CS323 Documentation – Assignment 1

1. Problem Statement

Write a lexical analyzer (lexer). Build the entire lexer using a FSM, or build using at least FSMs for Identifier, integer and real (the rest can be written ad-hoc/procedural).

The Lexer:

Write a procedure (Function) – lexer (), that returns a token when it is needed. Your lexer() should return a record, one field for the token and another field the actual "value" of the token (lexeme), i.e. the instance of a token.

The main program should test the lexer, reading in a file containing the source code given from class to generate tokens and write out the results to an output file. Both tokens and lexemes must be printed.

2. How to use your program

Running the program can be accomplished in MacOS or Linux by:

1. Open the terminal in the main file titled “Assignment1_JustinDrouin”
2. In the terminal **type ./Run MacOs** or **Linux ./RunLx**

It can also be compiled from the main file using the bash file:

1. Open the terminal in the main file titled “Assignment1_JustinDrouin”
2. In the terminal type **bash RunLex.sh for Linux** or **Mac type sh RunLex.sh**

3. Design of your program

Most of the work of the lexer program is done in the lexer.h header file. The main.cpp file initializes the token class which holds all the functions of the lexer. The infile and outfile is generated with the infile named “SampleInputFile1.txt” and outfile created named “tokenOutput.txt”. The CPP file checks if the stream from the infile is open and then calls the main lex function from the lexer header file that does most of the work.

In the lexer.h file the main data structures used are a single vector to hold identifiers for later retrieval, a two-dimensional char array for keywords, and two strings to hold both operators and separators, and a char buffer. There is also a two-dimensional array for the FSM, that holds all the states for validating the identifier. A majority of it is procedural however it is still less than 200 lines of code. Bool functions are used to identify operators, separators, and keywords. A flag system is also used to filter out large segments of text that are commented out.

The FSM validates the identifier by running a for loop on the word stored in the buffer total to its string length. With each iteration it calls the isValidIdentifier function passing the buffer, current state, and its index. After its done iterating a switch statement checks its final state and determines if it was valid or not. In the function there are 4 columns that are 2d array that are letter, digit, $, and invalid. The invalid column captures anything that is not otherwise a letter, digit, or $. A Switch statement is used to determine whether the char is a letter, digit, or $ and then uses its column number and current state to retrieve its next state.

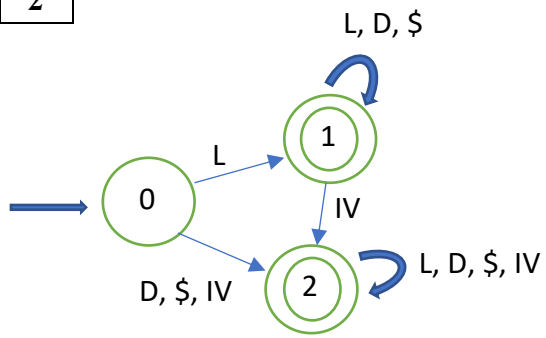
	0	1	2	3
	L	**D**	**$**	**IV**
-> 0	**1**	**2**	**2**	**2**
1	**1**	**1**	**1**	**2**
2	**2**	**2**	**2**	**2**

Σ = {L, D, $, IV}

Q = {0, 1, 2}

q0 = 0

F = {1, 2}



4. Any Limitation

There are no current limitations that I have found as of yet. Even though it is not all run through the FSM, it displays the tokens in order and correct syntax. Future version I will attempt to make the entire thing run through an FSM. Though a large chunk is procedural it is still under 200 lines of code. One limitation which can easily be changed though is the buffer size for the keywords and identifiers. It is set to 10, as no keywords or identifiers are larger than that, but if needed to can be easily increased.

5. Any shortcomings

I could not implement an FSM for the whole assignment, only for identifiers as my code was already built and fully functioning procedurally and transitioning the entire project over to a FSM would of meant starting from scratch. However future versions will likely incorporate a full FSM for the entirety.