CS323 Documentation

Assignment 1 - Lexical Analyzer

Group Members:

Rosa Cho

Preet Desai

Cuong Pham

Christin Takeyama

Kien Truong

1. **Problem Statement**

   We were assigned to create a Lexical Analyzer using C++ as our programming language. As per instruction, we had to ensure that the Analyzer will separate inputted text from a file into Identifiers, Operators, Separators, and Keywords then return them as tokens. In doing so, we construct an FSM utilizing a state table to convert and organize our tokens into their respective states.

2. **How to use your program**
   1. Provide text file as input.
   2. LexicalAnalyzer.cpp will read the file line-by-line, separate it into tokens.
   3. Have each token converted to a certain state via state table (aka an array).
   4. Repeat with each line until the end of the file.
   5. Each line will then be printed onto a separate output file thanks to the main.cpp.
   6. End program

- Steps to execute the program on the terminal running Ubuntu 20.04 or Tuffix
  - *$ ./run.sh*
  - *$ ./proj1.out input1.txt* (Format ./proj1.out <input file>)
  - *$ cat output.txt*
    - Notice: The output will be inside output.txt file

3. **Design of your program**

   A majority of the data structures used to create our FSM is held within the LexicalAnalyzer.cpp file. The first three functions following the constructor and destructor are booleans which, along with their helper functions FindChar and FindString, determine whether the first inputted value at hand is a separator, operator, or a keyword. The function ConvertCharToCol is what follows up to take the value at hand and then determine what to categorize it as through a series of if/else statements. The if/else statements can be divided into three different ways: the ones that determine if they are letters/operators/separators i.e. the ones that require their own functions; the ones that determine if they are certain characters (!, _, a white space, . (for floats), or %); and finally the ones that determine that a value is its own special case. A state table (as represented as a matrix) is then used to act as the vehicle for our FSM. Finally, the data is printed onto an outputted text file. The file should look like this:

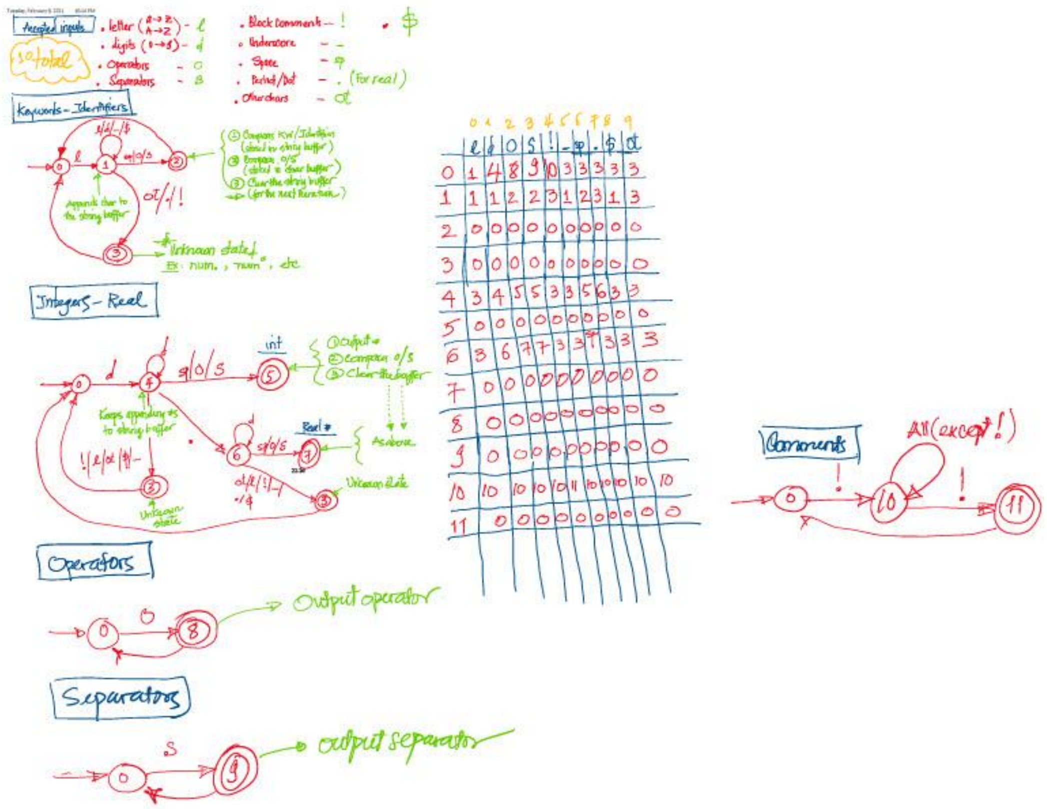
```
TOKENS                                   LEXEMES
```

```
KEYWORD     =           while
SEPARATOR   =           (
IDENTIFIER  =           fahr
OPERATOR    =           <
IDENTIFIER  =           upper
SEPARATOR   =           )
IDENTIFIER  =           a
OPERATOR    =           =
REAL        =           23.00
KEYWORD     =           whileend
KEYWORD     =           int
IDENTIFIER  =           num1
SEPARATOR   =           ,
IDENTIFIER  =           num2$
SEPARATOR   =           ,
IDENTIFIER  =           large_num
SEPARATOR   =           ;
KEYWORD     =           if
SEPARATOR   =           (
IDENTIFIER  =           num1
OPERATOR    =           >
IDENTIFIER  =           num2$
SEPARATOR   =           )
SEPARATOR   =           {
IDENTIFIER  =           large_num
OPERATOR    =           =
IDENTIFIER  =           num1
SEPARATOR   =           ;
SEPARATOR   =           }
KEYWORD     =           else
SEPARATOR   =           {
IDENTIFIER  =           large_num
OPERATOR    =           =
IDENTIFIER  =           num2$
SEPARATOR   =           ;
SEPARATOR   =           }
```

As you can see, the data is organized into three columns: token type, =, the value. The main.cpp ultimately writes the output file that will be printed.

FSM - Diagram (Credit: Kien Truong)

The FSM diagram below displays how different inputs are tokenized.



	0	1	2	3	4	5	6	7	8	9
	l	d	o	s	!	_	φ	.	$	α
0	1	4	8	9	10	3	3	3	3	3
1	1	1	2	2	3	1	2	3	1	3
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	3	4	5	5	3	3	5	6	3	3
5	0	0	0	0	0	0	0	0	0	0
6	3	6	7	7	3	3	7	3	3	3
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	11	10	10	10	10	10
11	0	0	0	0	0	0	0	0	0	0

New Section 1 Page 1

4. **Any Limitation**

None

5. **Any shortcomings**
   None, we essentially accomplished what the instructions wanted.