Eric Einhaus

MATH 4471 | dr. vOGEL

**Representations of Class Material in Code**

MATH 4471: computer algorithms

**Contents**

* **Part 1: DFA and NFA representations using Python**
  + DFA example 1 code and test cases
  + DFA example 2 code and test cases
  + NFA example 1 code and test cases
  + NFA example 2 code and test cases
* **Part 2: Regular Expressions representation in JavaScript**
  + Regular Expression example using HTML, CSS, and JavaScript
* **Part 3: Representation of a PDA in Java**
  + PDA example 1 code and test cases
  + PDA example 2 code and test cases
* **Reflection of Project**

**Part 1: DFA and NFA representations using Python**

Syntax and any helpful notes considered:

* Anything following a ‘#’ is a comment within the code.
* Variables in Python do not have to be declared with a primitive type. With the DFA and NFA mimics, I used integer and boolean type variables.
* Anything with the keyword ‘def’ is a function, (test) is a parameter of that function containing a string.
* ‘nonlocal’ is a keyword in Python to reference a variable outside the scope of the function within.
* Substrings in Python are represented with bracket notation. For example, test[i:i+1] represents a substring of the string test, containing 1 character from the starting position i, to the ending position i+1, which is one character.
* Conditional statements are self-explanatory. Anything following a colon (:) on the next line of code is within the scope of the condition statement. == represents checking for equality.
* The len() function checks the length of the string in the parameters.
* The main method toward the bottom represents the testing environment, where to test different strings and see if it is an acceptable language. In Python, the syntax ‘if \_\_name\_\_ == "\_\_main\_\_":’ represents the main (testing suite) method. I created test strings and printed out the acceptable Boolean value in the Python shell as shown in the output in the following pages.

**DFA Example 1:**

A picture containing object, clock, watch

Description automatically generated

* L: {w | w contains the substring ‘00’ or ‘11’}
* States: represented as A, B, C, D (acceptable state) in diagram represented as q0, q1, q2, q3 respectfully in Python code below:

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

Output in Python Shell:

A screenshot of a cell phone

Description automatically generated

**DFA Example 2:**

**A picture containing wall

Description automatically generated**

* L: {w | w contains an even amount of ‘0’ and ‘1’ bits}
* States: represented as A (acceptable state), B, C, D in diagram represented as q0, q1, q2, q3 respectfully in Python code below:

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

Output in Python Shell:

A screenshot of a cell phone

Description automatically generated

**NFA Example 1:**

**A picture containing object

Description automatically generated**

* L: {w | w contains all ‘a’ bits (including 0) or alternating ‘ab’ segments.}
* States: represented as A, B (acceptable state), C, D, and E (acceptable state) in diagram represented as q0, q1, q2, q3 and q4 respectfully in Python code below:

A screenshot of a social media post

Description automatically generated

A close up of a piece of paper

Description automatically generated

**A screenshot of a social media post

Description automatically generated**

Output in Python Shell:

**A screenshot of a cell phone

Description automatically generated**

**NFA Example 2:**

**A close up of a map

Description automatically generated**

* L: {w | w contains an odd number of 0’s or a number of 1’s not a multiple of 3 (or both)}
* States: represented as A, B, C, D (acceptable state), E (acceptable state), and F (acceptable state), in diagram represented as q0, q1, q2, q3, q4 and q5 respectfully in Python code below:

A screenshot of a social media post

Description automatically generated

A close up of a map

Description automatically generated

A screenshot of a social media post

Description automatically generated

Output in Python Shell:

A screenshot of a cell phone

Description automatically generated

**Part 2: Regular Expression Representations in JavaScript**

Syntax and any helpful notes considered:

* Anything in between ‘<!—-‘ and *‘*-->’ or after ‘//’ are comments
* The first 55 lines of code below is the HTML and CSS that makes the webpage look ‘good’. While important to the user interface and getting user input, it will not be looked over in depth.
* Anything from line 59 to 97 is JavaScript code. This is where our regular expression is declared and tested.
* The ‘const’ keyword declares a constant variable that, hence the name, cannot be mutated in the code.
* There are syntax changes that needed to be done to translate the regular expressions used in the textbook to regular expressions used in JavaScript. More detail about this is in the code.
* In the screenshots below, there are examples of where it shows what an acceptable or a rejectable string would output on the page. These results are also outputted in the webpage’s console in the last screenshot.
* Any additional information regarding syntax or detail about what each line is doing in the code is shown below in the code’s comments.

**Regular Expression Example:**

**L1 = {Strings that contain an odd number of 0’s or exactly two 1’s}**

∑: {0, 1}

**RE (in textbook syntax)**

(1\*(1\*01\*01\*)\*01\*)+(0\*10\*10\*)

**RE (in JavaScript syntax)**

/^(1\*01\*(1\*01\*01\*)\*)$|^(0\*10\*10\*)$/

A screenshot of a social media post

Description automatically generated

Figure 1: HTML/CSS of webpage

A screenshot of a social media post

Description automatically generated

Figure 2: JavaScript of webpage

A screenshot of a cell phone

Description automatically generated

Figure 3: The webpage on startup

A screenshot of a cell phone

Description automatically generated

Figure 4: The webpage when user inputs acceptable text

A screenshot of a cell phone

Description automatically generated

Figure 5: The webpage when user inputs rejectable text

A screenshot of a cell phone

Description automatically generated

Figure 6: Additional test cases for our regular expression in Google Chrome’s console.

**Part 3: PDA representations using Java**

Syntax and any helpful notes considered:

* Anything following a ‘//’ is a comment within the code.
* Variables in Java have primitive types. In these examples, I used int, boolean, and String.
* Anything starting with ‘public void’ or ‘public String’ with a pair of ‘{}’ is a method. ‘void’ means no return value is required.
* A constructor is used to initialize the values of the global variables and the stack seen in detail below.
* Substrings in Java are used with the substring(start, end) method. ‘start’ is the position where to start the substring from the current, similarly for ‘end’.
* Conditional statements are easy to follow. Anything following a ‘{‘ represents the actions if that condition is true.
* length() checks the length of the string, isEmpty() checks to see if the string is empty, charAt(i) gets the character at position i.
* The stack methods used are push(), pop(), peek(), isEmpty(), and clear(). peek() gets the element at the top of the stack, isEmpty() checks for an empty stack, and clear() empties the stack.
* ‘!’ before a method is the negation of it.
* The main method toward the bottom represents the testing environment, where to test different strings and see if it is an acceptable language. In Java, the syntax ‘public static void main(String[] args)’ represents the main (testing suite) method. I created test strings and printed out ‘Acceptable!’ or ‘Rejectable’ in the Java Complier as shown in the output in the following pages.

**PDA Example 1:**

**A drawing of a person

Description automatically generated**

* L: {10n10n1 : n >= 0}
* States: represented as S, A, B, C, D, and E (acceptable state) and represented with the same names in the Java method names below.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

Output in Java Complier:

A black and silver text on a white background

Description automatically generated

**PDA Example 2:**

**A screenshot of a cell phone

Description automatically generated**

* L: {0n1m0m1n : n, m > 0}
* States: represented as S, A, B, C, D, and E (acceptable state) and represented with the same names in the Java method names below.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Output in Java Complier:

A black storefront with white letters

Description automatically generated

**Reflection on the Project**

The ultimate goal of this problem was to use the skills that I have in multiple programming languages and my new gained knowledge of different machines to represent multiple examples of such machines in code.

Firstly, I used Python to represent different examples of deterministic and nondeterministic finite automata (DFA and NFA). Using methods to represent states, I simply represented these machines by calling the method related to the state and returned at the end if there was a pointer Boolean value at the acceptable state.

Second, I looked into how regular expressions worked in JavaScript. With the help of HTML and CSS, I made a webpage asking for user input of an acceptable string recognized in a given language. The JavaScript regular expression syntax was slightly different than the once used in the ‘Theory of Computation’ textbook, but easily translated. In the end, the webpage would dynamically change the text on the screen depending on if the user input is acceptable or rejectable by the language.

Lastly, I used Java to represent two examples of push down automata (PDA). Everything was about the same when it came to the set up of this compared to the NFA/DFA representations in Python, other than two things: 1) The syntax of Java is different, and 2) The use and implementation of Java’s ‘Stack’ library. Using these utilities, I was able to implement two examples of PDA’s using the functions in the ‘Stack’ library.

For all three parts of the project, I wanted to make sure I provided sufficient test data to make sure my representations are fully functional and error free. The fact that I used enough test data made me realize at many times that bugs were in my code. I was able to track the source of these bugs easily because of this practice.

There were a few times where I found myself struggling. Firstly, when implementing the JavaScript of the webpage in Part 2, I spent quite a bit of time troubleshooting and researching JavaScript regular expressions. In the end, it was a syntactical error in the regular expression that was causing my code to not run properly. Specifically, I needed to add the symbols ‘^’ and ‘$’ at the beginning and end (respectively) of the regular expression in able for the program to recognize to check the regular expression for the entire test string. Another problem came in the form of ‘out of bounds’ errors in Part 3. These were caused by a substring being empty and the program checking for a character at position 0, which there was none. To fix this, I added a condition statement for most states, checking for a non-empty string. If the string was nonempty, it proceeded to the original conditional statement which was now nested inside it. If it was empty and not in an acceptable state, the pointer would disappear.

Overall, I found this project very beneficial to my knowledge of both machines and regular expressions. Additionally, it provided with a hands-on opportunity to write code on practical machines and test my knowledge on the syntax and resources of various coding languages. Additionally, I was able to discover roadblocks, and learn the solutions to them in case they would appears again in the future.