CPSC 323 - Lexical Analyzer Documentation

By: Jason Duong

Date: Thursday, March 4, 2021

1. **Problem Statement**

The first assignment is to write a lexical analyzer program entirely using a FSM (finite state machine) or at least for identifiers, integers, and reals. The machine should return a token, the category type of a lexeme, and a lexeme, the instance of a token.

1. **How to use the program**

a. Using the Windows Command Prompt:

Extract the folder on the Desktop: FSM

Locate the extracted folder, FSM, from the terminal using ‘cd Desktop/FSM’.

Compile the program using: g++ -std=c++14 main.cpp -o main -Wl,--stack,268435456

Type, main.exe, to run the program

Note:

Enter only the test file names i.e. ‘valid1.txt’

After an input, press any key to exit the program

b. Using the Executable File

Extract the folder: FSM

Double click on the extracted folder

Double click on the executable file: main.exe

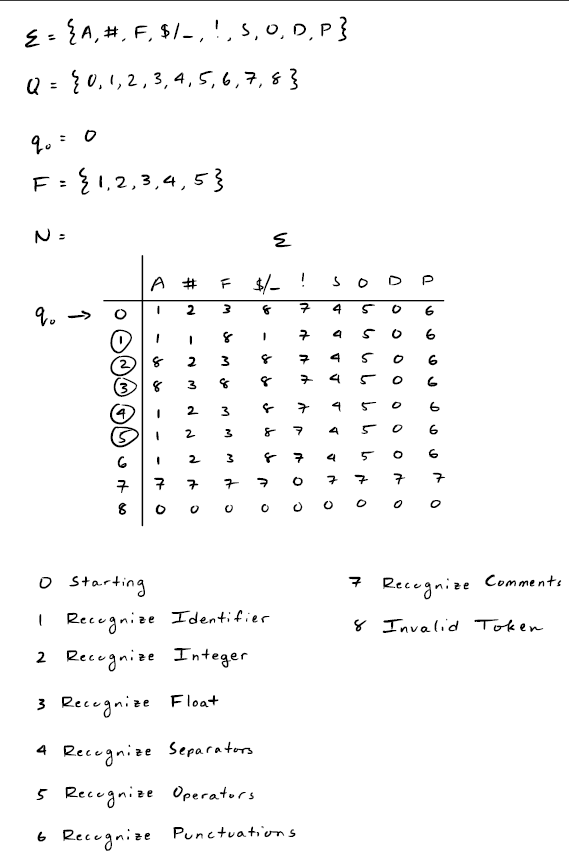
The extracted FSM folder should have a folder, test, which contains the seven files:

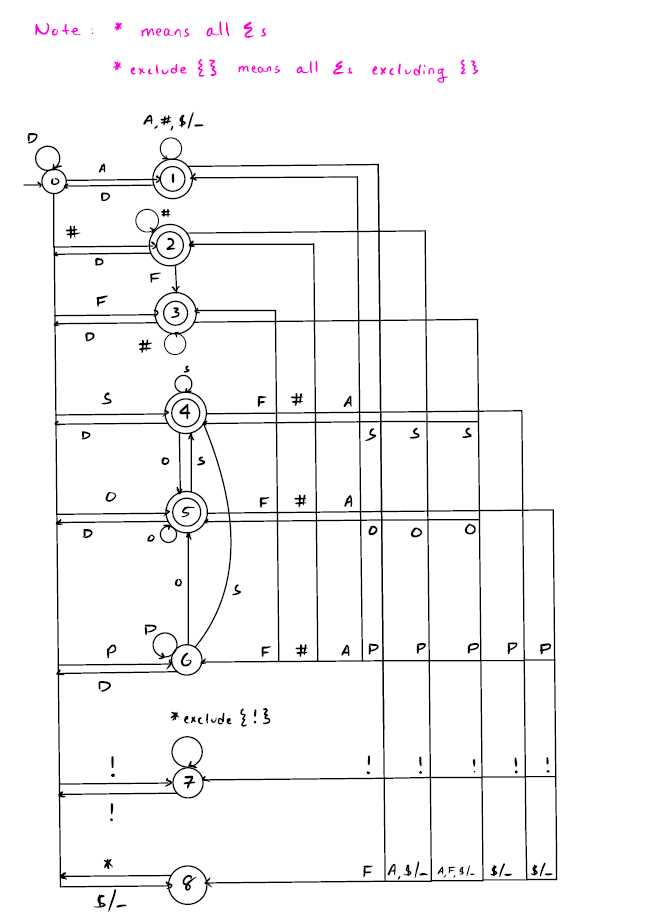
test folder, docs folder, error\_table.h, symbol\_table.h, lexer.h, main.cpp, main.exe

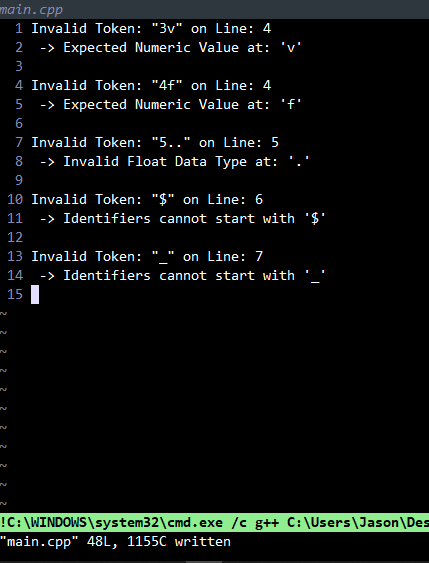
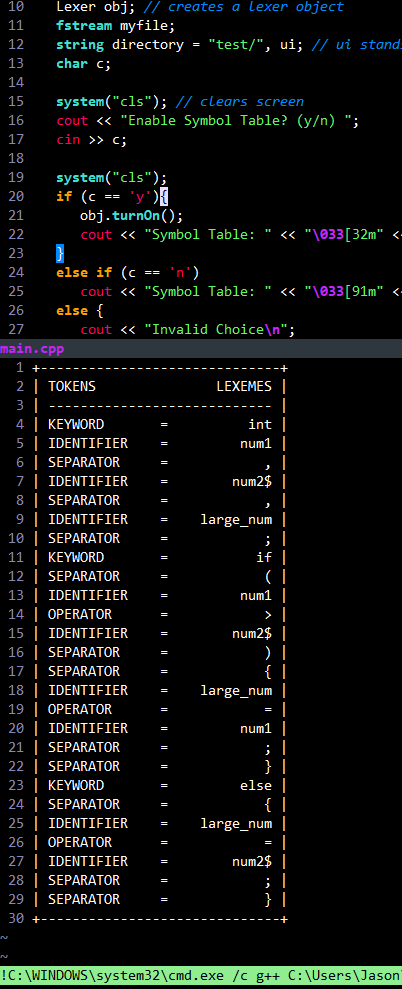
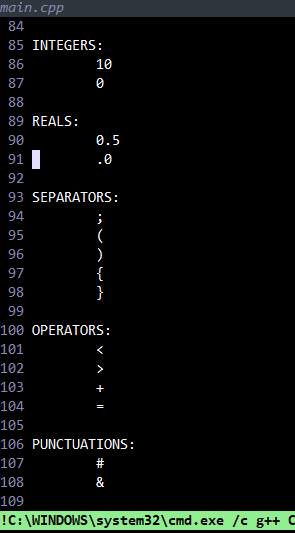
After running main.exe, it would prompt you to either activate the symbol table or not. After an input of ‘y’ or ‘n’, it should display a line of text: ‘Enter a Test File-Name: test/’. The default test file names are:

invalid, valid1, valid2, valid3, valid4

My program had already navigated into the test folder directory. Therefore, you would only need to type the test file names with the extension i.e. ‘valid1.txt’. After hitting the <Enter> key, a table structure should appear with two columns: Tokens and Lexemes except for the error file. It will display all errors occurred throughout the code even though some statements are valid.





****

1. **Design of the program**

Firstly, the data structures that are used in this program are only a vector and a queue container. The vector STL is used to hold the keywords and transition state. The transition state is an adjacency matrix, a vector of vectors. In the main.cpp file, you will see how each file is being read. The function, getline(myfile, ui), reads the file line by line and is stored in a string variable ‘ui’, short for user input. It will then pass that line into the FSM (Finite State Machine) function where each lexeme will be parsed and broken into their corresponding Tokens. Also, in the main.cpp file, it allows the user to enable or disable the symbol table prior to file reading. In the ‘lexer.h’ file, the major components are the FSM, getCol, isSign, and print functions. The FSM function will parse character by character from each line in the text file and concatenate them with a string variable, buffer. The FSM function will also access the tState adjacency matrix to get a new or existing state. Also, if the character being read is a delimiter, spaces, and tabs, it will reset the buffer since an identifier, nor an integer can be accepted with a delimiter. It will also check for the separators and operators and print them when necessary. Also, it would notify the error handler if an error occurred. The job for the error handler in the lexical phase is to determine the misspelling of a lexeme, in other words, an unrecognized token. The main job of the getCol function is to return an integer referring to the column number of the tState matrix. The getCol function checks for alphabets, numeric values, comments, underscore and dollar signs, reals, separators, operators, delimiters, and punctuations in the following order. The getCol function will also return a value corresponding to a dead state, 8, if an invalid token was found.

Certain characters will be passed into another function, isSign, to determine if the character is of the following values: Comments, Underscore and Dollar, Separators, Operators, and Reals. The purpose for dividing the process of getting the column number into two functions, getCol and isSign, is for ease of debugging. The isSign function is more of the computational checks for values while the getCol displays the return value of the column number. Lastly, the print function will use a switch statement to check only for the accepting states and print the Token and Lexemes accordingly. Notice, it will only print if no error occurs since an error will be passed to the error handler instead and execute another process. The print function will also add each token and lexeme to a symbol table where every lexeme and token will be recorded for future reference. If a lexeme exist, it will not be inserted. Lastly, there is also a function in ‘symbol\_table.h’ called display() which prints all unique lexemes. The symbol table can be turned on at the start of the program.

1. **Limitations**

The first limitation I had to make was for separators, operators, and punctuations. Since a separator, operator, or punctuation can be preceded by either a numerical or alphabetical value, it was hard to parse because all other lexemes require a space to end their token. Instead, it would check if the current character was either a separator, operator, or punctuation in the getCol function. This is before concatenation with the string buffer. If it was of those tokens, it will store the current buffer string and then clear it. This ensures the previous token was parsed and not forgotten. After concatenation with the string buffer, the FSM function will check if the current state is a 3, 4, or 8, which corresponds with separators, operators, and punctuations. If the states are 3, 4, or 8, the current buffer will get stored. However, this is different from the first time. After the first time, the buffered was cleared. Now the buffer has only one element, the separator, operator, or punctuation. Lastly, it will get cleared as usual and the next character will be read.

buffer = “main” buffer = “” buffer = “(” buffer = “” buffer = “)” buffer = “”

main() -> ( -> ) -> ) -> space -> stores “)”

stores “main” stores “(“

In addition, if two separator, operator, or punctuation were together i.e. () or {}, it will store the string buffer prior; however, the buffer is empty. Thus, there will be an extraneous line that will be printed. The fix was after checking for separators and operators in the getCol function, it will also check if the states are equal to 3, 4, or 8, which corresponds to separator, operator, and punctuation. If they are not equal, then it will proceed with storing the prior buffer string. If not, it will only return the column number.

The last limitation is when the last Lexeme of the line is being read. Normally programmers do not place spaces at the end of each line. However, my program only prints when a delimiter occurs as it marks the end of a token and will transition back to the starting state. Inside the FSM function, it will check at the end if the state is a dead state, 8, and if the buffer is not empty. The reason for checking if the buffer is empty or not is because of separator, operator, and punctuation. The separator, operator, and punctuation will be emptied. So, checking if the buffer is not empty will ensure the separator, operator, and punctuation will not be printed twice.

1. **Shortcomings**

None