Lexical Analyzer Documentation

Requirement: the program executable and the source input file MUST be in the same directory for the program to run independently from an IDE.

1. Problem Statement

The first step of compilation requires the source code to be converted from human readable programming language to that which is understood by the compiler. Characters from the source program are to be converted into tokens by the lex analyzer which needs to analyze the characters one at a time and break them into sequence of characters called tokens which is defined as a sequence of characters that represent a logical entity.

This lexer program should take the source code file as the input and read characters from it one at a time until it reaches end of file marker. It also needs to be able to back up once it reaches a separator in order to compare a sequence of tokens with a keyword and tokenize it as a keyword. The program is also required to identify floating point numbers and tokenize them accordingly. Additionally, it is required to identify operators and separators and tokenize them respectively. The program needs to ignore comment block move on to the next meaningful entity to analyze.

1. How to use the program

When the program is executed first it will ask the user for the name of the file including the extension .txt followed by the return key. For example, for our purposes the input would be: “SampleInputFile1.txt”. Once executed the program displays the table of tokens in one column and the corresponding lexeme in the second column. The program will also generate a file named: “output.txt” containing the table in the same format previously described.

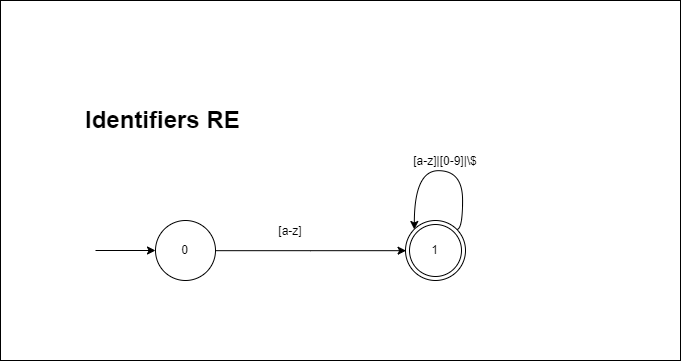
1. Program Design

The algorithm uses a 2 dimensional array of 6 by 7 for the implementation of the FSM using which the program navigates to different states given the input character from the source file utilizing a switch statement as the selection structure with the cases being the states and a function *characterTesting()* for identifying character types. The parameters for this function are input character and the current state. The function will then update and return the new *currentState*.

The states are referenced as follow: 0 = start , 1 = letter, 2 = digit, 3 = float, 4 = comment, 5 = invalid.

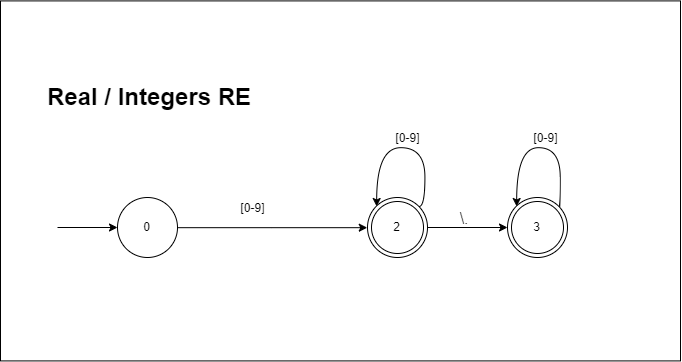
There is no number assigned to the final state but when selection structure reaches the end of an accepted valid state, we have a final sate where the program stores the token-lexeme pair and continues which is either moves back to the starting state or the true final state being the end of file/line for the FSA.

The Thompson’s algorithm for the NFA of how an identifier is tokenized using regular expression is as follow:



The NFA above as regular expression is: ([a-z]+([0-9]|[a-z])\*|\$\*)\*

Once the first alphabetic character is received the machine will go to state 1 and if a valid identifier is received as the input it will stay in that state until a any other non-qualified as identifier character is received.

The following NFA is the regular expression for real numbers using Thompson’s algorithm: 

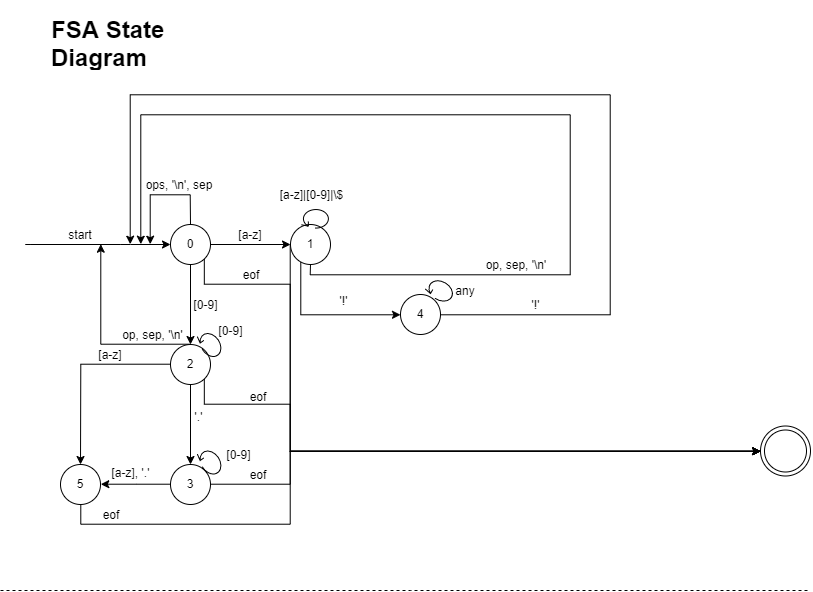
The regular expression for the above diagram is: [0-9]+\.\*[0-9]+

1 or multiple digits will set the machine in state 2 and if a floating point is received it will go into state 3 where it will stay until an invalid input is received which will take it to state 5 or a separator, operator or space is received in which case we have reached the final state for real number lexeme.

The algorithm is looped until the end of the file while processing one line of input at a time using a second loop ranged to the length of the one line of string from the source file.

The data structure used for storing the token and lexeme pair is a vector which stores them as a pair of strings and that is chosen because of the dynamic sizing and allowing the use of pairs. It is also indexable which allows for mapping the location of token types if necessary, for use in the future stages of compiler development.

Finally, below is the state diagram of the FSA used in this program:



At every state of the diagram the machine keeps going back to state 0 for the next input except for if there is eof in which case it will terminate. In the diagram above the acronyms used are: sep for separator, op for operator, and ‘!’ is begin and end of comment block as specified in the requirements and the any refers to anything but the specific input character.

1. Limitations

There are no limitations as far as the assignment requirements goes. However there are things to consider in the future for when there is invalid input going to state 5 and also for double operators such as += , ++, etc. Looking to handle those when developing the next stages of the compiler depending on the workload of the following assignments.

1. Shortcomings

None