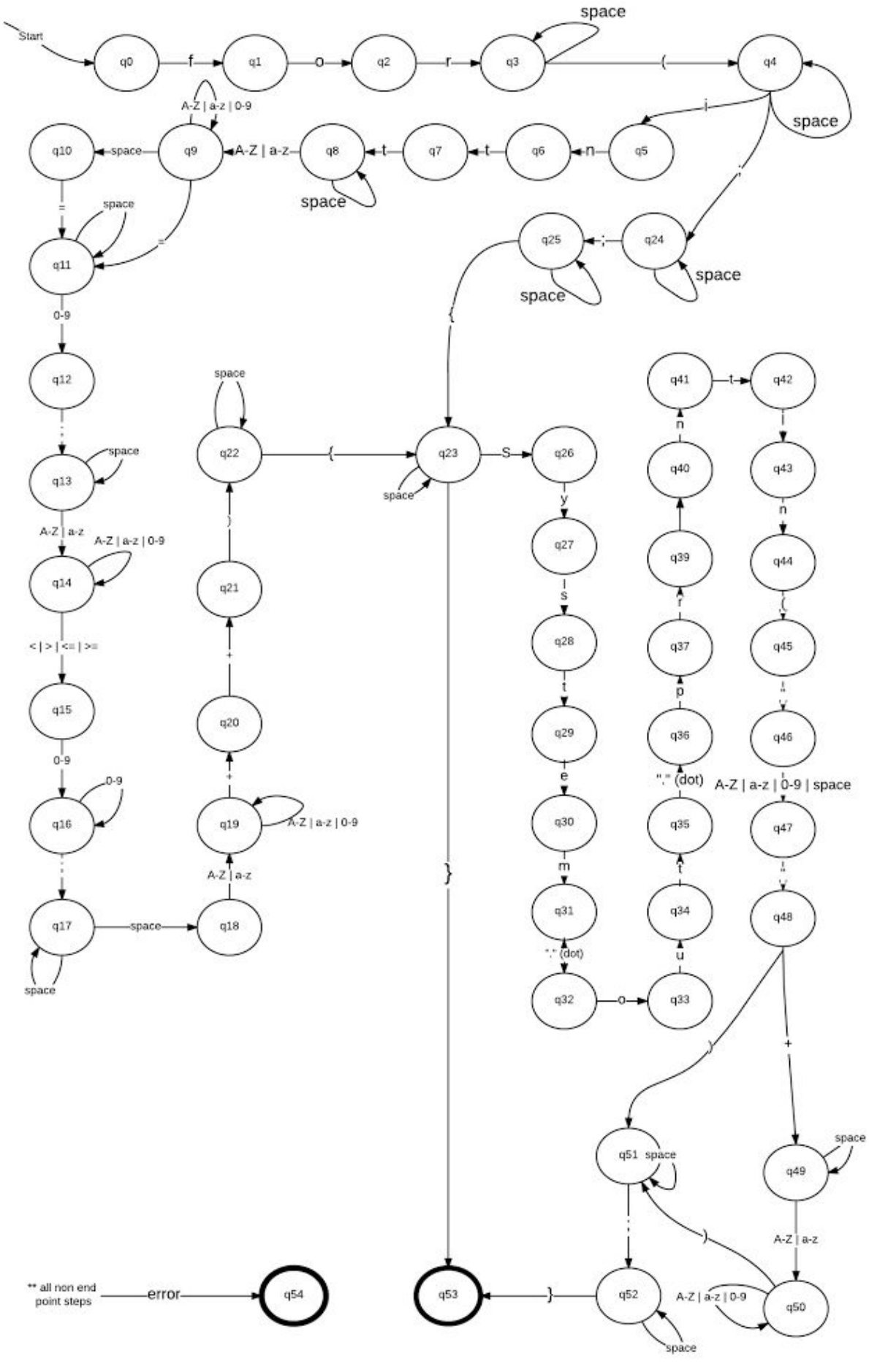
Evan Hopkins
Formal Languages & Computability
Pablo Rivas
2/1/2016



Evan Hopkins
Formal Languages & Computability
Pablo Rivas
2/1/2016

2)

The main problem that I encountered was how to handle variable names. The first issue that arose was how to diagram a name that varies in length and characters. I chose to represent the ranges of characters allowed, meaning A-Z (uppercase), a-z (lowercase), and 0-9 (numbers). I handled the varying length by having an arrow going into a state, which meaning at least one character is required. Then an arrow looping form that new state back into itself, meaning an infinite amount of additional characters can be added.

Another issue I ran into was how to deal with spaces. Sometimes a space is required, but other times it is optional. Where ever there was an optional space, I just created a loopback of ‘space’ indicating that 1 or more spaces could be added. Mandatory spaces move the diagram to the next state, meaning they cannot be skipped.

The last issue I ran into was diagramming errors. Theoretically, every state could have an error. This means each state should have an error arrow that points to a final (error) state. I didn’t want to overcomplicate and crowd the graph, so I created a separate one for errors. It states that any state could throw and error.

3)

I think the best way to represent the DFA itself would be with a modified version of a tree. Each ‘node’ should contain a state number, its children, and what set of characters allow you to enter the current state. When evaluating the string, you could iterate over the string while traversing the tree with each character. If you ever fail to find legal children while traversing the tree, the string is not valid. In practice, this would likely be a lot harder than I am picturing it. I am sure once I begin to implement, I will realize this.

One interesting aspect that needs to be handled is when anything in a set of characters is allowed. For example, once state may be entered by a comparison operator. There are a few of these (>, <, ==, !=, <=, >=) which could all be used interchangeably. One solution would to create a CharacterSet class which would define a set of legal characters. This class could have a method, evaluateCharacter which would return a boolean indicating whether or not the character fits the set. Breaking this functionality into its own class would be a good way of separating logic as much as possible.