CS323 Documentation
Assignment #1 - Lexer

1. Problem Statement

In this assignment, we are tasked with creating a lexical analyzer (a.k.a. lexer), which is the first component of any compiler. The lexer must be created using a Finite-State Machine. The lexer will have a method called lex() which reads a line of input and returns a list of valid parsed tokens.

2. How to use your program

In order to use the program, simply ensure that the test case files ("testCase1.txt", testCase2.txt" and "testCase3.txt" are in the same folder relative to the executable ("program.exe"). Run the executable in a command-line prompt and it will automatically run the three test cases, pausing at the end.

3. Design of your program

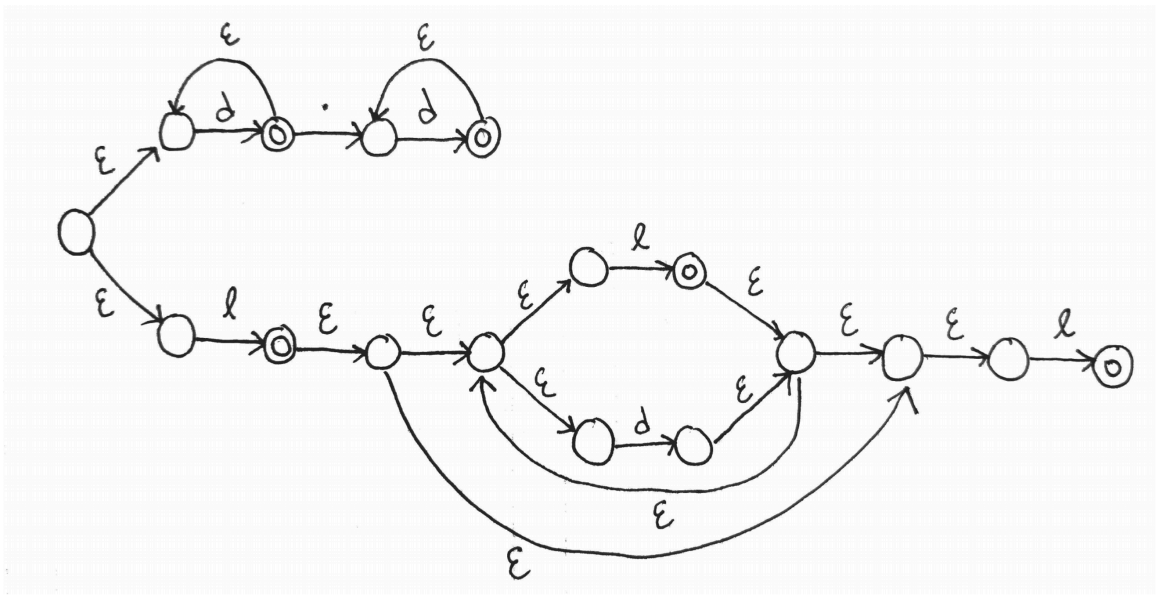
The first step to properly designing our Lexer was creating **regular expressions** for Identifiers, Reals and Integers:

Identifiers: **[l(l|d)*]**

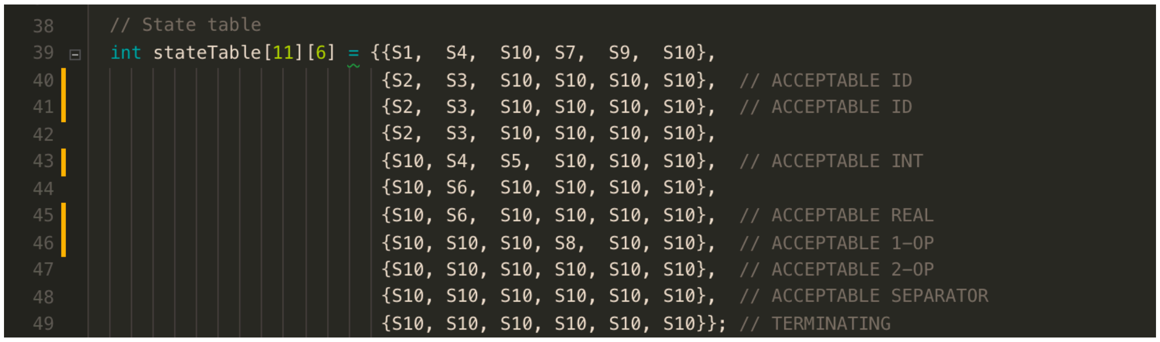
Real: **(d+.d+)**

Integers: **(d+)**

Next, we constructed a Non-Finite State Machine from the three regular expressions using Thompson's method:

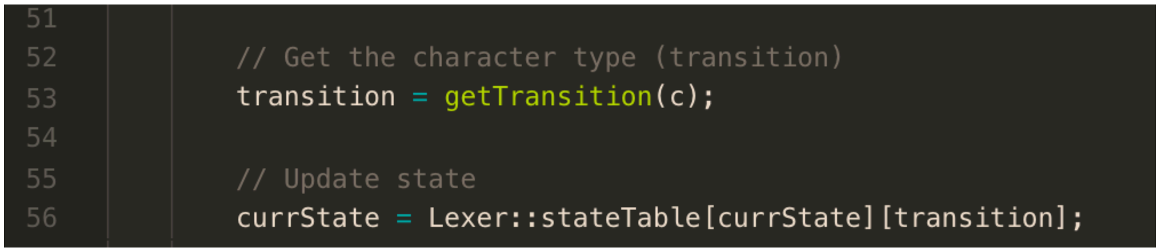


From this NFSM, we were able to construct a more detailed DFSM to use in our Lexer (X-axis is transition type, Y-axis is state):



```
// State table
int stateTable[11][6] = {{S1,  S4,  S10, S7,  S9,  S10},
                         {S2,  S3,  S10, S10, S10, S10},  // ACCEPTABLE ID
                         {S2,  S3,  S10, S10, S10, S10},  // ACCEPTABLE ID
                         {S2,  S3,  S10, S10, S10, S10},
                         {S10, S4,  S5,  S10, S10, S10},  // ACCEPTABLE INT
                         {S10, S6,  S10, S10, S10, S10},
                         {S10, S6,  S10, S10, S10, S10},  // ACCEPTABLE REAL
                         {S10, S10, S10, S8,  S10, S10},  // ACCEPTABLE 1-OP
                         {S10, S10, S10, S10, S10, S10},  // ACCEPTABLE 2-OP
                         {S10, S10, S10, S10, S10, S10},  // ACCEPTABLE SEPARATOR
                         {S10, S10, S10, S10, S10, S10}}; // TERMINATING
```

The design of the lex() method is simple. We start by reading every single character of the line of text passed into the method. We determine which type of transition the character is (integer, real, identifier, separator or operator), and we retrieve a new state from the table using this information in conjunction with the current state:



```

        // Get the character type (transition)
        transition = getTransition(c);

        // Update state
        currState = Lexer::stateTable[currState][transition];
```

After this, we check if the current state is equal to the terminating state. If so, we check if the previous state was an accepting state. If this is also true, we know we have found a valid token and we save it to the list of tokens. Otherwise, we clear the lexeme. If we haven't arrived at the terminating state yet, then we just add the current character to the lexeme and iterate.

4. Any Limitation

There are no currently known limitations to our Lexer. However, in the case of the single double-separator "$$", this token is handled in an ad-hoc manner. This could potentially cause some problems down the road if we are reading invalid tokens such as "$$$$", since we are only checking if the current character is '$', and the next character is also '$'. A better way to handle the '$$' token would be to incorporate it into our current state table. This is an area that we plan on improving before continuing work on the compiler. It is currently the only token (besides comments) which is not handled through the state table.

5. **Any shortcomings**

   None