Sonny Uy Ly

41080 Theory of Computing Science

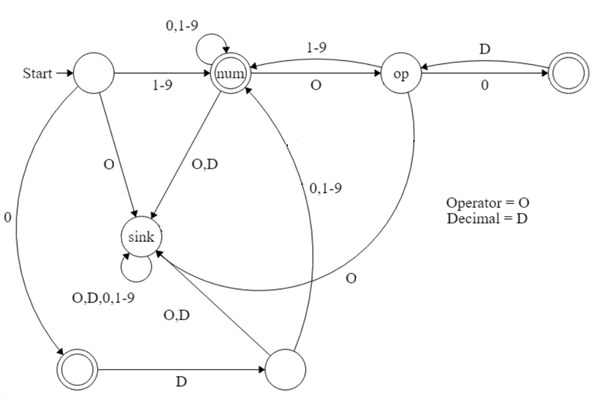
Assignment 1 – Finite Automata

HOW DID YOU EXTRACT FINITE AUTOMATA FROM THE SPECIFICATION? **/** WHAT IS THE FORMAL SPECIFICATION OF YOUR AUTOMATA?

Reducing the requirements of the assignment into smaller blocks of operations played a crucial role in completing the task. In the workshop exercises for this subject, we had to create finite state machines from a given formal language. For me to draw a finite automaton, I thought it would be easiest if I first created a formal language. I came up with the following language, *L* over the alphabet Σ = {‘0’, ’1’,’ 2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’+’, ’-‘, ’/’, ’\*’, ’.’}:

*L* = {x ∈ Σ\* | x is a valid number followed by an operator, followed by a valid number. A valid number is a string of digits possibly followed by a decimal point and a non-empty string of digits. If there is a decimal point, the string before the decimal point should consist only of 0. If the number starts with 0, it is either just 0, or 0.[something] where the [something] is a string of digits. A valid number is also a valid expression}

This formal language was based on the specifications given in the assignment. The next step in creating a finite state machine is to convert the formal language into a Deterministic Finite Automata (DFA).



There is a total of seven states with three accept states, for when a valid expression ends with numbers 1-9 or a 0 by itself. To help keep track of all the possible accepted strings and transitions of my DFA, I created a table of lexemes and token identifiers. A lexeme is a sequence of characters where the pattern is matched to a token. I have only three token identifiers in my finite state machine: Operators, Decimal and Numbers. The following table is an example of what I used to determine the state of a transition.

For the following expression: 0.11 + 3

|  |  |
| --- | --- |
| **Lexeme** | **Token category** |
| 0 | Number |
| . | Decimal |
| 1 | Number |
| 1 | Number |
| + | Operator |
| 3 | Number |

If the expression obliges to the specifications of the DFA and ends in an accept state, then the expression is a valid string. The process of using this table allows me to go through state by state and determine the outcome of the expression and adjust my code/DFA accordingly.

HOW DOES YOUR CODE IMPLEMENT THE FINITE AUTOMATA THAT YOU DESIGNED?

The first thing that came to mind when converting my finite automata to C++ code was to use a loop to increment until the last character of the input string and to use if statements to set the states. To generate tokens, the right restrictions had to be met. The if statements will correspond to the transitions of the DFA accordingly and will set the state (using Enums) and push back the token into the vector *analyse*. In my DFA, my code does not handle any whitespace. This is because whitespace is only important when it is in between 2 numbers/digits. If a digit is detected in the input string, a function will be called to obtain the full value of the number. The function will loop character by character the input string at the current index of the main loop until a digit or decimal is not present and return the substring of the string to be pushed back as a token.

If we have the expression: 2 2 + 3, the first two digits will be pushed back as two separate tokens. For the implementation of the sink state, I have a counter in each if statement that will be incremented according to the identified token. If the counter for any identifier goes above one, an exception to be thrown. Thus, for the above expression, an expression exception will be thrown because the number counter is more than one.

DID YOU USE ANY ADDITIONAL TECHNIQUES TO IMPROVE THE AUTOMATA?

During the early stages of the assignment, I was searching the internet for ways to complete the task. I came across some stackoverflow answers recommending the ‘shunting-yard algorithm’. Although I did not end up implementing the algorithm, studying its methods of parsing mathematical expressions from infix to Reverse Polis notation gave me new insights and ways to complete the task.

A technique I implemented was that I hard coded the sink states. This allows the program to throw an error as soon as an exception is found. This makes the code more efficient as it does not need to loop through the whole input to throw an exception.

DID YOU ENCOUNTER ANY CHALLENGES OR LIMITATIONS, EITHER TECHNICAL OR CONCEPTUAL, IN IMPLEMENTING A THEORETICAL CONSTRUCT IN A CONCRETE PROGRAMMING LANGUAGE?

There were many challenges throughout the assignment. Firstly, the limitation in only being able to go through the input string character by character. This eliminated the use of many inbuilt functions in the C++ library (regex, split, etc). I have always been told to ‘not reinvent the wheel’. However, programming a finite state machine manually was significantly beneficial to my learning.

Secondly, turning the DFA into code was a major challenge because the conditions of a sink state required a lot of thinking to produce. After each sink state condition and if statements were implemented, the readability of the code greatly reduced which made it difficult to progress. I found that documenting the code helped tremendously.

A challenge at the start of the assignment was that I was creating too many tokens, thus failing the testcases. To combat this problem, I implemented a function to get the substring of the entire number and concatenate the value into a buffer string. Also, deciphering how the skeleton code worked took me a while to understand due to my lack of knowledge of vector objects and how Enums work.

REFERENCES

Karl, 2016 *3.1. Tokens, Lexemes and the Lexer,* viewed 28 August 2020

<<http://www.cs.ecu.edu/karl/5220/spr16/Notes/Lexical/token.html>>

Shunting-yard algorithm, 2015 *Wikipedia*, viewed 28 August 2020 <[https://en.wikipedia.org/?title=Rush\_(band) (Accessed: 18 June 2015)](https://en.wikipedia.org/?title=Rush_(band)%20(Accessed:%2018%20June%202015).)>

Wallace, 2010 *Finite State Machine Designer*, viewed 1 September 2020

<<http://madebyevan.com/fsm/>>