Lexical Analzyer description and approach

This lexical analyzer, implemented in C++, employs Deterministic Finite Automatons alongside transition tables for their operations. The transition tables are represented using 2D arrays to facilitate the state transition process.

The analyzer processes the input file by reading it character-by-character. Upon encountering a character that may contribute to forming a token, the lexer employs conditional logic to select the appropriate DFA for that token category. This decision is pivotal as it directs the lexer to invoke a specific function designed for handling the DFA's state transitions. Each function leverages a 2D array (matrix) as a transition table to navigate through the states of the DFA as it processes the input string.

Upon successfully processing an input fragment, the function evaluates the token's validity and returns a boolean value to indicate its legitimacy. If the token is deemed valid, the lexer then classifies it according to its type and adds it to the collection of identified tokens.

How to Run the Program

To simplify the process for our users, we have provided a Makefile. You can easily compile and run the main program by executing:

```
make run
```

This command compiles the source code and creates an executable format of the main program. If you need to remove the executable and clean up your directory, you can use:

```
make clean
```

Alternative Compilation Method

If you prefer not to use our Makefile :(, or if you encounter any issues with it, you can compile the program manually. For Unix-based systems, use the following command to compile LexicalAnalyzer.cpp and create the executable:

```
g++ LexicalAnalyzer.cpp -o LexicalAnalyzer
```

Run it with

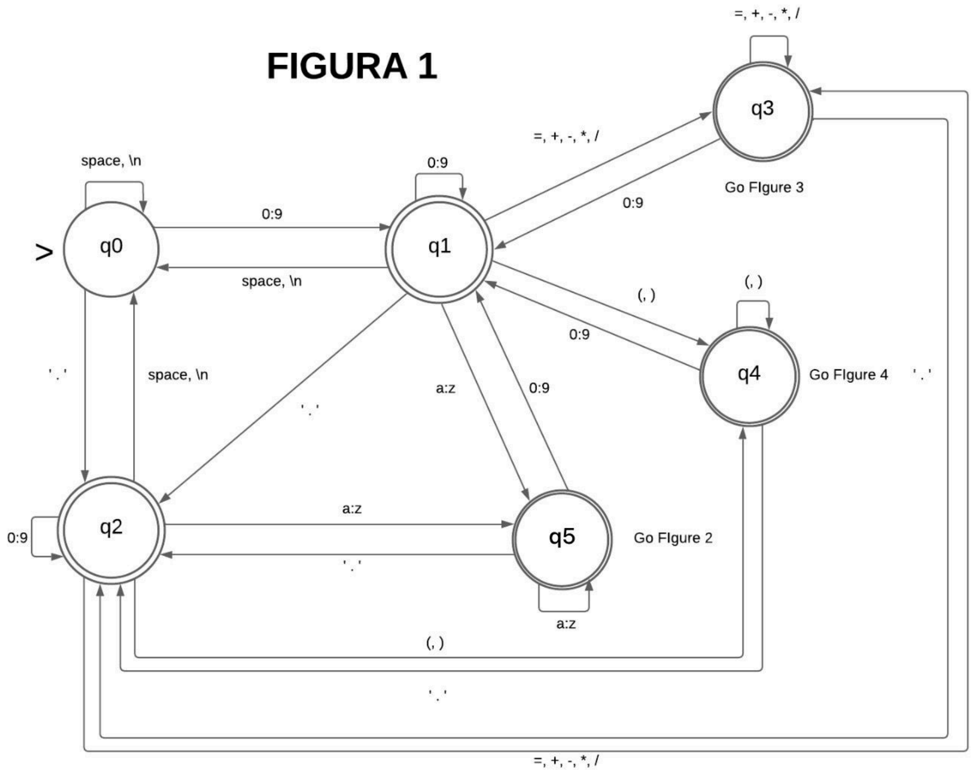
```
./LexicalAnalyzer
```

Customizing Input

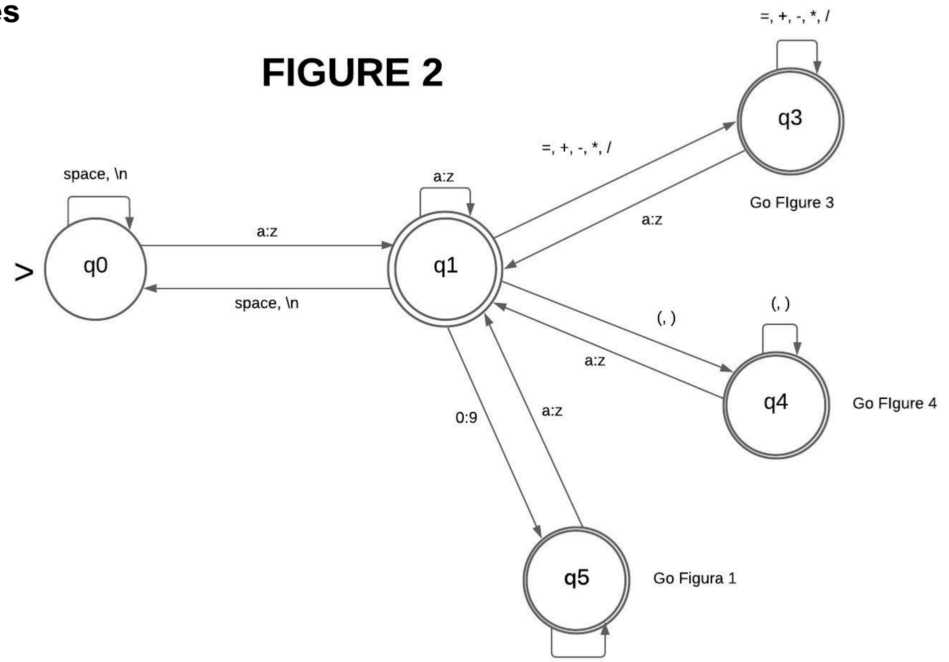
To test the lexical analyzer with different expressions, you can modify the contents of the `expressions.txt` file. This allows you to experiment with various inputs and observe how the program responds.

DFAs Section

Enteros y Decimales

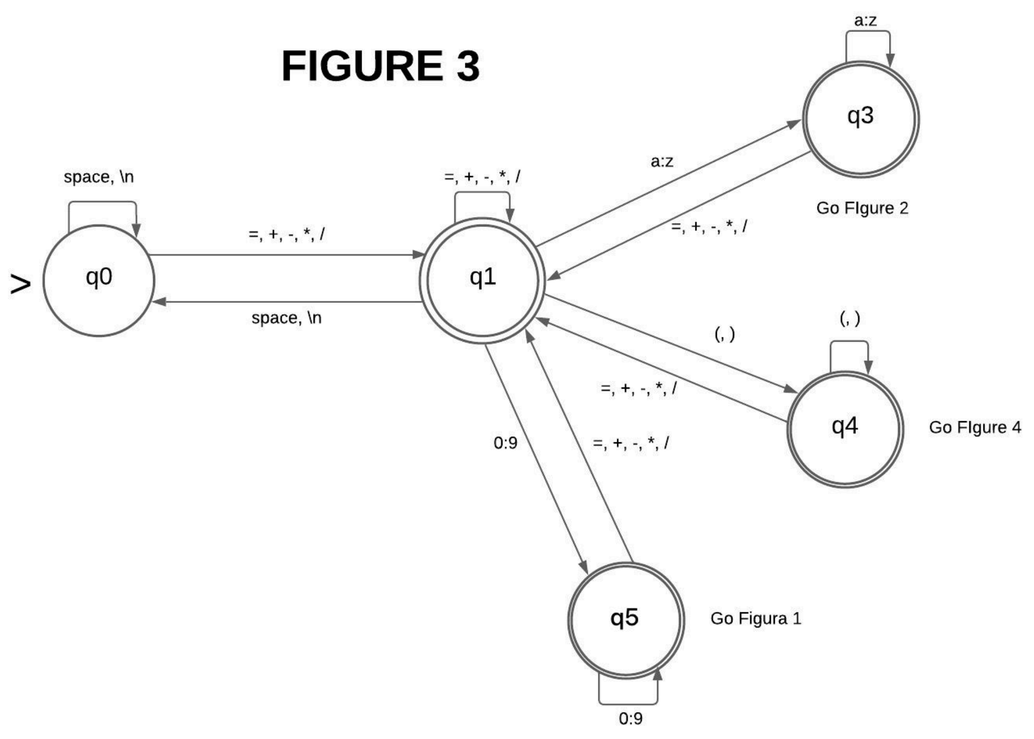


Variables



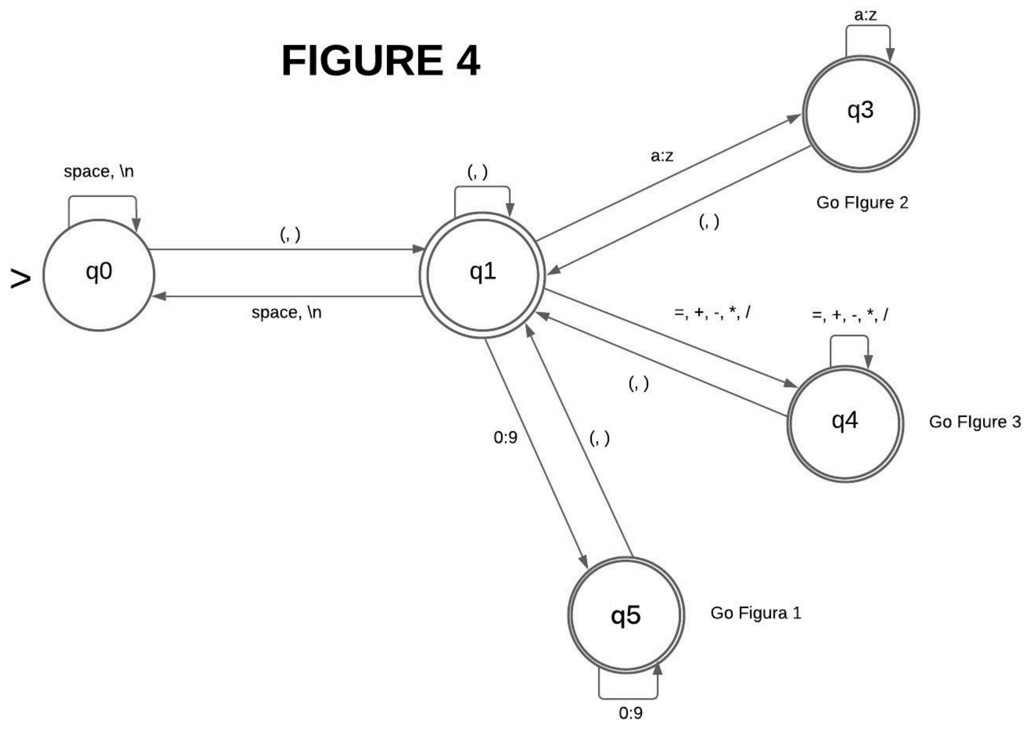
Operadores

FIGURE 3



Paréntesis

FIGURE 4



a:z
q3
a:z
space, \n
(,)
Go FIgure 2
(,)
(,)
q0
q1
space, \n
=, +, -, *, /
=, +, -, *, /
(,)
q4
Go FIgure 3
0:9
(,)
q5
Go Figura 1
0:9