**COVER PAGE**

# **CS323 Programming Assignments**

**Fill out all entries 1 - 7. If not, there will be deductions!**

**Check one**

1. Names [ 1. John Tu ], (MW [ X ] or R class [ ] )

[ 2. ], (MW [ ] or R class [ ] )

[ **if 3**. ], (MW [ ] or R class [ ] )

2. Assignment Number [ ]

3. Due Dates **Softcopy**  [ 2/18 ], **Hardcopy** [ 3/5 ]

4. Turn-In Dates **Softcopy** [ 2/18 ], **Hardcopy** [ 3/5 ]

5. Executable File Name [ Lexer.exe ]

(**A file that can be executed without compilation by the instructor**)

6. Lab Room [ CS 104 ]

**(Execute your program in a lab in the CS building before submission)**

7. Operating System [ Windows 10 ]

**To be filled out by the Instructor:**

GRADE:

COMMENTS:

Documentation #1

By John Tu CPSC 323

1: In this program, a lexical analyzer will act as a Finite State Machine (FSM), which reads in the sample source code and deconstructs each part into tokens. These tokens are then scanned as lexemes and categorized by their type. Once the analysis of the tokens is executed successfully, the results will be printed out consisting of lexemes and the type of tokens. The categories that will be used are identifiers, keywords, numbers, operators, and seperators.

2: How will the FSM lexer be implemented in C++ code? In order to answer that question, there will be 3 files: Reader.h, Reader.cpp, and TestFile.cpp. The header file of Reader contains the basic declarations of all public functions and private members for the lexicon analysis. The source file of Reader has all the functions declared and this is where most of the work takes place. TestFile.cpp serves only to test out the main program. First, the text input will be read in as the input, then the line in the text file will be scanned and deconstructed into individual tokens. While there is still some text to be read, continue to update the state, line number, and index while simultaneously sorting out the characters by the lexeme type and. Once that is done, create another text file as the output, and store the newly written information inside the file, and display the results on the console window.

3: What are the design features that make up the bulk of the source code? There are two void functions that have significance in playing major roles in building the lexicon: addToLex and updateState. In the addToLex function, the character is being appended to the array of tokens in the corresponding index location. This technique is related to the StringBuilder algorithm in Java and C#. What about the updateState function? This one will serve as the FSM function, and the Boolean functions that checks the type of character act as the state. If any of the conditions are true, then the reader character is accepted into the lexicon, and the state is updated. What if none of the Boolean functions and the switch conditions return true for any character? Then I assume that it should still be accepted, but rather than stating where does the token belongs to, it is read into the FSM as an invalid token. Therefore, the updateState function consist of a switch statement that checks the status of the current state and inside of it the if-else conditions of whether the token character conforms to the statement.

4: If there were limitations included in working with the finite state machine, then it will be the dynamically allocated memory in the array variables. For the dynamic variables token, tokenType, and tokenLineNumber, I can assume that there may be large numbers of data to process ranging from a thousand to almost a million. In order to prevent excessive memory from being consumed with each execution of the program, I set the maximum number of array space for memory allocation to be a million as a safeguard procedure. In addition, I shortened the void function updateState in order to simplify the syntax structure, and I do that by using a switch statement in place of the if-else conditions.

5: Are there any shortcomings that occurred when compiling the FSM code in C++? In other words, perhaps, if any, there are some explanations about any parts that I was supposed to implement into the lexical analysis, but simply unable to do so due to time constraints or the limitations of the program. It would be implementing another source code file called Analyzer, which was supposed to scan the token and print out the lexeme type. I think that due to the massive amount of data to read and the complex tasks of matching the categories, I didn’t bother to implement additional coding into the program.