Deterministic Finite Automata Report

Understanding

Thus far, through the content and materials in the slide with Deterministic Finite Automata (DFA) it would be beneficial to create a diagram before coding to both refer to it and grasp a better understanding of the task. This diagram, shown below, goes through the different options starting with either a Non-Zero number or Zero and listing the various options that branch of it leading to an invalid or valid expression (whether the DFA rejects the string or accepts its). These options had to take into account a SPACE, NUMBER, ZERO, DECIMAL and OPERATORS [+-/*].

$$M = (Q, \Sigma, \delta, q_0, F)$$

Where:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$$
$$\Sigma = \{\mathbb{N}\}$$
$$\delta: Q \times \Sigma \to Q$$
$$q_0 \in Q$$
$$F = \{q_1, q_2, q_3\} \subseteq Q$$

From both the lecture explanation and the specification, the valid expressions included either a zero number, a zero decimal number, a non-zero number and operators had to be between numbers. This lead to invalid expressions if a non-zero number had a decimal or an operator is not followed by a number etc.

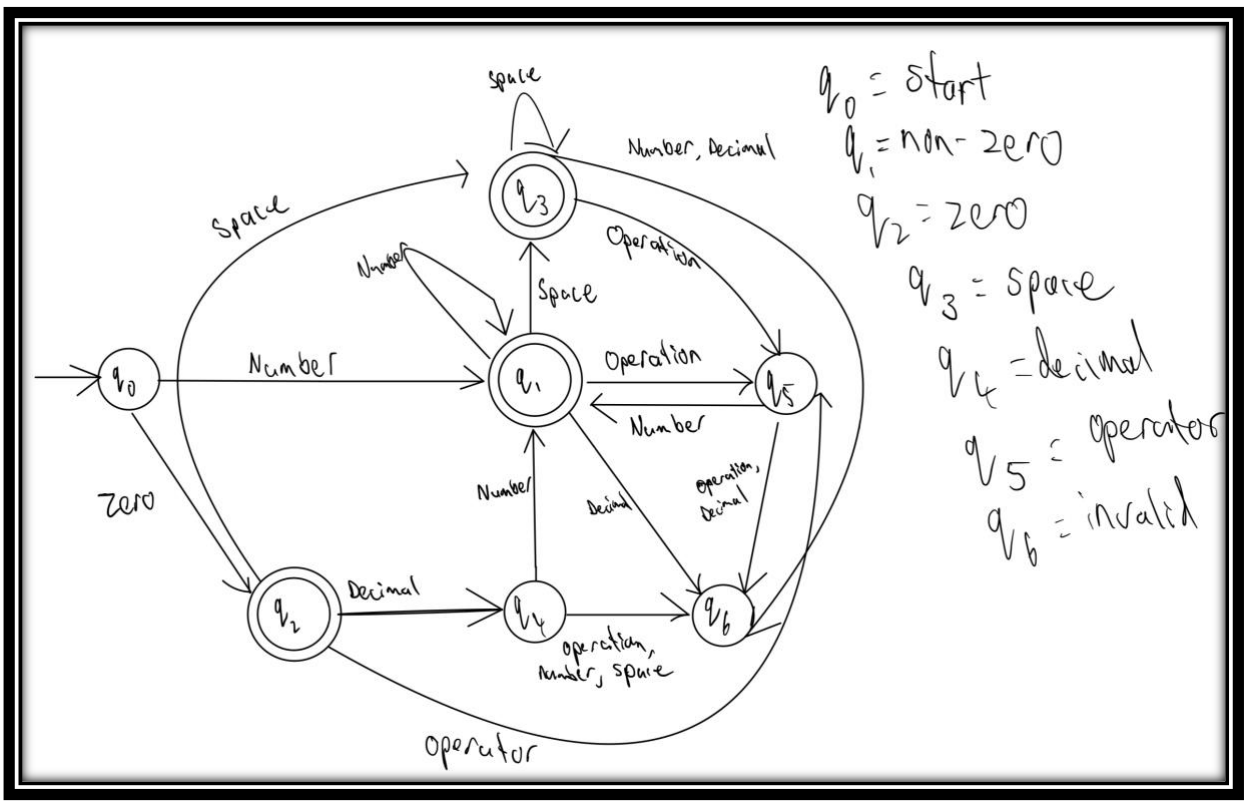

Figure 1 DFA For Assignment

Implementation

For implementation of the DFA into code, I used a String variable to contain the inputted string which would then be used to be added to a Token List. This is later used and checked against strings that contained the number 1-9 and the operators. The different states are outlined with START, ZERO, NUMBER, DECIMAL, SPACE, OPERATION to keep track of what the current state is with START being the first state the program is in.

```
private static final String number = "123456789";
// String zero = "0";
private static final String operations = "+-/*";
// String decimal = ".";

private enum State {START, ZERO, NUMBER, DECIMAL, SPACE, OPERATION};

public static ArrayList<Token> analyse(String input) throws NumberException, ExpressionException {
    String current = "";
    State state = State.START;
    ArrayList<Token> token = new ArrayList<Token>();
```

Figure 2 Declared variables

The code first goes through the input one character at a time and uses if statements for whether it is a number, a space, decimal, zero etc (note there are 5 if statements for 0, 1-9, space, decimal and operators). These contain a switch statement which dictates whether the following input was invalid or valid by going through each option available [shown on the left] with the option depending on the current state.

```
for (int i = 0; i < input.length(); i++) {
    if (input.charAt(i) == '0') {
        switch (state) {
            case START:
                current += input.charAt(i);
                state = State.ZERO;
                break;

            case ZERO:
                current += input.charAt(i);
                break;
```

Figure 3 for loop

```
if (operations.contains(String.valueOf(input.charAt(i)))) {
    switch (state) {
        case START:
            throw new ExpressionException();

        case ZERO:
            token.add(new Token(Double.valueOf(current)));
            current = "";
            token.add(new Token(new Token().typeOf(input.charAt(i))));
            state = State.OPERATION;
            break;

        case DECIMAL:
            throw new NumberException();

        case NUMBER:
            token.add(new Token(Double.valueOf(current)));
            current = "";
            token.add(new Token(new Token().typeOf(input.charAt(i))));
            state = State.OPERATION;
            break;

        case SPACE:
            token.add(new Token(Double.valueOf(current)));
            current = "";
            token.add(new Token(new Token().typeOf(input.charAt(i))));
            state = State.OPERATION;
            break;

        case OPERATION:
            throw new ExpressionException();
```

Figure 4 if statement

If it is valid then it adds it to the string and continues otherwise it throws and exception. The string is only added to a Token List when it is either before an operator or it reaches the end of the input. This ensures that the whole number is added, and it is not cut off early. This code is shown by token.add(new Token(Double.valueOf(current))); for each respective valid case. If the case is invalid given the current state it either throws an expression exception or a number exception.

States were used to keep track on what the last state it was before it changes, hence a START state is used to denote the start of the input and the other states to denote the change to a number or zero etc which is kept track of using the ‘state’ variable.

```
switch (state) {
    case START:
        break;

    case ZERO:
        token.add(new Token(Double.valueOf(current)));
        // token.add(new Token(new Token().typeOf(input.charAt(i))));
        break;

    case DECIMAL:
        throw new NumberException();

    case NUMBER:
        token.add(new Token(Double.valueOf(current)));
        break;

    case SPACE:
        token.add(new Token(Double.valueOf(current)));
        break;

    case OPERATION:
        throw new ExpressionException();
}
return token;
```

Figure 5 Final switch statement

There is also a switch statement outside the for loop to catch the last numbers of the expression. This is followed by the ArrayList token which is returned if it passes through all the switch cases.

Improvements

I took into account spaces to help the DFA run smoother if there were spaces, this includes throwing an error if there is a space between two numbers. Placing an enum with the state within the same class helped change the state easier and using my own ArrayList was due to my familiarity with it over the Collection library.

Challenges

Initially figuring out how different states to and how that would translate to the code was the hard part. Having an if statement to allow it to go through each option with a switch statement was the concrete solution to translate it from theory to the program.