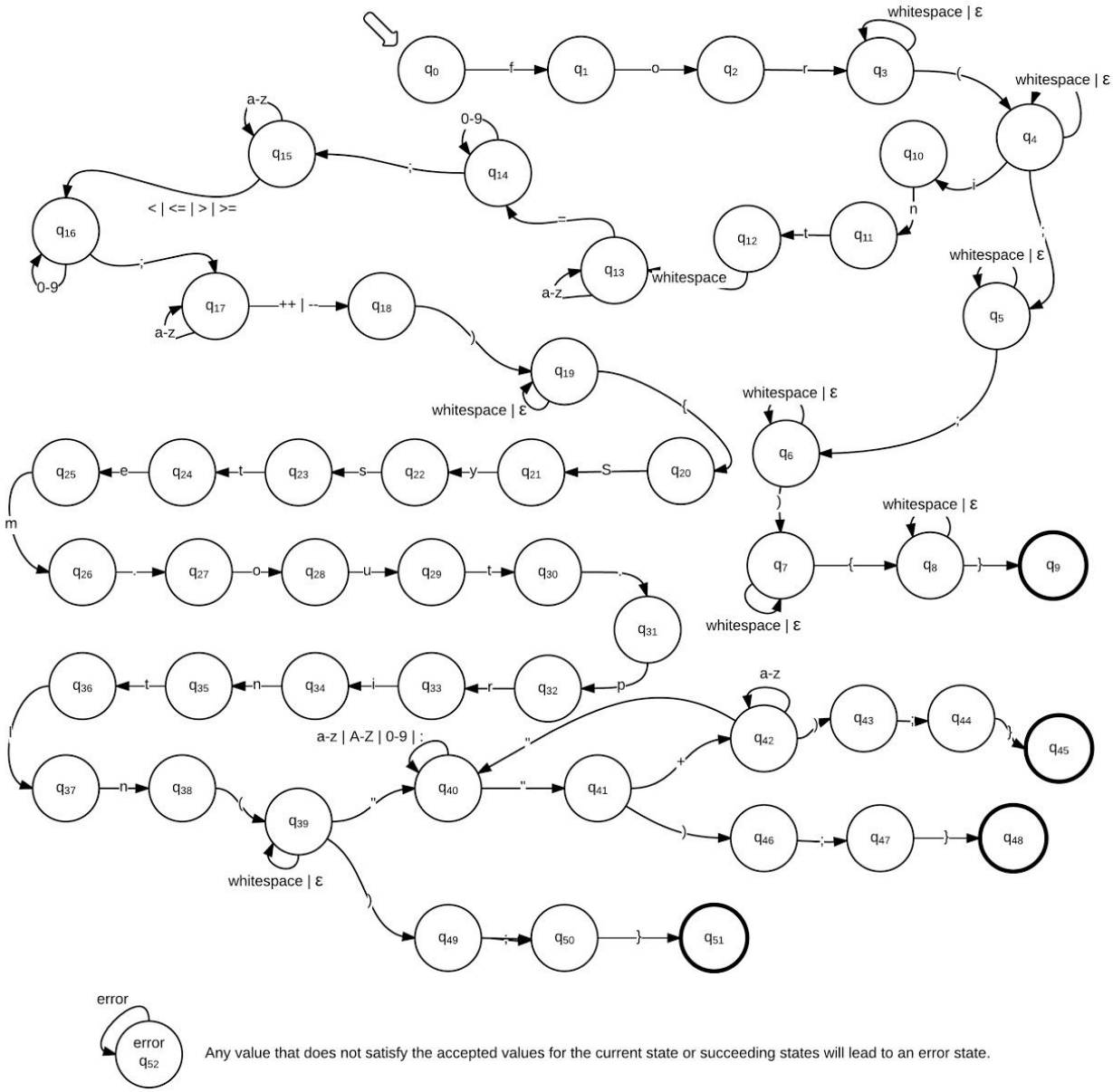
## DFA Diagram


q0
f
q1
o
q2
r
q3
whitespace | ε
(
q4
whitespace | ε
a-z
q15
0-9
q14
;
< | <= | > | >=
q16
0-9
;
q17
a-z
++ | --
q18
)
q19
whitespace | ε
{
q10
i
n
q11
t
q12
=
q13
a-z
whitespace
;
whitespace | ε
q5
;
whitespace | ε
q6
)
q7
{
q8
}
q9
whitespace | ε
q20
S
q21
y
q22
s
q23
t
q24
e
q25
m
q26
.
q27
o
q28
u
q29
t
q30
.
q31
p
q32
r
q33
i
q34
n
q35
t
q36
f
q37
n
q38
(
q39
whitespace | ε
"
a-z | A-Z | 0-9 | :
q40
"
q41
+
a-z
q42
)
q43
;
q44
}
q45
)
q46
;
q47
}
q48
)
q49
;
q50
}
q51
error
error
q52
Any value that does not satisfy the accepted values for the current state or succeeding states will lead to an error state.

**Essay 1**

Upon creating the DFA diagram that supported the two for loop syntaxes from the worksheet, I came across a number of problems that were eventually solved. First, as I was creating a sequence of states within the diagram I would later realize I can create another path from a previous state, which resulted me in adjusting several pointers and shifting the states I already created, to make room for a new sequence. For instance, I had to create a sequence that would accept "int i=1;" and then create another one that would accept an empty string or "whitespaces" followed by a semicolon. Another issue I had was trying to handle the parameter value in System.out.println(). The method takes a string value for its parameter and I wanted to make sure that I cover a variety of syntaxes - string literals, string concatenation, identifiers - that are normally found inside System.out.println(). I overcame this issue by drafting down a list of possible values that would normally be accepted in Java, then attempted to create a state diagram that accepted those values. While, for the most part, it was straightforward I addressed concatenation by pointing back to a previous state that has a loopback accepting a variety of characters (a-z, A-Z, 0-9, :), and a route towards statement termination and end of the loop (accepted state). In short, I am getting the hang of using LucidChart to draft out a DFA diagram. Considering the diagram I created only accepted the for loops from the worksheet, it only represents a very small part of a programming language. To which, I am showing a greater appreciation towards compilers for doing a tremendous job in checking the program's syntax for instance.

**Essay 2**

In order to implement the DFA diagram in Java, we would need to read the source code - character by character, for starter. After reading a certain number of characters would we eventually hit a whitespace and then figure out whether that set of characters we just read is a keyword, identifier, data type, symbol, or a constant. Then we would store the set of characters (or word) in a list or array as tokens, in the order that we receive them (so the most recent token would be the last item in the list). Next, we would read the next set of characters until we reach the end of the code. Once we collected all the tokens from the source code, we can send the tokens through a parser, which then checks whether the tokens are properly ordered in a grammatical sense (i.e., there should be a whitespace between "int" and an identifier). Finally, the code would run through a semantic analyzer that checks for valid data types and scope. In the context of this lab, I would store reserved keywords (i.e. for). data types (i.e. int), and symbols (i.e. '(') in their own tables, to which when I read a character I can compare it with any of those tables for a possible match. Furthermore, creating a regular expression pattern that satisfies the for loop syntax make string validation possible in Java.