**Names:** Fahad Alsowaylim

Giuliana Pham

Fulya Kocaman

**Assignment Number:** #1 Lexical Analyzer

**Due Date:** July 12, 2020

**Turn-In Dates:** July 12, 2020

**Executable FileName:** Lexical Analyzer

**Operating System:**  Windows/Linux

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

**GRADE:**

**COMMENTS:**

**CS323 Documentation**

**Problem Statement**

The problem we are trying to solve is to parse source code into meaningful units of interpretation. We want to be able to process the inputted source code and output a list of recognized versus unrecognized tokens. The recognized tokens should be identified by their type. These token types include identifiers, integers, separators, operators, and keywords. The unrecognized tokens will be indicated as unknown. The output will print the token type and the associated lexeme that has been identified as such.

**How to Use the Program**

To execute the program:

Windows

1. RUN Lexical Analyzer.exe
2. Enter the .txt file name, or directory\name.txt if it’s not in the same directory

Linux

1. Within command prompt, open ‘lexer’ directory.
2. Compile it with the command “**g++ lexical.cpp -o lexical**”.
3. Run it with “**./lexical**”
4. When the program runs it will prompt you to enter the source file name(test.txt)
   1. Make sure you list the directory for the test file, or just place it in the same directory as lexical.cpp

The **output.txt** file will be created the first time you run the program, but overwritten every time after that. (change the file name to keep it)

**Design of the Program**

Algorithm Description:

In the analyze while loop:

- If previous character is ‘ \* ’ and next character is ‘ ] ’:

- comment = false

- else if previous character is ‘ [ ’ and next character is ‘ \* ’:

- comment = true

- else if !comment

- else if next 2 characters construct a "$$" separator then:

- Add the previous token (if any) and

- add this separator to our token list.

- else if is any other separator:

- Add the previous token (if any) and

- add this separator to our token list.

- else if any operator:

- Add the previous token (if any) and

- add this operator to our token list.

- else if is a space character: add the previous token

- Add the previous token (if any)

- else:

- add the character to the current lexeme and

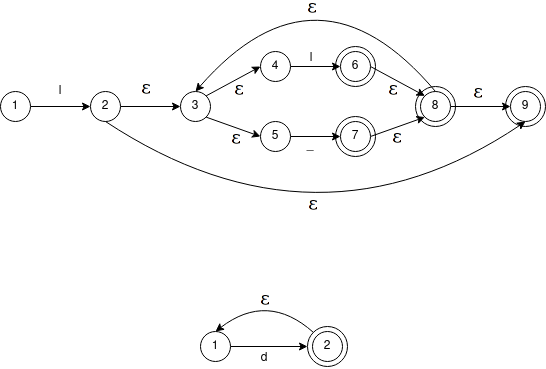
- call the FSM for with the current state and the character to go to next

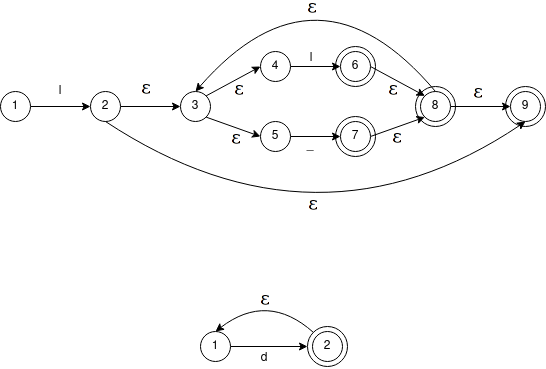
Data Structures Utilized:

* We defined a ‘token’ data structure which holds the token type and the associated lexeme.
* We used a vector of tokens to build the list which is outputted at the end of the program.
* We used arrays for the sets of keywords, operators, and separators.
* We used an enumeration for the defined states of the FSM.

The program is designed with the following specifications for its tokens:

* Identifiers:
  + Regular Expression: [a-z, A-Z]([a-z, A-Z]|\_)\*
  + NFSM:



* Keywords:
  + { integer, boolean, if, fi, otherwise, put, get, while, true, false }
* Integers:
  + Regular Expression: [0-9]+
  + NFSM: 
* Operators:
  + { |, <, >, /, \*, -, = }
* Separators:
  + { $$, , {, }, ;, , (, ) }

**Limitations**

* Integers are 10 characters maximum.
* Identifiers are 15 characters maximum.
* Whitespace between tokens is considered as a separator.

**Shortcomings**

None.