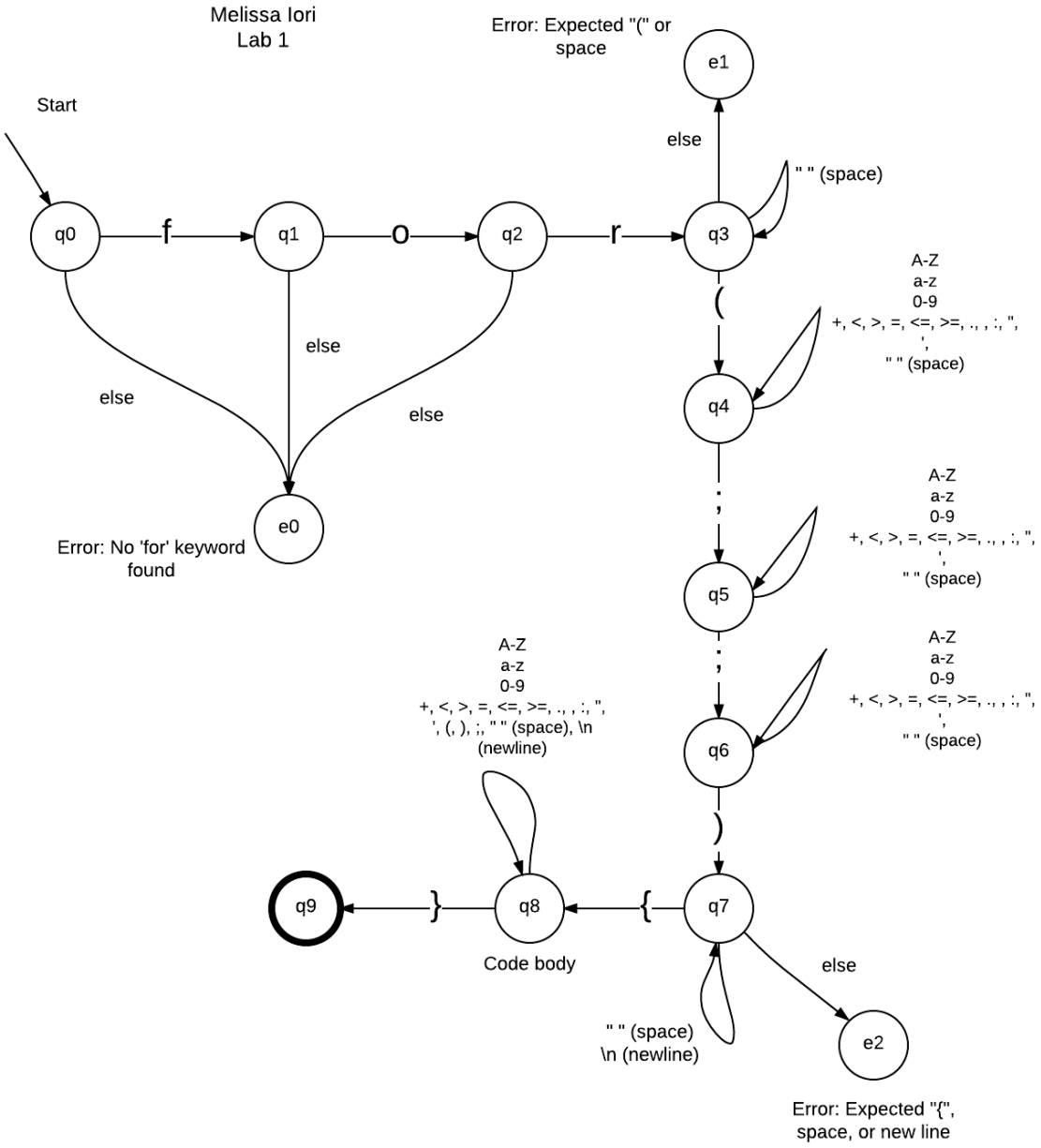
Melissa Iori
CMPT 440
Lab 1

1. The .png version is available on Github, https://github.com/1PhoenixM/formal-languages-and-automata .



Assuming: that Java code in the “for” conditions and in the code body was valid.

Chose a straightforward method that verifies the syntax of the for loop and is flexible enough to accept complex for loops with lots of code, as well as the empty for loop from the example.

2. In terms of problems, I found that some for loops might have a space between the "for" keyword and the opening parentheses "(", and some don't, but both ways are syntactically correct in Java. To get around this, I created an error state, e1, that allows the string to contain one or many spaces there, or to simply continue with the parentheses. However, if any other character is used, the string will go to the e1 state and produce an error. A similar issue exists with the closing parentheses, ")" and the beginning of the Java code body with "{" curly brace. For this I created a separate error state, e2, which is the result if other characters are used between the end of the conditions and the start of the body. I also allowed the newline character here, since that can be used as well, though it is the user's syntactic choice of how they want to write the for loop. The other error state, e0, is used if the "for" keyword is not complete or not found at the start. I also used self-loops to allow any number of identifiers, values and operators within the "for" conditions (the initialization, termination, and increment) and also the body (the statement).

3. In Java, an object-oriented language, the best way to represent a deterministic finite automaton would likely be to create a class called "DFA". That way, every time someone wanted to create a new DFA, they could use "myDFA = new DFA();". I would create a constructor in this class that would take in the five parameters of formal DFA definition, Q, sigma, delta, q, and F. So we can amend the former to myDFA = new DFA(Q, sigma, delta, q, F);", all of which would be required parameters when creating a new DFA instance. Also, a class called State would be created, that represents State instances, each one having its own name. Q is a set of all states, so it could be represented by an ArrayList of States, or ArrayList<State>. Sigma would be an ArrayList of characters, or type char, because it's the alphabet containing all valid symbols, or characters. Another class called Transition could be made to represent each transition possible, each stored as a State and a character. Delta, being the transition function that maps Transitions on to next States, could be a HashMap<Transition,State>. q, being the starting state, would be a State. And F, containing the accepting states, would be an ArrayList<State>. Or maybe, each State could have its own Boolean property, "isAcceptingState" for it to be included in F. The DFA class

could also have its own “accepts” method which returns “true” if the DFA accepts a String passed in as a parameter, and “false” if not. This method would start at the start state, or q, of the DFA, and loop through the passed in String one character at a time. Each Transition would be composed of the current state and the next character, and the new current state would be set based on the delta function mapping, simply looked up in the HashMap. Finally, at the end of the loop, the current (final) state would be tested to see it F contains it, or if isAcceptingState is true for that State. If it is, return true, else, return false. This method could be used by any DFA on any String, such as to validate a for-loop syntax, as “myDFA.accepts(“testString”)” to return a true/false answer of whether the string is accepted or not.