Alexandru Bara 40132235 Comp 442 assignment 1

Section 1

I did not change the lexical specifications which means it is the same as the problem description. I have decided to add everything under one regular expression which gives me this:

id	::=	Letter alphanum*
alphanum	::=	letter | digit | _
integer	::=	nonzero digit* | **0**
float	::=	integer fraction [**e**[**+**|**-**] integer]
fraction	::=	.digit* nonzero |**.0**
letter	::=	**a..z| A..Z**
digit	::=	**0..9**
nonzero	::=	**1..9**
eq	::=	**==**
plus	::=	**+**
or	::=	**or**
openpar	::=	**(**
semi	::=	**;**
while	::=	**while**
localvar	::=	**localvar**
noteq	::=	**<>**
minus	::=	**-**
and	::=	**and**
closepar	::=	**)**
comma	::=	**,**
if	::=	**if**
constructor	::=	**constructor**
lt	::=	**>**
mult	::=	*
not	::=	**not**
opencubr	::=	**{**
dot	::=	**.**
void	::=	**void**

then	::=	**then**
attribute	::=	**attribute**
gt	::=	**>**
div	::=	**/**
closecubr	::=	**}**
colon	::=	**:**
class	::=	**class**
else	::=	**else**
function	::=	**function**
leq	::=	**<=**
assign	::=	**=**
opensqbr	::=	**[**
returntype	::=	**=>**
self	::=	**self**
read	::=	**read**
public	::=	**public**
geq	::=	**>=**
closesqbr	::=	**]**
scopeop	::=	**::**
isa	::=	**isa**
write	::=	**write**
private	::=	**private**
return	::=	**return**
intnum	::=	**integer**
floatnum	::=	**float**
inlinecmt	::=	**//** nonbreak
blockcmt	::=	**/*** anything ***/**

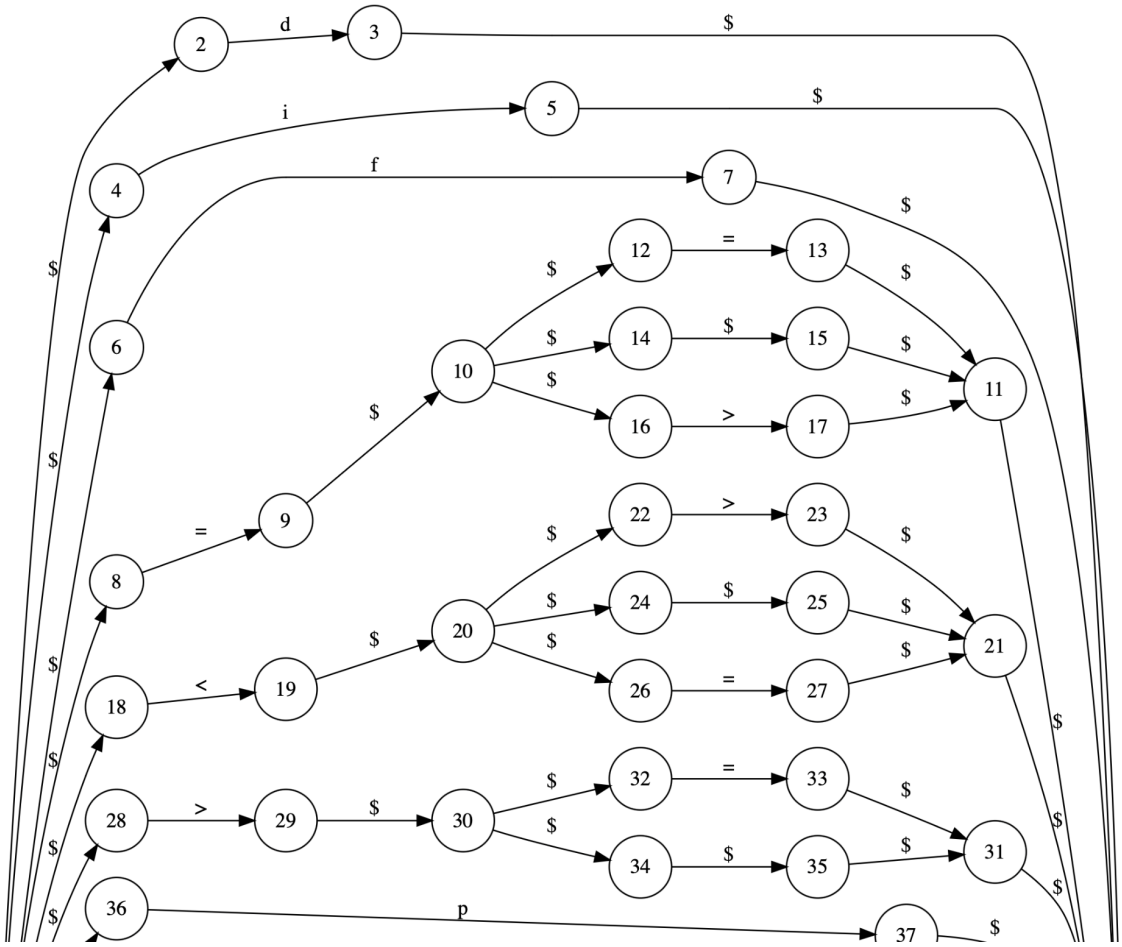
**note: nonbreak represents all characters except the break line character
**note: anything represents all of the alphabet

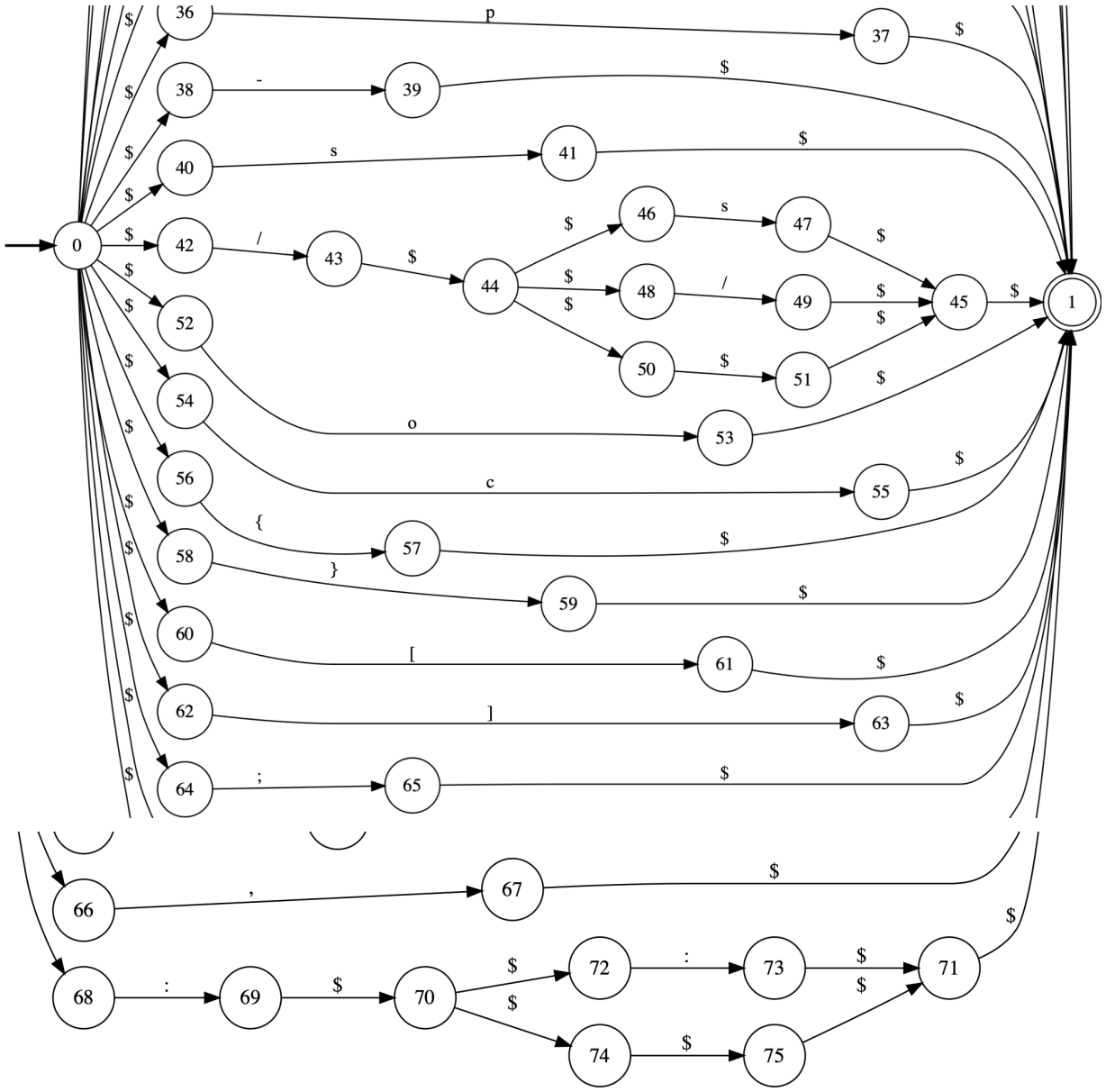
Section 2

I have made a finite state machine for the high level overview of the lexical analyzer and also for all the subparts of it. I can't fit everything under one finite state machine.

Finite state machine for lexical analyzer:

d represents an id
i represents an integer
f represents a float
p represents a +
s represents a *
o represents a (
c represents a)
This finite state machine shows all the different paths the program can take.
In order to represents id, integer and float, I will make another finite state machine for them specifically. The diagram would be too big to show if I did it here. But we can simply take the diagrams for id, float and integer and replace them in this one. Note that I did not go in details for how comments work, but I will in later finite state machines for the comments.

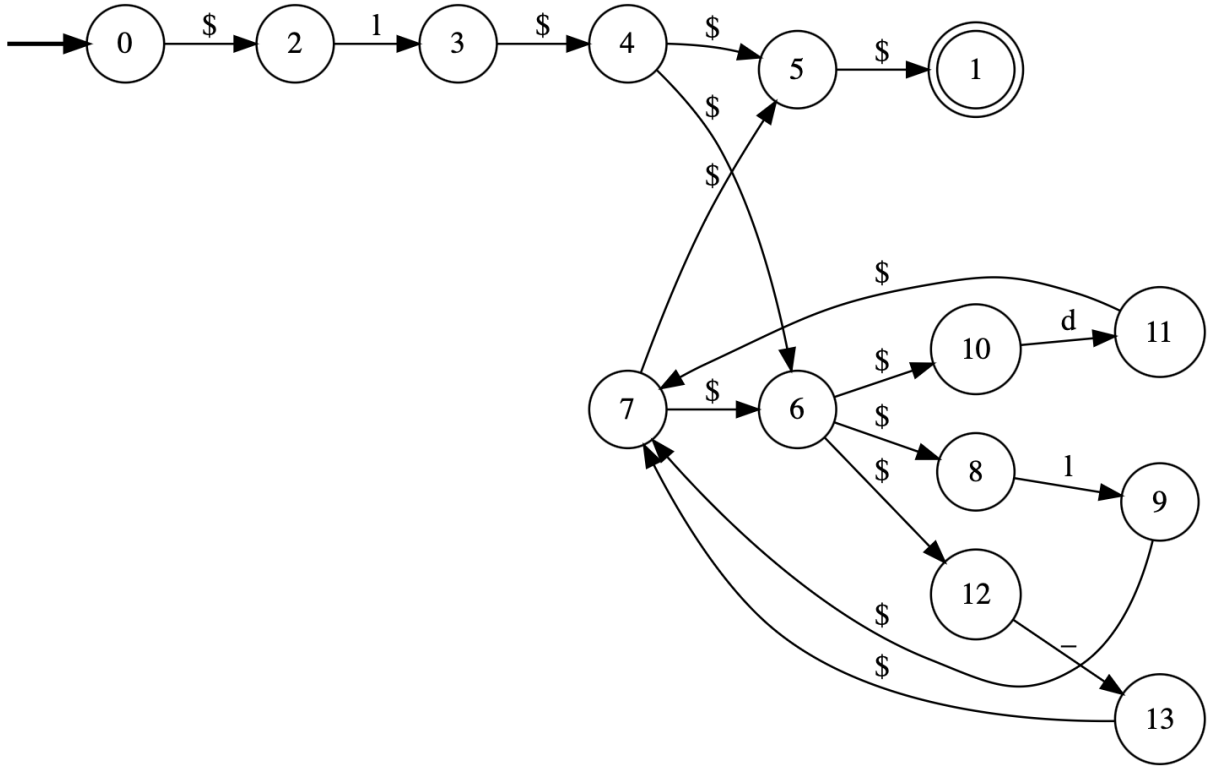




Regular expression for lexical analyzer

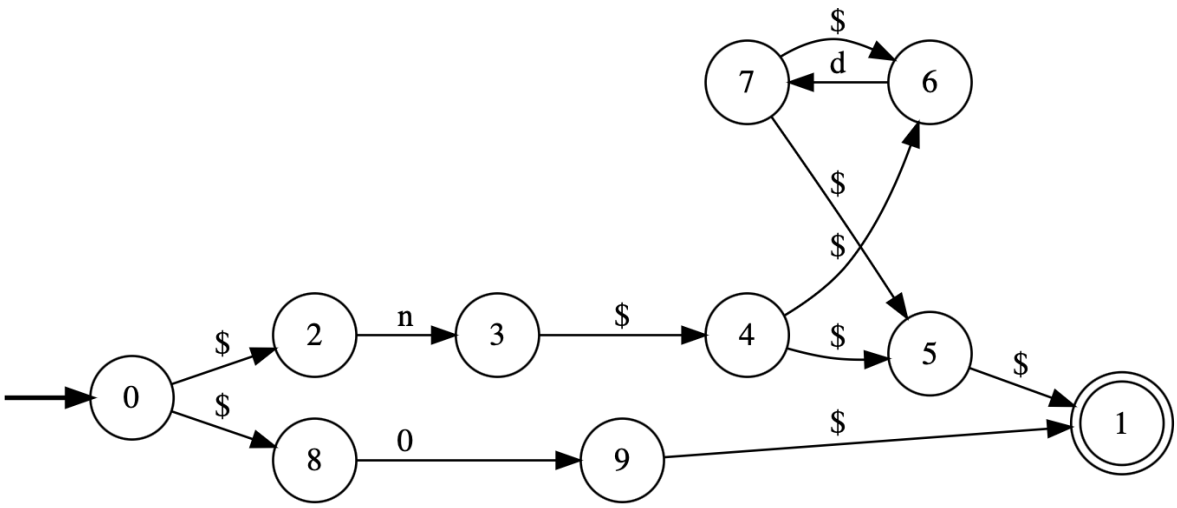
d+i+f+=(=+$+>)+<(>+$+=)+>(=+$)+p+-+s+/+o+c+{+}+[+]+;+,+:(:+$)

Finite state machine for id
l represents a letter
d represents a digit



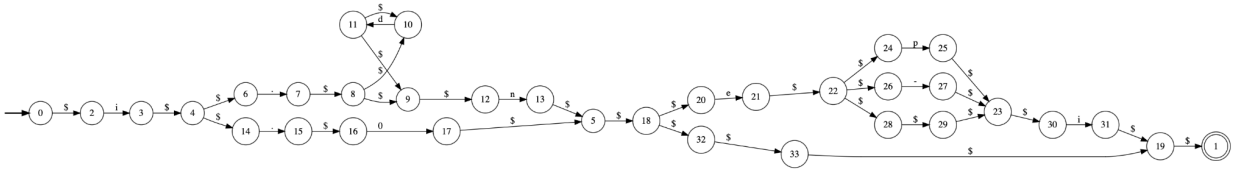
Regular expression for id: l(l+d+_)*

Finite state machine for integer
N represents a nonzero
D represents a digit

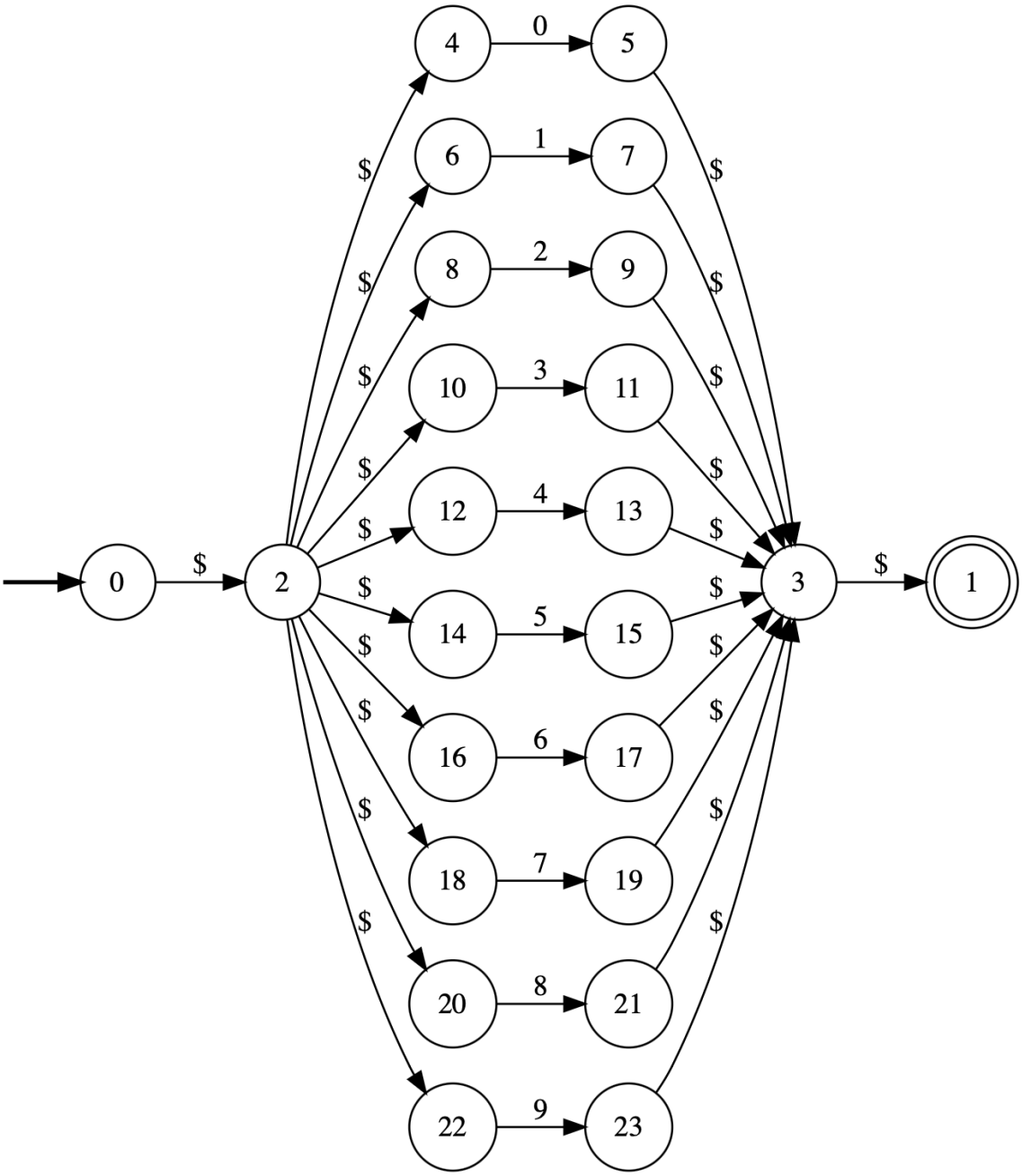


Regular expression for integer: nd*+0

Finite state machine for float
i represents an integer
d represents a digit
n represents a nonzero
p represents a +

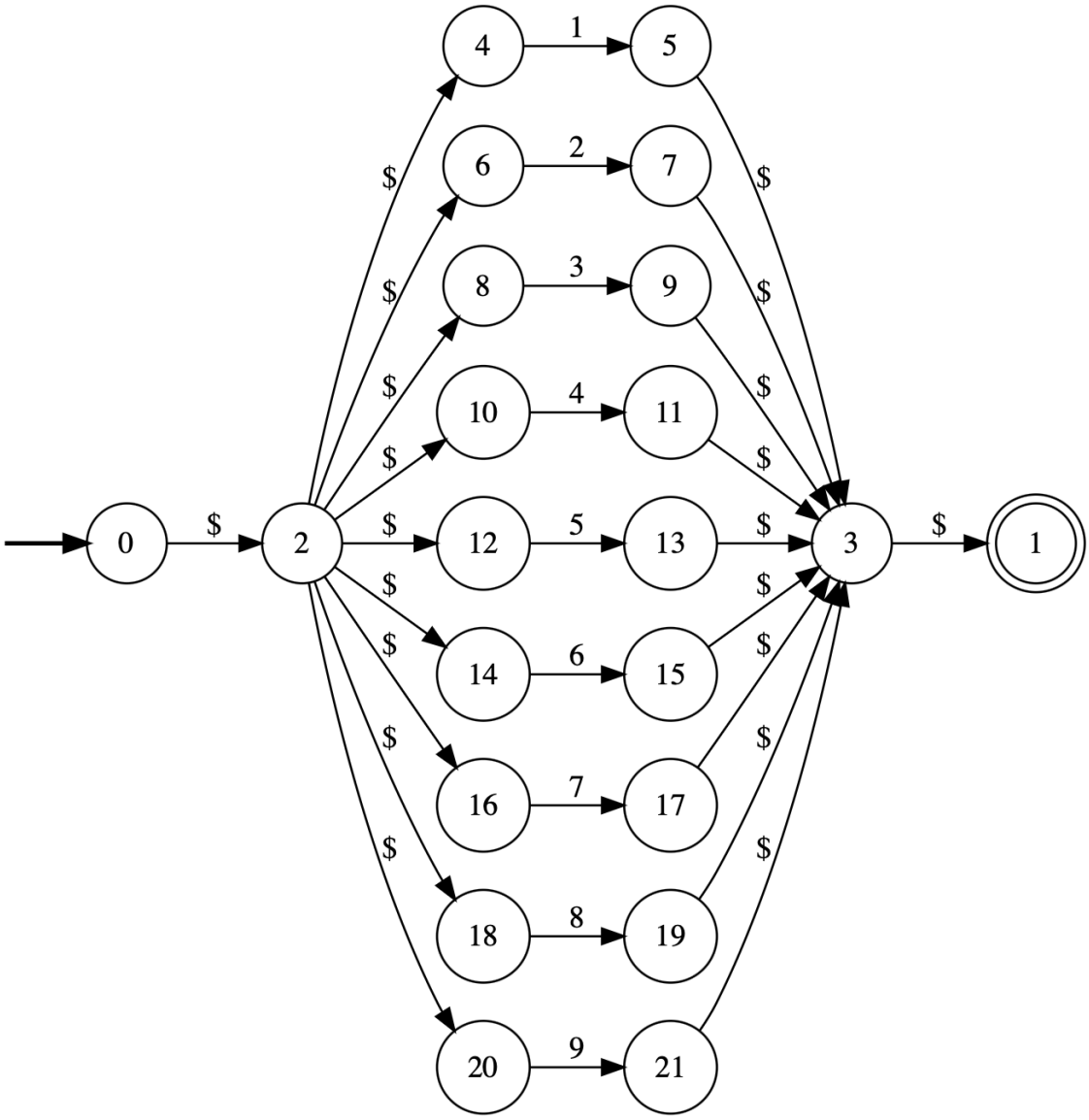


Regular expression for float: i(.d*n+.0)(e(p+-+$)i+$)
Finite state machine for digit



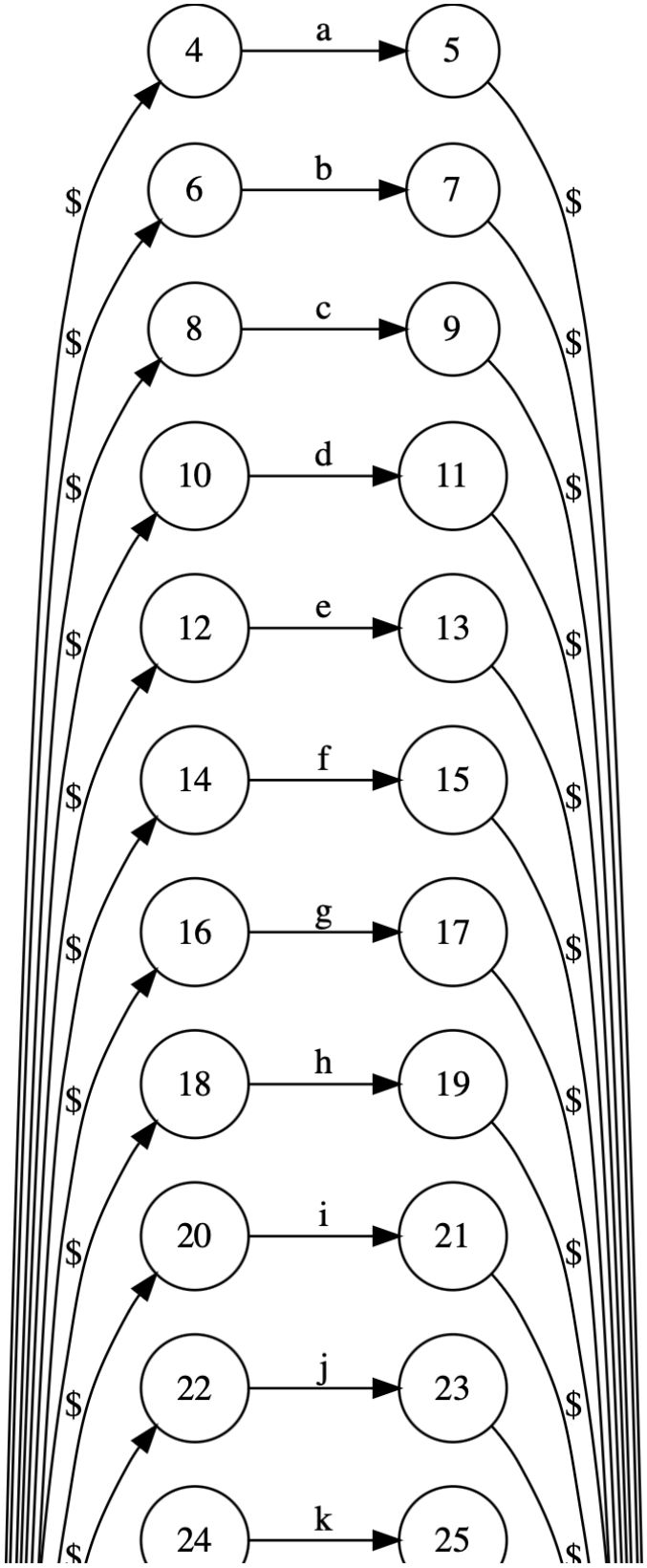
Regular expression for digit: (1+2+3+4+5+6+7+8+9)

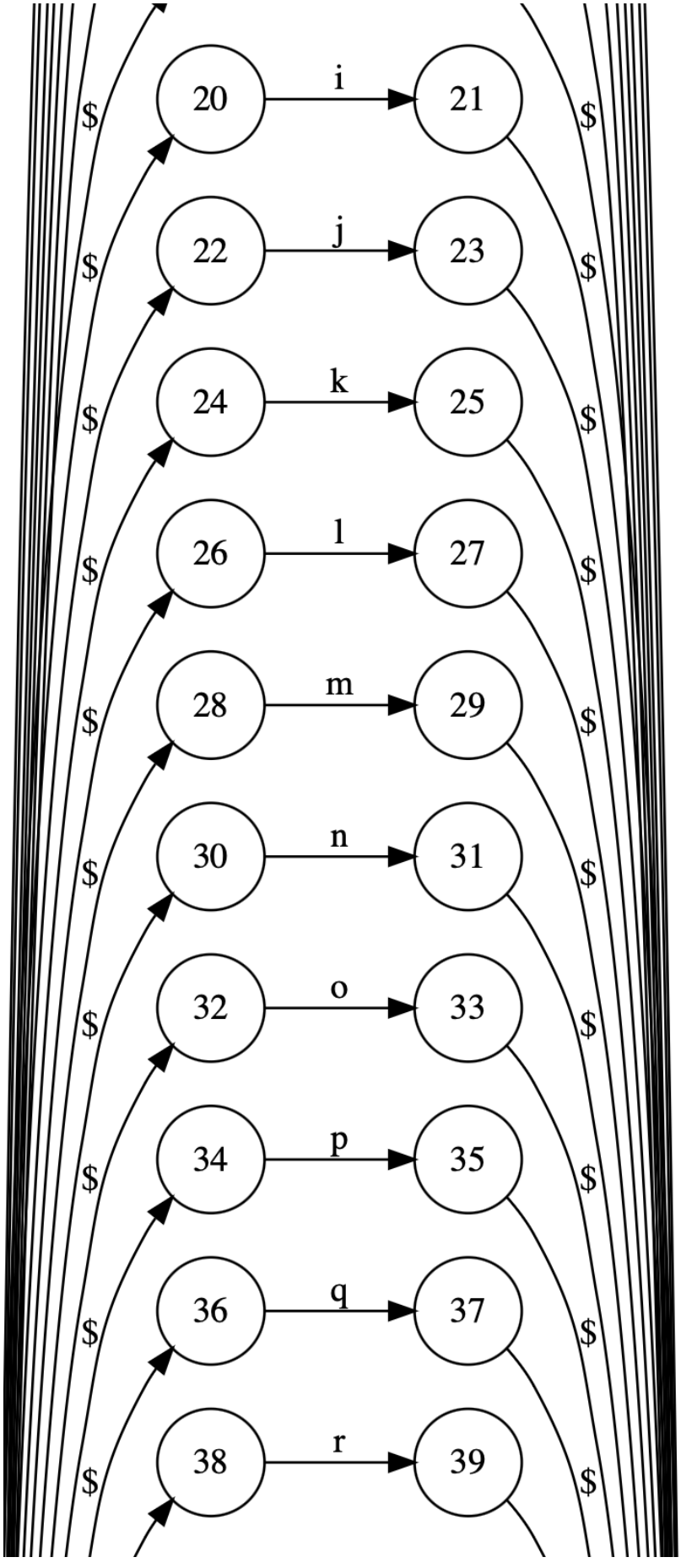
DFA for nonzero

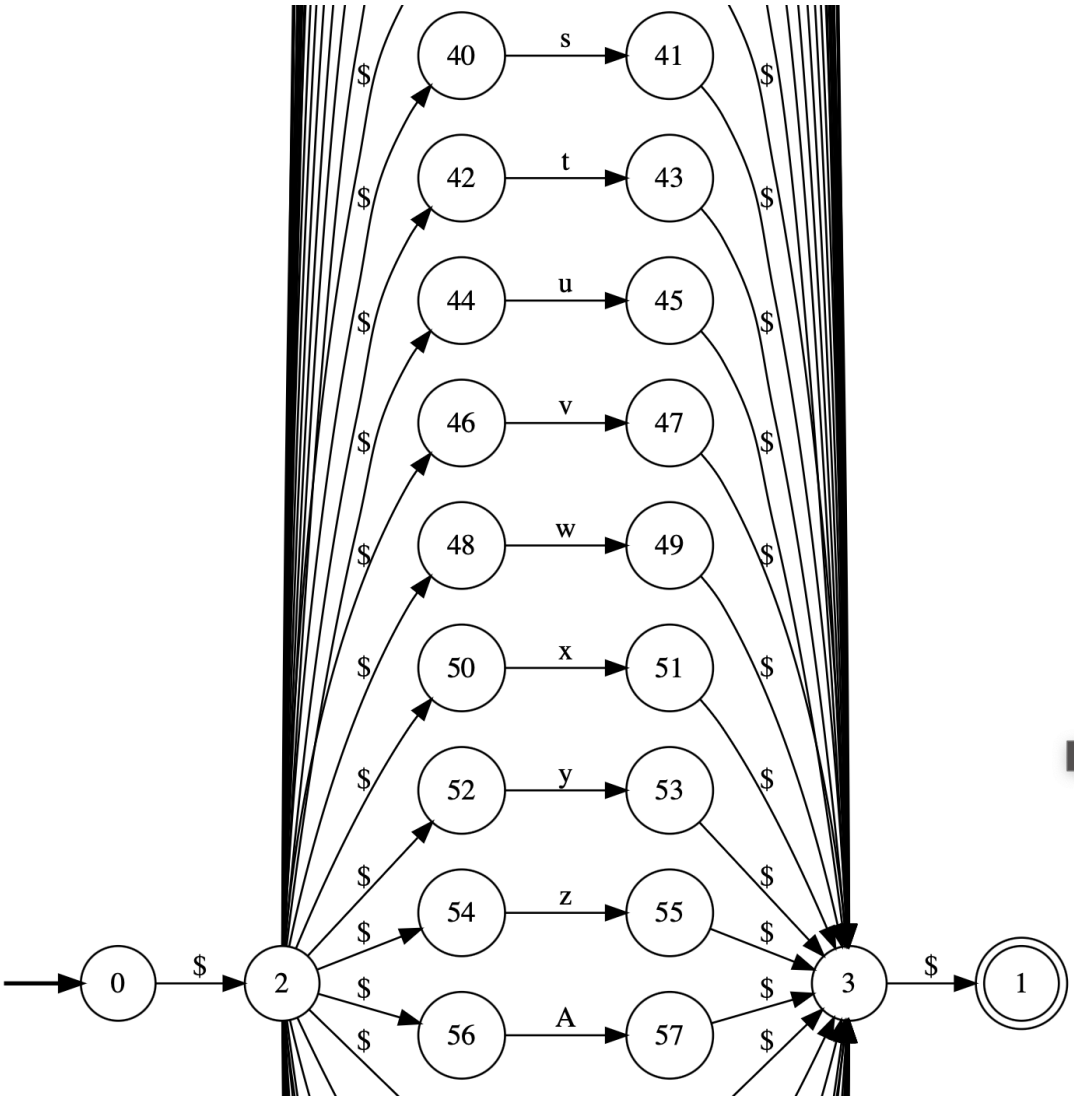


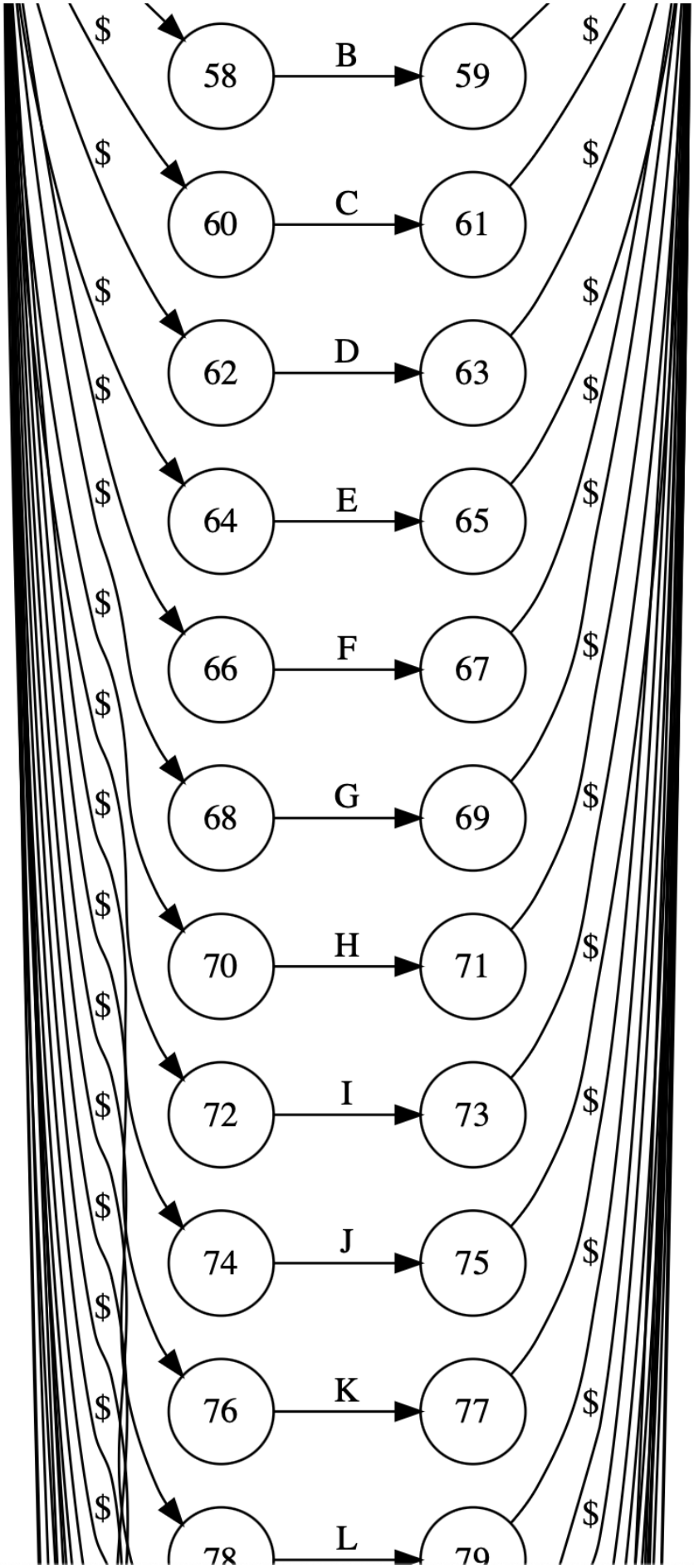
Regular expression for nonzero: (1+2+3+4+5+6+7+8+9)

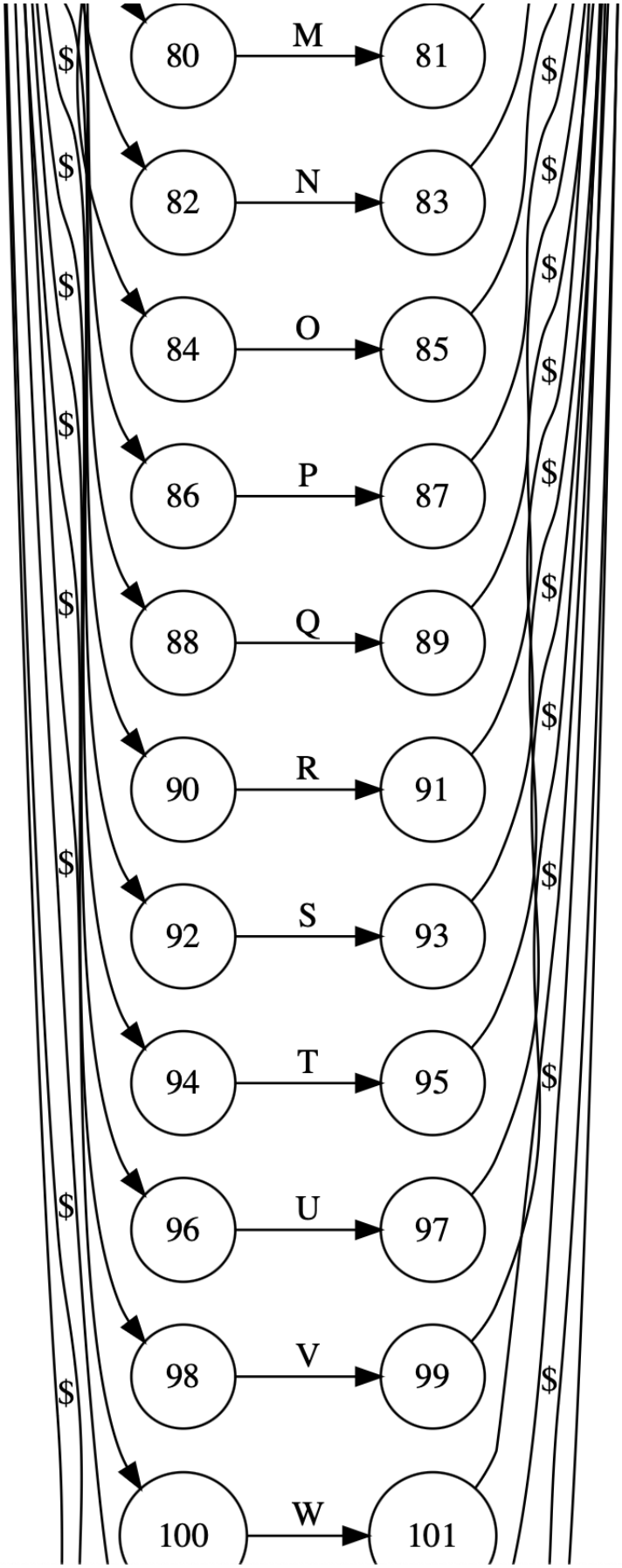
Finite state machine for letter

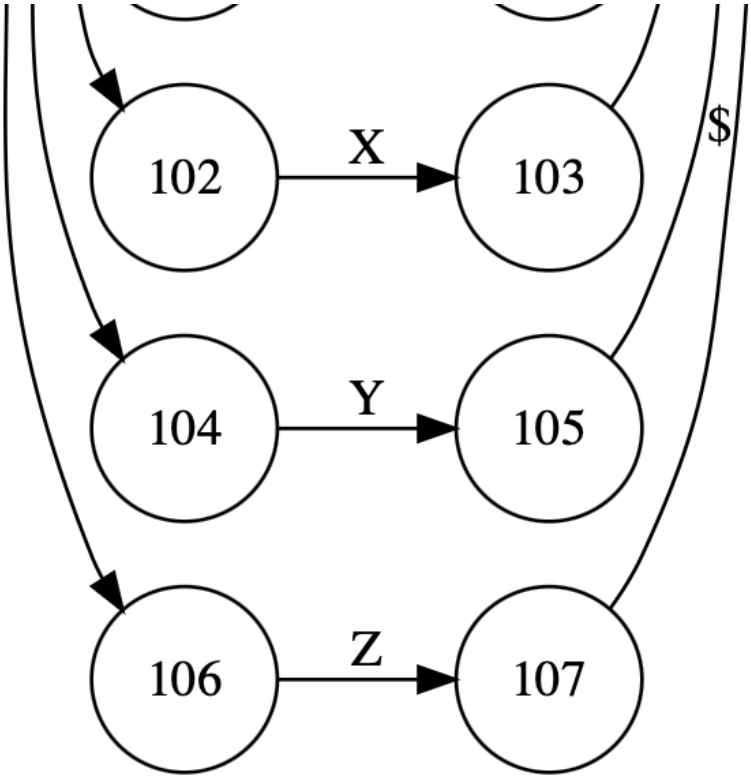

4
a
5
6
b
7
8
c
9
10
d
11
12
e
13
14
f
15
16
g
17
18
h
19
20
i
21
22
j
23
24
k
25
$


20
i
21
22
j
23
24
k
25
26
l
27
28
m
29
30
n
31
32
o
33
34
p
35
36
q
37
38
r
39
$


40
s
41
42
t
43
44
u
45
46
v
47
48
w
49
50
x
51
52
y
53
54
z
55
56
A
57
0
2
3
1
$

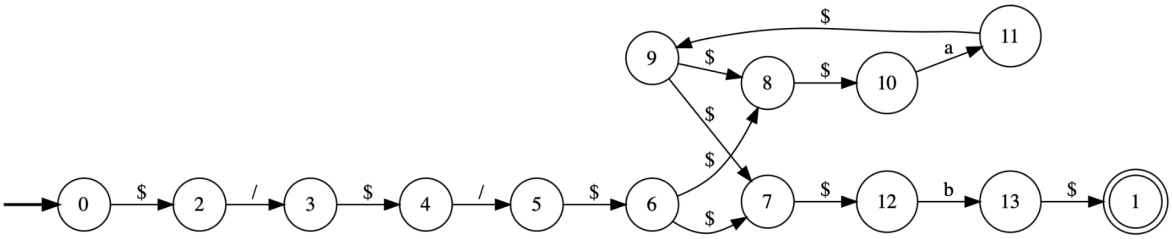

58
B
59
60
C
61
62
D
63
64
E
65
66
F
67
68
G
69
70
H
71
72
I
73
74
J
75
76
K
77
78
L
79
$


80
M
81
82
N
83
84
O
85
86
P
87
88
Q
89
90
R
91
92
S
93
94
T
95
96
U
97
98
V
99
100
W
101
$



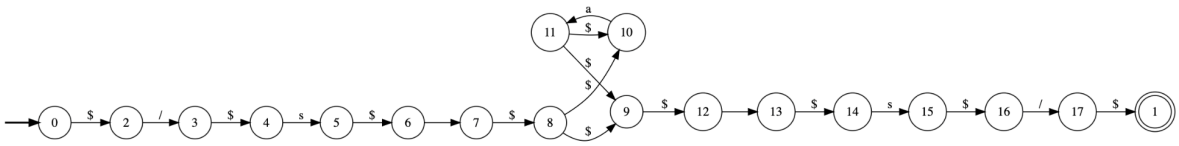
Regular expression for letter:
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z+A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+U+V+W+X+Y+Z)

Finite state machine for inline comment
A represents anything except a line break
B represents a line break



Regular expression for inline comment: //(a)*b

Finite state machine for block comment
S represents a *
A represents anything



Regular expression for block comment: /s a* s/

Section 3

I have a main file that is called lexdriver and is in charge of calling getNextToken and creating the outlextokens/outlexerrors. The actual logic for getNextToken is implemented in the class called LexicalAnalyzer. When an instance of that class is created, it will create a byte array with the bytes of the file as well as keep a current pointer. When getNextToken is called, it will get rid of all blankspaces until it can read a character. Based on what character it reads, it will go down a specific path that has been described in the dfa. For operations like “=”, “=>” and “==” I used backtracking in order to make sure I don’t skip over any characters. For longer tokens like id, float and integer, I read all the elements from the specific alphabet and I use regex afterwards to determine if it is a valid token (alternative 1).

I also have a TokenType enum which keeps track of all the types possible which makes it easier for building the token. Finally, I have a Token class that is responsible for creating the tokens and returning them to the user.

Section 4

I did my lexical analyzer in Java.
I used Stringbuilder to construct the lexeme.
I used Java.io for file manipulation.
I used an enum for my token type (to be able to infer the type)
I used Jest for unit testing and to keep track of my code coverage.