**CS323 Documentation**

**1. Problem Statement**

Create a Lexical Analyzer (lexer). Build an FSM for identifier, integer, and real.

**2. How to use your program**

When you first press the executable, it asks for the file name. Enter the full filename, including extension. Once you have entered it in, it will either tell you the results of our lexer or that it’s an invalid filename that can’t be opened for reading. The results are recorded into “results.txt”. The compiler must be run in “Release” mode instead of “Debug” mode if compiling in Visual Studio.

**3. Design of your program**

We used a struct to store the Token, so we can group the Lexeme, Token Names, and Token Type (number classification). We also used a global double array as a means for our Finite State Machine, which had 8 states and three inputs: letter, digit, and dot(‘.’). We used a global enum to classify the states. We also have global boolean variable “comments” for ignoring comments.

We created a *“getColumn”* function to help us determine the char input. Based on an integer and string input, We have a “getTokenName” function to help us classify the Token. We have a “idIntRealHelper” function calls *getTokenName* and fills out the rest of the Token information.

Based on a string input, we created the *“isSeparator”*, *“isOperator”*, *“isKeyword”*, “*isDoubleOp”*(for double letter separators and operators), and isPunct (ispunct function - the dot(.)) functions that return true or false if it is. We created a *“sepOpHelper”* function to return a Token if it is a separator or operator. This function utilizes the “is” functions above.

Finally, we created two function. One function (Lexer) that returns one token and another (Parser) that returns a vector of Tokens based on buffer input. In the Parser function, we created a vector of Tokens called “tokenHolder” to hold all the variables. In both functions, we created a string called “*currentHolder*” to hold the current input, and the “*currentState*” of the Finite State Machine.

In the Parser function, we used a for loop based on the buffer size to essentially call the Lexer function as it passes the *buffer*, *currentState*, *currentHolder*, and a reference to index. There are some error checking for comments and invalid input like empty space. Then it returns a vector of Tokens.

In the Lexer function, we have a reference input for the index to help us update the index in the Parser function. We make a while loop based on buffer length. We first check for empty spaces. If there is , we increment and maybe return a Token if state is greater than initial state(1). We use a while loop to check for [Punctuations - the dot](isPunct). We then check for the comments that occur in the middle of our buffer because “[\*” are punctuations. If it is a comment, we return a Token with type -2, which will not be push backed into the vector in the Parser function.

If it’s not, we check if current state is not equal to the initial state. If it isn’t, we decrement and return the Token based on the *currentState* and *currentHolder*. This is for stuff like (punctuation, letter, punctuation). Then we add buffer[index] into currentHolder. We check if it’s a double punctuation. If it is, we check for double Operators and Separators. If it is, we use our helper functions to return a double operator/separators and increment. If it isn’t, we use our helpers to return a single punctuation Token. If there aren’t two consecutive punctuation, we, for sure, return a single character punctuation Token.

If the whole thing is not a punctuation or space, we use our getColumn function with our buffer[index]. It specifies the input value into the FSM. We use that to determine currentState, add our buffer[index] to the currentHolder, and increment the index. We then loop back to the start of the loop. If it sees a punctuation or a space, it will return the Token based on the current State. Then once we get all the Tokens in the (Parser?) function, we return all the Tokens in a vector called tokenHolder.

In the main function, we ask the user to input their file. Then, we get the input line by line, enter it into our Parser function. Then we output it into the “results.txt”.

**4. Any Limitation**

We ran and created the executable in Release Mode, instead of Debug mode. (Visual Studios 2015/2017)

**5. Any shortcomings**

*NONE.*