format in which the symbol table is storedin the file "**symbol_table_1.out**" is as follows:

- Each symbol's description is written in a new line. On every line, there is a symbol name followed by space and then a number - 0 or 1. The number is 0 if it is a keyword and 1 if it is an identifier.

  Examples:
    - value 1
    - switch 0

b. **NFA**–In this case, take as input the regular expressions as described in the sample file "**regexp.in**", attached with the assignment. Each line of this file contains the token ID and its associated regular expression, delimited by a space. The syntax followed is that used in lex for the identifier, integer_constant and floating_constant tokens. For the rest, the specifications are just symbols.

   Construct an NFA representation of the language mentioned in the input file. Use the methods of Thompson's construction to convert the above regular expressions to a single NFA. Next, simulate the NFA to recognize tokens. Remember to use the longest prefix match in case where multiple tokens are possible. The NFA should then be simulated on the input file **input.in.** Implementation of the NFA is up to you (either state transition table or graph representation). The output file **a4_2.out**is that similar as mentioned in (Part a) with each token in a separate line and the symbol tables updated in **symbol_table_2.out**

c. **DFA** - Convert the NFA constructed above to a DFA. Simulate the DFA on the input **input.in**and output the tokens recognized in the format as specified in Part (a) in **a4_3.out** and update the symbol tables accordingly in **symbol_table_3.out**

Note that "input.in" is just a sample file. Your code should work for any input program written following the specified guideline.

Submit the source code to generate the 3 executable – named FSM, NFA, DFA: one for each of the part questions. The following procedures will be used to evaluate the assignment:

input.in: Source code on which lexical analysis is to be done.

1. ./FSM < input.in
    a. Initialize symbol table

    b. Should read the symbol table from symbol_table_1.out
    c. Should output the tokens in a4_1.out
    d. Should output the updated symbol table in symbol_table_1.out

2. ./NFA < input.in
    a. Should read the regular expression file “regexp.in” in the current folder
    b. Should output the tokens in a4_2.out
    c. Should output the symbol table in symbol_table_2.out
3. ./DFA < input.in
    a. Should read the regular expression file “regexp.in” in the current folder
    b. Should output the tokens in a4_3.out
    c. Should output the symbol table in symbol_table_3.out

Readme file:
Also, submit a Readme file containing instructions to run your programs and to create the executables mentioned above. Submit a **Makefile** to do the same. If you make any assumption, explain that in details. Remember to stick to the executable and output file names mentioned here. Failure to do so might result in penalty. For Part b and c, explain briefly how you have implemented the NFA, DFA and their simulation.