Implementing a Functional Finite Automata In Programming

Tom Golding - 14316196
41081 Theory of Computer Science - Assignment 1 (Report)
Coding Language: Java

1 Introduction

Within this report, careful consideration has been placed to understand the concepts revolving around Finite Automata and their application within programs. A proper understanding of a lexical analyzer and its application will be discussed. Among detailed discussion on the formal specification of a finite automata. I will then further extrapolate on how I extracted my finite automata from the assignment specification, to form my own detailed specification on it, and further implement it into code. Concluding the report, with additional techniques I used, and challenges that I faced during the assignment.

2 Formal Definition of a Lexical Analyzer

A lexical analyzer is the implementation of a process that allows for the conversion of inputted characters, into a sequence of outputted lexical tokens. It is the first phase of a compiler that handles high-level languages, among these phases there consists, of the syntax analyzer and the semantic analyzer. On application, the lexical analyzer groups the inputted characters into lexemes (a sequence of characters in the source program), and further each lexeme is identified with a unique token. It also further allows for the ability to throw errors if a character is not identified with a token. Doing this allows for a lexical analyzer to be implemented within a codebase to allow an individual to successfully apply a functional deterministic finite automata (DFA). As the ability to read input and determine aspects such as the active state and dead state are crucial in a DFA's structure.

3 Formal Specification of a Finite Automata

A DFA consists of five components, commonly known as a five-tuple. These components include;

1) The alphabet Σ
2) A set of states Q
3) The start state q0 ∈ Q
4) A set of accepted states F ⊆ Q
5) A transition function : Q x Σ → Q

Which can be formally identified as DFA M = (Σ, Q, q0, F,) and a string w = w_1 … w_n ∈ $Σ^n$. Where M is the Machine and L(M) is the language the DFA recognizes, further continuing, M accepts w if there exists a sequence of states (r_0, …, r_n), r_i ∈ Q, such that:

1) $r_0 = q_0$, meaning begins with the start state.
2) (r_i, w_{i+1}) = r_{i+1}, meaning M reads a letter and changes its state
3) r_n ∈ F, meaning ends with an accept state.

So M rejects w if it does not accept w.

4 Extracting the Finite Automaton

To extract my version of a finite automaton from the specification, I started with identifying what input variations constitute a valid or invalid expression. Once properly identified, I was able to get a sufficient alphabet for my components, this alphabet consists of numbers 0 (zero) through 9, the decimal symbol "**.**", mathematical operators "+, -, *, /", and white space that I will identify as "_". Giving me Σ = {0-9, +, -, *, /, "_", "**.**"}as my alphabet. However, following specification, certain accepted elements are dependent on the first character being a 0 following a decimal such as, the number 0.1 is accepted but 1.1 isn't. So my numbers will be identified as either 0 or numbers from 1 to 9. As shown in figure 1, the alphabet has been labeled appropriately, apart from the mathematical operators which I will denote as "OPE" for short, and recursions labeled "ALL" consist of all elements in Σ. Following this, I then proceeded to determine what paths would lead to a dead state and an active state. By using the knowledge previously established, this step can be shown as I've mapped out the entire DFA in figure 1 below.

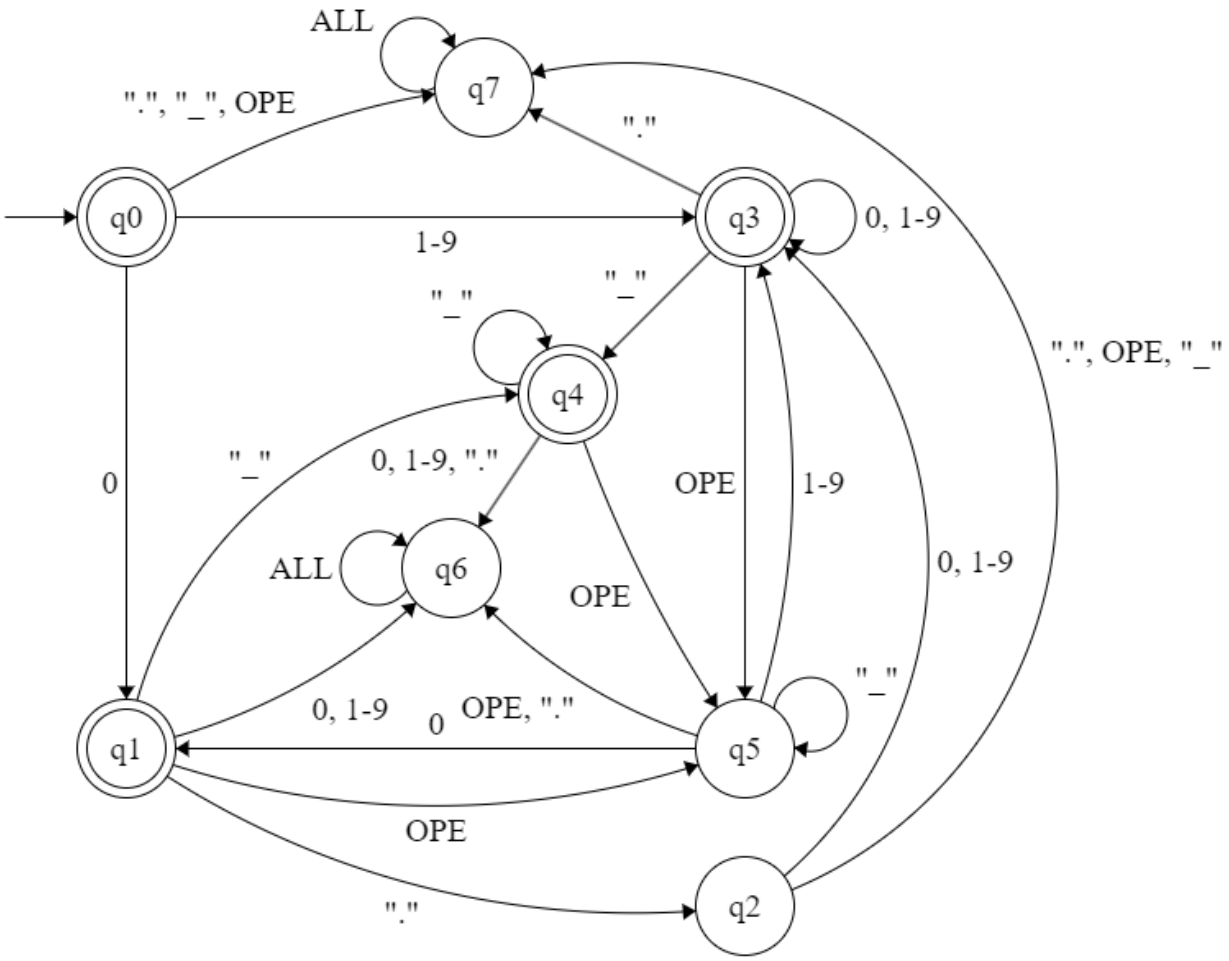

Figure 1.
Graphical representation of my constructed DFA

5 Formal Specification of my Extracted Automata

Given the information gathered from section 4, the formal specification for my extracted automata is as follows, for DFA M = (Σ, Q, q0, F,), where:

1) Σ = {0, 1-9, +, -, *, /, "_", "."}
2) Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7}
3) q_0 = q_0 ∈ Q
4) F = {q_0, q_1 q_3} ⊆ Q
5) = Q x Σ → Q

Below is my transition function table to easily understand the character transitions from state to state.

	0	1-9	OPE	“_”	“.”
q_0	q_1	q_3	q_7	q_7	q_7
q_1	q_6	q_6	q_5	q_4	q_2
q_2	q_3	q_3	q_7	q_7	q_7
q_3	q_3	q_3	q_5	q_4	q_7
q_4	q_6	q_6	q_5	q_4	q_6
q_5	q_1	q_3	q_6	q_3	q_6
q_6	q_6	q_6	q_6	q_6	q_6
q_7	q_7	q_7	q_7	q_7	q_7

6 Implementing Into Code

```
private enum State {
    EMPTYSTRING,  //q0
    ONEZERO,      //q1
    ONEDECIMAL,   //q2
    NUMBER,       //q3
    SPACE,        //q4
    ONEOPERATION, //q5
    EXPEXC        //q6
    NUMEXC,       //q7
};
```

Figure 2.
Code State Implementation

To implement the gathered information above, I had to understand thoroughly what I needed within my code. To do this, I studied and examined the example DFA supplied in great detail. During my examination of the example, I understood how the concepts of a finite automaton were transcribed to code. So the first implementation was to set the states as shown in figure 2. To keep my code clean and readable, I used descriptive words and commented to the side, their states as described in this report. I then preceded to initialize my “tokenList” which will store all tokenized lexemes, and initialize my start state, which can both be seen on lines 21 and 22. Furthermore, as per the specification, I understood that I needed memory allocation for my numbers that further get tokenized. So the variable “wholeNum” is initialized on line 23, which will store the appended digits of the current number. Continuing with examining the example DFA, I then turned to implement my transition functions. To implement this correctly, I considered what the process consisted of. Which is, to loop through each character of an inputted string, identify that character, change the current state to the corresponding state, tokenize, and throw exceptions when necessary. To do this, I started with an initial for loop which can be seen on line 28, which will then be used to increment through the input index to return the current character. Within this loop, I have five if statements that correspond to an element in my alphabet ({0, 1-9, +, -, *, /, “_”, “.”}). However with

operations, they are identified within one statement, by using a technique that I will explain shortly. Furthermore, these if statements have switch statements inside each one of them. This allows me to switch over the current state, and set the new state corresponding to their transition function, as shown for character 0 in figure 3.

```
if(character == '0') {
    wholeNum+= character;
    switch(state) {
    case EMPTYSTRING: state = State.ONEZERO; break; //q0 --> q1
    case ONEZERO: state = State.EXPEXC; break;      //q1 --> q6
    case ONEDECIMAL: state = State.NUMBER; break;   //q2 --> q3
    case NUMBER: state = State.NUMBER; break;       //q3 --> q3
    case SPACE: state = State.EXPEXC; break;        //q4 --> q6
    case ONEOPERATION: state = State.ONEZERO; break;//q5 --> q1
    case EXPEXC: state = State.EXPEXC; break;       //q6 --> q6
    case NUMEXC: state = State.NUMEXC; break;       //q7 --> q7


                }
```

Figure 3.
Code transition function implementation

Continuing through with my implementation, tokenization of my lexemes can be seen through two if statements on lines 60 and 106. Which parses the string “wholeNum” into a double, then passes that double and/or operator into a constructor to create the token, and then adds that token to the “tokenList”. This tokenizes the lexemes accordingly if the condition is met, and empties the string “wholeNum” if necessary. Thus, the final part of the implementation was exception handling, which can be seen on lines 104 and 105. These lines consist of two if statements which will throw a number exception if the final state is equal to NUMEXC (q_7) or ONEDECIMAL (q_2), or an expression exception if the final state is equal to EXPEXC (q_6) or ONEOPERATION (q_5). Due to the inability for the program to change the state to EXPEXC or NUMEXC if the input ended in either a decimal or operation. Adding these two states as dead states was necessary for the DFA to be fully functional.

7 Additional Techniques

Additional techniques were used within my implementation to minimize code and maximize efficiency, this additional technique was using regular expressions (Regexp). To incorporate regexp’s into my codebase, I had to import two packages java.util.regex.Pattern and java.util.regex.Matcher. Importing these packages allowed

me to initialize regexp's patterns for my numbers from 1-9, and my operators (+, -, *, /), as shown on lines 7 and 8. I then had to pass these patterns as parameters through the compiler function, which can be seen on lines 24 and 25. Doing this allowed me to pass these compiled patterns through as parameters into the matcher function, as seen on lines 30 and 31. Furthermore, I could use the variables associated with the previous lines as conditions for my numbers from 1-9 and operator (+, -, *, /) if statements, which can be seen on lines 47 and 59. Using regexp's in my code, allowed me to identify complex patterns and use these as conditions for the correct corresponding set of characters. Allowing me to reduce the code needed within my if statements, leading to a more detailed and readable end result.

8 Challenges & Limitations

During the assignment, I had several challenges that were not allowing me to pass certain test cases due to my poor implementation earlier on in the process. My first implementation consisted of global variables, public functions, and a single switch statement for my looped characters, which I soon found to be the root of my problems. During this phase of implementation, I was noticing strange and infuriating errors. These errors consisted of outputting tokens that exceeded the expected token output by tenfold, not throwing correct exceptions or not throwing at all, and not identifying what inputs were invalid or valid expressions. These errors were all due to my poor implementation of the specification, as I didn't correctly study and understand what steps I needed to take. Furthermore, the use of global variables and functions was the main cause for my token output to reach numbers of that degree. Another key limitation that lead to a slow start on the assignment, was that I hadn't coded in Java since enrolled in Applications Programming the previous semester. So a portion of my time spent when first approaching the assignment was allocated to researching and studying previous notes I had written about the language.

9 Acknowledgments

I had discussions with another student, Samuel Franks-Pearson, regarding the theoretical side of the assignment. Some coding approaches were discussed, however, different languages were used by both of us.

I used the website by Evan Wallace called https://madebyevan.com/fsm/, which is a Finite State Machine designer.