

## PLC Test 2 - Jose Diaz

---

1. (20 points) Create code that allows you to create an ordered list of tokens. This code should take in a file as input and process that file for the following lexemes:
  - All of the code can be found here: [https://github.com/joseishere/jose\\_test2](https://github.com/joseishere/jose_test2)
  - For all of these problems I wrote them in python and made one file that would read all of my test strings from a file
  - I have several files so here is all of the files with their respective code and the last file is the one that combines it all

```

1 # tryFloat.py
2 import re
3
4 def floatChecker(arr):
5
6     # I did this one with regex after people started to talk about
7     # of course all of the other ones I didn't do with regex bc th
8     # really wish i would have done everything with regex
9     # also I used this website to help me build the regex string
10    # it is really well made and you should donate to help keep it
11    # the website is : https://regex101.com
12
13    regexBase = r"(-)?(\d)*(.)?\d+(e|E)?(-)?\d(L|l|f|F)?"
14
15    if(re.fullmatch(regexBase, arr)):
16        # print('is valid')
17        return True
18    else:
19        return False
20
21 def main():
22
23    words = ["23.75", "0.59201E1", "1312221215e-2", "-2.5e-3", "15
24
25    for word in words:
26        if(floatChecker(word)):
27            print(word + " is valid")
28        else:
29            print(word + " is not valid")
30
31 if __name__ == "__main__":
32     main()

```



```

24 #print(letterFound)
25 if(size < 2):
26     return False
27 if(arr[0] == '0' and arr[1] == 'x' or arr[1] == 'X'):
28
29     xo={
30         'a' : 'u',
31         'b' : 'U',
32         'c' : 'l',
33         'd' : 'L'
34     }
35
36     size = len(arr) - 1
37     suffix = [None] * len(arr)
38
39     while(size >= 0):
40         #print(size)
41         if(arr[size] in xo.values()):
42             suffix[size] = arr[size]
43         else:
44             break
45         size -= 1
46     #print(suffix)
47     temp = ''
48     for each in suffix:
49         if(each is not None):
50             temp += each
51     #print(temp)
52     #print("final size : " + str(size))
53
54     if(size != len(arr) - 1):
55         for char in arr[2:size+1]:
56             #print(char)
57             if(char.isnumeric() or char in hex_vals.values()):
58                 #print('what')
59                 pass
60             else:
61                 print('ere')
62                 return False
63         return (True and checkEnd(arr, size+1, 'hex'))
64     else:
65         return True
66
67 elif(arr[0] == '0' and arr[1] < '8'):
68     if(letterFound != -1):

```

```

69         for char in arr[2:letterFound+2]:
70             #print(char)
71             if(char < '8'):
72                 pass
73             else:
74                 #print('ere')
75                 return False
76         return (True and checkEnd(arr, letterFound+2, 'oct'))
77     else:
78         return True
79 elif(arr[0] != 0):
80     if(letterFound != -1):
81         for char in arr[2:letterFound+2]:
82             #print(char)
83             if(char.isnumeric()):
84                 pass
85             else:
86                 #print('ere')
87                 return False
88         return (True and checkEnd(arr, letterFound+2, 'dec'))
89     else:
90         return True
91 else:
92     return False
93
94 def checkEnd(arr, startLetter, type):
95     endString = arr[startLetter:]
96     #print(endString, "end string")
97     dec_endings = {
98         'u': 'u',
99         'l': 'l',
100         'ul': 'ul',
101         'll': 'll',
102         'ull': 'ull',
103     }
104     hex_endings = {
105         'u': 'u',
106         'l': 'l',
107         'uL': 'uL',
108         'lL': 'lL',
109         'uLL': 'uLL',
110     }
111     oct_endings = {
112         'u': 'u',
113         'l': 'l',

```

```

114         'UL':'UL',
115         'll':'ll',
116         'Ull':'Ull',
117     }
118
119     whereToLook = str(type) + "_endings"
120     #print(whereToLook)
121     if(type == 'dec'):
122         if(endString in dec_endings.values()):
123             #print('checkend returned true')
124             return True
125         else:
126             return False
127     elif(type == 'hex'):
128         if(endString in hex_endings.values()):
129             #print('checkend returned true')
130             return True
131         else:
132             return False
133     elif(type == 'oct'):
134         if(endString in oct_endings.values()):
135             #print('checkend returned true')
136             return True
137         else:
138             return False
139     else:
140         return False
141
142 def main():
143
144     words = ["28", "4000000024u", "2000000022l", "4000000000ul", "9000000000ul"]
145
146     for word in words:
147         if(intChecker(word)):
148             print(word + " is valid")
149         else:
150             print(word + " is not valid")
151
152 if __name__ == "__main__":
153     main()

```

```

1 # tryChar.py
2 symbols = {
3     '~' : '~' ,

```

```
4      '\': '\',
5      '!': '!',
6      '@': '@',
7      '#': '#',
8      '$': '$',
9      '%': '%',
10     '^': '^',
11     '&': '&',
12     '*': '*',
13     '(': '(',
14     ')': ')',
15     '-': '-',
16     '_': '_',
17     '+': '+',
18     '=': '=',
19     '{': '{',
20     '[': '[',
21     '}': '}',
22     ']': ']',
23     '|': '|',
24     ':': ':',
25     ';': ';',
26     '<': '<',
27     ',': ',',
28     '>': '>',
29     '.': '.',
30     '?': '?',
31 }
32
33 after_slash = {
34     'b': 'b',
35     'f': 'f',
36     'n': 'n',
37     'r': 'r',
38     '"': '"',
39     '\\': '\\',
40     "'": "'",
41     'v': 'v',
42     'a': 'a',
43     '?': '?',
44     'N': 'N',
45     'X': 'X',
46     't': 't',
47 }
48
```

```

49 def charChecker(arr):
50     # this is very similar to the java string, so took the same al
51     # i think we first need to handle the simplest case just makin
52     size = len(arr)
53     count = 0
54     # need this to handle the /XN
55     isX = False
56
57     # we cant have an empty string or 'a
58     # and we know we can't have anything more than 5
59     if(size >= 5 or size < 3):
60         return False
61     # print(arr[0], arr[-1])
62
63     if((arr[0] == '"' and arr[-1] == '"') or (arr[0] == "'" and ar
64
65     # don't need to check first letter since we know what it is
66     # now we need to loop through the string and if we have a slash
67     # we need to know that the next number, in this case arr[num] is a
68     # and we need to make sure that we only have an even number of sla
69     # is not valid even though you can have a \ after a \
70     num = 1
71     for letter in arr[1:-1]:
72         #print(letter, "printing letter hereeee")
73         num +=1
74         if(letter.isalnum() or letter in symbols.values()):
75             pass
76         elif(letter == '\\'):
77             count+=1
78             # print(count)
79             if(isX):
80                 if(arr[num] != 'N'):
81                     return False
82             elif(arr[num] in after_slash.values()):
83                 #print(str(size) + " size")
84                 if(arr[num] == 'X'):
85                     isX = True
86                     if(num < size-1):
87                         count+=-1
88                         #print(count)
89             elif(arr[num] not in after_slash.values()):
90                 return False
91         else:
92             pass
93     else:

```



```

94         return False
95     if(count == 0):
96         return True
97
98     return True
99
100 def main():
101
102     words = ["\'1\'", "\'!\'", "\"$\"", "\'\\t\'", "\'\\?\'", "\'\\\\
103
104     for word in words:
105         if(charChecker(word)):
106             print(word + " is valid")
107         else:
108             print(word + " is not valid")
109
110 if __name__ == "__main__":
111     main()

```

```

1  # tryJava.py
2  after_slash = {
3      't':'t',
4      'r':'r',
5      'n':'n',
6      'f':'f',
7      '"':'"',
8      '\\':'\\',
9
10 }
11
12 def javaChecker(arr):
13     # i think we first need to handle the simplest case just making
14     size = len(arr)
15     count = 0
16     preCount = 0
17     if(size < 3):
18         return False
19     for letter in arr:
20         if(letter == '\\'):
21             preCount+=1
22
23     if(arr[0] == '"' and arr[1] != "\\" and arr[2] == '"' and len(arr) > 2):
24         return True
25     # this is so that we know that the string starts and ends with quote

```

```

26     if(arr[0] == '' and arr[-1] == ''):
27         # don't need to check first letter since we know what it is
28         # now we need to loop through the string and if we have a slash
29         # we need to know that the next number, in this case arr[num] is a
30         # and we need to make sure that we only have an even number of slashes
31         # is not valid even though you can have a \ after a \
32         num = 1
33         for letter in arr[1:-1]:
34             num +=1
35             if(letter == '\\'):
36                 count+=1
37                 # print(count)
38                 if(arr[num] in after_slash.values()):
39                     if(num < size-1):
40                         count+=-1
41                         # print(count)
42                 if(arr[num] not in after_slash.values()):
43                     return False
44         if(count != 0):
45             return False
46
47         return True
48
49 def main():
50     # the way this works is if you copy and paste my output into a
51     # this is due to how the strings go into the function but it is
52     words = ['a', 'string?', 'string\t', 'str\\', 'stri\\
53
54     for word in words:
55         if(javaChecker(word)):
56             print(word + " is valid")
57         else:
58             print(word + " is not valid")
59
60 if __name__ == '__main__':
61     main()

```

```

1 # tryOperator.py
2 def operatorChecker(op):
3     # we need to make sure that we get an operator and not empty string
4     # and that the length of the operator is never more than 4
5     if(len(op) > 4):
6         return False
7

```

```
8      # now we just see what it is
9
10     if (op == '+' ):
11         return True
12     elif (op == '-' ):
13         return True
14     elif (op == '=' ):
15         return True
16     elif (op == '-'):
17         return True
18     elif (op == '/'):
19         return True
20     elif (op == '*'):
21         return True
22     elif (op == '%'):
23         return True
24     elif (op == '{'):
25         return True
26     elif (op == '}'):
27         return True
28     elif (op == '('):
29         return True
30     elif (op == ')'):
31         return True
32     elif (op == '++'):
33         return True
34     elif (op == '--'):
35         return True
36     elif (op == '&&'):
37         return True
38     elif (op == '||'):
39         return True
40     elif (op == '!'):
41         return True
42     else:
43         return False
44
45 def main():
46     words = ['+', '-', '/', '%', '+-', '', 'faill', '$$', '1']
47
48     for word in words:
49         if(operatorChecker(word)):
50             print(word + " is valid")
51         else:
52             print(word + " is not valid")
```

```
53
54 if __name__ == "__main__":
55     main()
```

```
1  # tryPerl.py
2  def perlChecker(word):
3      if(len(word) < 2):
4          return False
5      foundStart = None
6      for letter in word:
7          if(letter.isalnum() or letter == '$' or letter == '@' or letter == '%'):
8              if (foundStart == None):
9                  if (letter == "$" or letter == "%" or letter == "@"):
10                     foundStart = letter
11                 else:
12                     return False
13             else:
14                 # now we just need numbers or underscore
15                 if( letter.isalnum() or letter == "_"):
16                     pass
17                 else:
18                     return False
19             else:
20                 return False
21     return True
22
23 def main():
24
25     words = ['$var_sas', '@another2', '@test', '%another\s', '$test']
26     for word in words:
27         if(perlChecker(word)):
28             print(word + " is valid")
29         else:
30             print(word + " is not valid")
31
32 if __name__ == "__main__":
33     main()
```

```

1 # getWords.py
2 # gets all of the words from a file
3 def fromFile(fileName):
4     f = open(fileName, 'r')
5     finalList = f.read().split('\n')
6     return finalList
7
8 def main():
9     print(fromFile('testInputs.txt'))
10
11 if __name__ == '__main__':
12     main()

```

```

1 # finalOut.txt
2 # output of above program
3
4 '1'
5 '1' C Char:          VALID
6 '1' C Float: not VALID
7 '1' C Int:           VALID
8 '1' Java String:     VALID
9 '1' Operator: not VALID
10 '1' Perl Identifier: not VALID
11
12 '!'
13 '!' C Char:          VALID
14 '!' C Float: not VALID
15 '!' C Int:           VALID
16 '!' Java String:     VALID
17 '!' Operator: not VALID
18 '!' Perl Identifier: not VALID
19
20 "$"
21 "$" C Char:          VALID
22 "$" C Float: not VALID
23 "$" C Int:           VALID
24 "$" Java String:     VALID
25 "$" Operator: not VALID
26 "$" Perl Identifier: not VALID
27
28 '\t'
29 '\t' C Char:          VALID
30 '\t' C Float: not VALID

```

```
31 '\t' C Int: not VALID
32 '\t' Java String:          VALID
33 '\t' Operator: not VALID
34 '\t' Perl Identifier: not VALID
35
36 '\?'
37 '\?' C Char:              VALID
38 '\?' C Float: not VALID
39 '\?' C Int:              VALID
40 '\?' Java String:          VALID
41 '\?' Operator: not VALID
42 '\?' Perl Identifier: not VALID
43
44 '\\
45 '\\ C Char:              VALID
46 '\\ C Float: not VALID
47 '\\ C Int:              VALID
48 '\\ Java String:          VALID
49 '\\ Operator: not VALID
50 '\\ Perl Identifier: not VALID
51
52 '\f'
53 '\f' C Char:              VALID
54 '\f' C Float: not VALID
55 '\f' C Int: not VALID
56 '\f' Java String:          VALID
57 '\f' Operator: not VALID
58 '\f' Perl Identifier: not VALID
59
60 '\xN'
61 '\xN' C Char: not VALID
62 '\xN' C Float: not VALID
63 '\xN' C Int: not VALID
64 '\xN' Java String:          VALID
65 '\xN' Operator: not VALID
66 '\xN' Perl Identifier: not VALID
67
68 ']'
69 ']' C Char:              VALID
70 ']' C Float: not VALID
71 ']' C Int:              VALID
72 ']' Java String:          VALID
73 ']' Operator: not VALID
74 ']' Perl Identifier: not VALID
75
```

```

76 'n'
77 'n' C Char:          VALID
78 'n' C Float: not VALID
79 'n' C Int:           VALID
80 'n' Java String:     VALID
81 'n' Operator: not VALID
82 'n' Perl Identifier: not VALID
83
84 'e"
85 'e" C Char: not VALID
86 'e" C Float: not VALID
87 'e" C Int:           VALID
88 'e" Java String:     VALID
89 'e" Operator: not VALID
90 'e" Perl Identifier: not VALID
91
92 v\'v\'
93 v\'v\' C Char: not VALID
94 v\'v\' C Float: not VALID
95 v\'v\' C Int: not VALID
96 v\'v\' Java String:   VALID
97 v\'v\' Operator: not VALID
98 v\'v\' Perl Identifier: not VALID
99
100 23.75
101 23.75 C Char: not VALID
102 23.75 C Float:       VALID
103 23.75 C Int:         VALID
104 23.75 Java String:   VALID
105 23.75 Operator: not VALID
106 23.75 Perl Identifier: not VALID
107
108 0.59201E1
109 0.59201E1 C Char: not VALID
110 0.59201E1 C Float:   VALID
111 0.59201E1 C Int: not VALID
112 0.59201E1 Java String: VALID
113 0.59201E1 Operator: not VALID
114 0.59201E1 Perl Identifier: not VALID
115
116 1312221215e-2
117 1312221215e-2 C Char: not VALID
118 1312221215e-2 C Float:   VALID
119 1312221215e-2 C Int: not VALID
120 1312221215e-2 Java String: VALID

```

121	1312221215e-2	Operator: not VALID
122	1312221215e-2	Perl Identifier: not VALID
123		
124	-2.5e-3	
125	-2.5e-3	C Char: not VALID
126	-2.5e-3	C Float: VALID
127	-2.5e-3	C Int: not VALID
128	-2.5e-3	Java String: VALID
129	-2.5e-3	Operator: not VALID
130	-2.5e-3	Perl Identifier: not VALID
131		
132	15E-4	
133	15E-4	C Char: not VALID
134	15E-4	C Float: VALID
135	15E-4	C Int: not VALID
136	15E-4	Java String: VALID
137	15E-4	Operator: not VALID
138	15E-4	Perl Identifier: not VALID
139		
140	121.0L	
141	121.0L	C Char: not VALID
142	121.0L	C Float: not VALID
143	121.0L	C Int: not VALID
144	121.0L	Java String: VALID
145	121.0L	Operator: not VALID
146	121.0L	Perl Identifier: not VALID
147		
148	122.0F	
149	122.0F	C Char: not VALID
150	122.0F	C Float: not VALID
151	122.0F	C Int: not VALID
152	122.0F	Java String: VALID
153	122.0F	Operator: not VALID
154	122.0F	Perl Identifier: not VALID
155		
156	1x0.0F	
157	1x0.0F	C Char: not VALID
158	1x0.0F	C Float: not VALID
159	1x0.0F	C Int: not VALID
160	1x0.0F	Java String: VALID
161	1x0.0F	Operator: not VALID
162	1x0.0F	Perl Identifier: not VALID
163		
164	.02ef3	
165	.02ef3	C Char: not VALID



```

166 .02ef3 C Float: not VALID
167 .02ef3 C Int: not VALID
168 .02ef3 Java String: VALID
169 .02ef3 Operator: not VALID
170 .02ef3 Perl Identifier: not VALID
171
172 0.01ee1
173 0.01ee1 C Char: not VALID
174 0.01ee1 C Float: not VALID
175 0.01ee1 C Int: not VALID
176 0.01ee1 Java String: VALID
177 0.01ee1 Operator: not VALID
178 0.01ee1 Perl Identifier: not VALID
179
180 0.5e1lf
181 0.5e1lf C Char: not VALID
182 0.5e1lf C Float: not VALID
183 0.5e1lf C Int: not VALID
184 0.5e1lf Java String: VALID
185 0.5e1lf Operator: not VALID
186 0.5e1lf Perl Identifier: not VALID
187
188 69e--2
189 69e--2 C Char: not VALID
190 69e--2 C Float: not VALID
191 69e--2 C Int: not VALID
192 69e--2 Java String: VALID
193 69e--2 Operator: not VALID
194 69e--2 Perl Identifier: not VALID
195
196 28
197 28 C Char: not VALID
198 28 C Float: VALID
199 28 C Int: VALID
200 28 Java String: not VALID
201 28 Operator: not VALID
202 28 Perl Identifier: not VALID
203
204 4000000024u
205 4000000024u C Char: not VALID
206 4000000024u C Float: not VALID
207 4000000024u C Int: VALID
208 4000000024u Java String: VALID
209 4000000024u Operator: not VALID
210 4000000024u Perl Identifier: not VALID

```

```

211
212 2000000022l
213 2000000022l C Char: not VALID
214 2000000022l C Float:          VALID
215 2000000022l C Int:            VALID
216 2000000022l Java String:      VALID
217 2000000022l Operator: not VALID
218 2000000022l Perl Identifier: not VALID
219
220 4000000000ul
221 4000000000ul C Char: not VALID
222 4000000000ul C Float: not VALID
223 4000000000ul C Int:            VALID
224 4000000000ul Java String:      VALID
225 4000000000ul Operator: not VALID
226 4000000000ul Perl Identifier: not VALID
227
228 9000000000LL
229 9000000000LL C Char: not VALID
230 9000000000LL C Float: not VALID
231 9000000000LL C Int:            VALID
232 9000000000LL Java String:      VALID
233 9000000000LL Operator: not VALID
234 9000000000LL Perl Identifier: not VALID
235
236 900000000001ull
237 900000000001ull C Char: not VALID
238 900000000001ull C Float: not VALID
239 900000000001ull C Int:            VALID
240 900000000001ull Java String:      VALID
241 900000000001ull Operator: not VALID
242 900000000001ull Perl Identifier: not VALID
243
244 024
245 024 C Char: not VALID
246 024 C Float:          VALID
247 024 C Int:            VALID
248 024 Java String:      VALID
249 024 Operator: not VALID
250 024 Perl Identifier: not VALID
251
252 04000000024u
253 04000000024u C Char: not VALID
254 04000000024u C Float: not VALID
255 04000000024u C Int:            VALID

```

256	04000000024u	Java String:	VALID
257	04000000024u	Operator:	not VALID
258	04000000024u	Perl Identifier:	not VALID
259			
260	02000000022l		
261	02000000022l	C Char:	not VALID
262	02000000022l	C Float:	VALID
263	02000000022l	C Int:	VALID
264	02000000022l	Java String:	VALID
265	02000000022l	Operator:	not VALID
266	02000000022l	Perl Identifier:	not VALID
267			
268	04000000000UL		
269	04000000000UL	C Char:	not VALID
270	04000000000UL	C Float:	not VALID
271	04000000000UL	C Int:	VALID
272	04000000000UL	Java String:	VALID
273	04000000000UL	Operator:	not VALID
274	04000000000UL	Perl Identifier:	not VALID
275			
276	04400000000000l		
277	04400000000000l	C Char:	not VALID
278	04400000000000l	C Float:	not VALID
279	04400000000000l	C Int:	VALID
280	04400000000000l	Java String:	VALID
281	04400000000000l	Operator:	not VALID
282	04400000000000l	Perl Identifier:	not VALID
283			
284	0444000000000000l		
285	0444000000000000l	C Char:	not VALID
286	0444000000000000l	C Float:	not VALID
287	0444000000000000l	C Int:	VALID
288	0444000000000000l	Java String:	VALID
289	0444000000000000l	Operator:	not VALID
290	0444000000000000l	Perl Identifier:	not VALID
291			
292	0x2a		
293	0x2a	C Char:	not VALID
294	0x2a	C Float:	not VALID
295	0x2a	C Int:	VALID
296	0x2a	Java String:	VALID
297	0x2a	Operator:	not VALID
298	0x2a	Perl Identifier:	not VALID
299			
300	0XA0000024uu		

```

301 0XA0000024uu C Char: not VALID
302 0XA0000024uu C Float: not VALID
303 0XA0000024uu C Int: not VALID
304 0XA0000024uu Java String:          VALID
305 0XA0000024uu Operator: not VALID
306 0XA0000024uu Perl Identifier: not VALID
307
308 0x20000022ll
309 0x20000022ll C Char: not VALID
310 0x20000022ll C Float: not VALID
311 0x20000022ll C Int:          VALID
312 0x20000022ll Java String:      VALID
313 0x20000022ll Operator: not VALID
314 0x20000022ll Perl Identifier: not VALID
315
316 0XA0000021uLLL
317 0XA0000021uLLL C Char: not VALID
318 0XA0000021uLLL C Float: not VALID
319 0XA0000021uLLL C Int: not VALID
320 0XA0000021uLLL Java String:      VALID
321 0XA0000021uLLL Operator: not VALID
322 0XA0000021uLLL Perl Identifier: not VALID
323
324 0x8a000000000000lll
325 0x8a000000000000lll C Char: not VALID
326 0x8a000000000000lll C Float: not VALID
327 0x8a000000000000lll C Int: not VALID
328 0x8a000000000000lll Java String:      VALID
329 0x8a000000000000lll Operator: not VALID
330 0x8a000000000000lll Perl Identifier: not VALID
331
332 0x8A40000000000010uLLL
333 0x8A40000000000010uLLL C Char: not VALID
334 0x8A40000000000010uLLL C Float: not VALID
335 0x8A40000000000010uLLL C Int: not VALID
336 0x8A40000000000010uLLL Java String:      VALID
337 0x8A40000000000010uLLL Operator: not VALID
338 0x8A40000000000010uLLL Perl Identifier: not VALID
339
340 "a"
341 "a" C Char:          VALID
342 "a" C Float: not VALID
343 "a" C Int:          VALID
344 "a" Java String:      VALID
345 "a" Operator: not VALID

```

```
346 "a" Perl Identifier: not VALID
347
348 "string?"
349 "string?" C Char: not VALID
350 "string?" C Float: not VALID
351 "string?" C Int: not VALID
352 "string?" Java String:          VALID
353 "string?" Operator: not VALID
354 "string?" Perl Identifier: not VALID
355
356 "string\t"
357 "string\t" C Char: not VALID
358 "string\t" C Float: not VALID
359 "string\t" C Int: not VALID
360 "string\t" Java String:          VALID
361 "string\t" Operator: not VALID
362 "string\t" Perl Identifier: not VALID
363
364 "str\\"
365 "str\\" C Char: not VALID
366 "str\\" C Float: not VALID
367 "str\\" C Int: not VALID
368 "str\\" Java String: not VALID
369 "str\\" Operator: not VALID
370 "str\\" Perl Identifier: not VALID
371
372 "stri\\"s"
373 "stri\\"s" C Char: not VALID
374 "stri\\"s" C Float: not VALID
375 "stri\\"s" C Int: not VALID
376 "stri\\"s" Java String:          VALID
377 "stri\\"s" Operator: not VALID
378 "stri\\"s" Perl Identifier: not VALID
379
380 "st\\"ri\\"s"
381 "st\\"ri\\"s" C Char: not VALID
382 "st\\"ri\\"s" C Float: not VALID
383 "st\\"ri\\"s" C Int: not VALID
384 "st\\"ri\\"s" Java String:          VALID
385 "st\\"ri\\"s" Operator: not VALID
386 "st\\"ri\\"s" Perl Identifier: not VALID
387
388 "valid??@123"
389 "valid??@123" C Char: not VALID
390 "valid??@123" C Float: not VALID
```

```
391 "valid??@123" C Int: not VALID
392 "valid??@123" Java String: VALID
393 "valid??@123" Operator: not VALID
394 "valid??@123" Perl Identifier: not VALID
395
396 "val33\\{1!@#$$%"
397 "val33\\{1!@#$$%" C Char: not VALID
398 "val33\\{1!@#$$%" C Float: not VALID
399 "val33\\{1!@#$$%" C Int: not VALID
400 "val33\\{1!@#$$%" Java String: not VALID
401 "val33\\{1!@#$$%" Operator: not VALID
402 "val33\\{1!@#$$%" Perl Identifier: not VALID
403
404 +
405 + C Char: not VALID
406 + C Float: not VALID
407 + C Int: not VALID
408 + Java String: not VALID
409 + Operator: VALID
410 + Perl Identifier: not VALID
411
412 -
413 - C Char: not VALID
414 - C Float: not VALID
415 - C Int: not VALID
416 - Java String: not VALID
417 - Operator: VALID
418 - Perl Identifier: not VALID
419
420 /
421 / C Char: not VALID
422 / C Float: not VALID
423 / C Int: not VALID
424 / Java String: not VALID
425 / Operator: VALID
426 / Perl Identifier: not VALID
427
428 %
429 % C Char: not VALID
430 % C Float: not VALID
431 % C Int: not VALID
432 % Java String: not VALID
433 % Operator: VALID
434 % Perl Identifier: not VALID
435
```

```

436 +-
437 +- C Char: not VALID
438 +- C Float: not VALID
439 +- C Int:          VALID
440 +- Java String: not VALID
441 +- Operator: not VALID
442 +- Perl Identifier: not VALID
443
444 fail
445 fail C Char: not VALID
446 fail C Float: not VALID
447 fail C Int: not VALID
448 fail Java String:          VALID
449 fail Operator: not VALID
450 fail Perl Identifier: not VALID
451
452 $$
453 $$ C Char: not VALID
454 $$ C Float: not VALID
455 $$ C Int:          VALID
456 $$ Java String: not VALID
457 $$ Operator: not VALID
458 $$ Perl Identifier: not VALID
459
460 1
461 1 C Char: not VALID
462 1 C Float: not VALID
463 1 C Int: not VALID
464 1 Java String: not VALID
465 1 Operator: not VALID
466 1 Perl Identifier: not VALID
467
468 $var_sas
469 $var_sas C Char: not VALID
470 $var_sas C Float: not VALID
471 $var_sas C Int: not VALID
472 $var_sas Java String:          VALID
473 $var_sas Operator: not VALID
474 $var_sas Perl Identifier:          VALID
475
476 @another2
477 @another2 C Char: not VALID
478 @another2 C Float: not VALID
479 @another2 C Int: not VALID
480 @another2 Java String:          VALID

```

```

481 @another2 Operator: not VALID
482 @another2 Perl Identifier:          VALID
483
484 @test
485 @test C Char: not VALID
486 @test C Float: not VALID
487 @test C Int: not VALID
488 @test Java String:                  VALID
489 @test Operator: not VALID
490 @test Perl Identifier:              VALID
491
492 %another\s
493 %another\s C Char: not VALID
494 %another\s C Float: not VALID
495 %another\s C Int: not VALID
496 %another\s Java String:             VALID
497 %another\s Operator: not VALID
498 %another\s Perl Identifier: not VALID
499
500 $test_$
501 $test_$ C Char: not VALID
502 $test_$ C Float: not VALID
503 $test_$ C Int: not VALID
504 $test_$ Java String:                VALID
505 $test_$ Operator: not VALID
506 $test_$ Perl Identifier: not VALID
507
508 #testt
509 #testt C Char: not VALID
510 #testt C Float: not VALID
511 #testt C Int: not VALID
512 #testt Java String:                 VALID
513 #testt Operator: not VALID
514 #testt Perl Identifier: not VALID
515
516 @test#w
517 @test#w C Char: not VALID
518 @test#w C Float: not VALID
519 @test#w C Int: not VALID
520 @test#w Java String:                VALID
521 @test#w Operator: not VALID
522 @test#w Perl Identifier: not VALID
523
524 C Char: not VALID
525 C Float: not VALID

```



526	C Int: not VALID
527	Java String: not VALID
528	Operator: not VALID
529	Perl Identifier: not VALID

2. (9 points) Write three functions in C or C++: one that declares a large array statically, one that declares the same large array on the stack, and one that creates the same large array from the heap. Call each of the subprograms a large number of times (at least 100,000) and output the time required by each. Explain the results.

- For this program I wrote mine in C. This question was actually not too hard and I enjoyed it because it had been a while since I programmed in C.
- Anyways, for my program I created arrays that were of length/size 10000 and then I ran each function for 10,000,000. AKA 10 million times
- After waiting for the execution the results were the following:
- the stack function was the quickest taking only .020545 seconds
- Second quickest was the static function which took .021356 seconds
- And finally the slowest by far was the heap function which took 15.688600 seconds
- I believe that the reason the stack is the fastest is because pushing on to the stack is incredibly easy. It probably happens instantly. While when we are doing the heap we have to allocate space in memory and then deal with pointers and anytime we have to go to memory we are slowed down significantly.
- The stack is better for anything short term that only needs to be there when the function is alive then we can use the stack.
- But if we are dealing with big arrays or structs that can change in size then we should use the heap.
- I have included the output in a txt file as well
- And here is the source code
- jose\_question2.c

```
1 // question2_jose.c
2
3     #include <stdio.h>
4     #include <stdlib.h>
5     #include <time.h>
6
7     int staticArr();
8     int stackArr();
9     int heapArr();
10
11     int main() {
```

```

12
13 // Now here I will try to get the amount of time that each f
14 // I will run it 10000000 times bc i don't want to blow up m
15
16 // the code for finding out how long it takes to run is from
17 // geeks for geeks
18 // https://www.geeksforgeeks.org/how-to-measure-time-taken-by
19
20 printf("Hello, World! \n");
21
22 clock_t first;
23 first = clock();
24
25 for(int i = 0; i < 10000000; i++){
26     staticArr();
27 }
28
29 first = clock() - first;
30 double firstTime = ((double)first) / CLOCKS_PER_SEC;
31 printf("staticArr() took %f seconds to execute \n", firstTi
32
33 clock_t second;
34 second = clock();
35
36 for(int i = 0; i < 10000000; i++){
37     stackArr();
38 }
39
40 second = clock() - second;
41 double secondTime = ((double)second) / CLOCKS_PER_SEC;
42 printf("stackArr() took %f seconds to execute \n", secondTi
43
44 clock_t third;
45 third = clock();
46
47 for(int i = 0; i < 10000000; i++){
48     heapArr();
49 }
50
51 third = clock() - third;
52 double thirdTime = ((double)third) / CLOCKS_PER_SEC;
53 printf("heapArr() took %f seconds to execute \n", thirdTime
54
55 printf("----- Jose Diaz -----\n");
56 return 0;

```

```

57     }
58
59     int staticArr() {
60
61         // not sure if this is correct. have to declare 'statically'
62         // keyword sooo. But I read that static means that the variable is stored
63         // from the file that created it but still shaky on this
64         // i read the info from here: http://www.mathcs.emory.edu/~cl
65
66         static int balance[10000];
67
68         // for(int i = 0; i < 1000; i++){
69         //     balance[i] = i;
70         //     //printf("%d", balance[i]);
71         // }
72
73         return 0;
74     }
75
76     int stackArr() {
77
78         // this one is also weird bc whenever you make a variable in a function, it's
79         // sooooo
80         // this is it???
81
82         int stacked[10000];
83
84         return 0;
85     }
86
87     int heapArr(){
88
89         // Inspiration for this part came from:
90         // https://gribblelab.org/CBootCamp/7\_Memory\_Stack\_vs\_Heap.h
91         // this part showed that to do it in a heap you had to use
92         // the malloc function
93
94         int *heaped = malloc(10000 * sizeof(int));
95
96         return 0;
97     }

```

- output is also here:

```

1 | Hello, World!
2 | staticArr() took 0.021356 seconds to execute
3 | stackArr() took 0.020545 seconds to execute
4 | heapArr() took 15.688600 seconds to execute
5 | ---- Jose Diaz ----

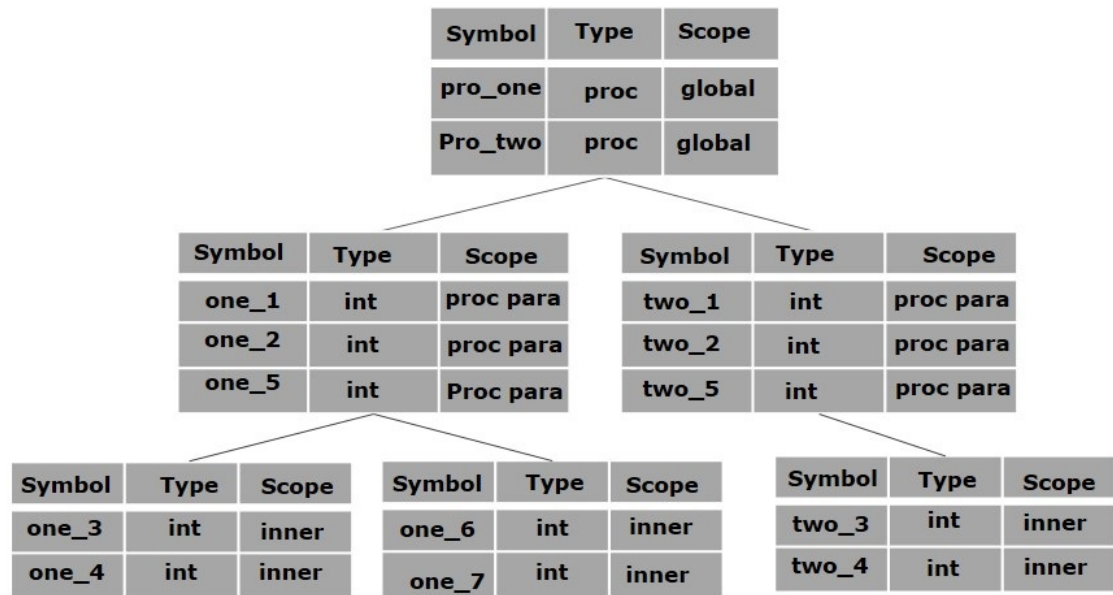
```

3. (11 points) Write an EBNF or CFG that while handle prefix/preorder Arithmetic Operations (addition, subtraction, multiplication, division, modulo) with the proper order of operations? What all types of parsers can be used to show the syntax for this? Justify your answer.

- $\langle \text{statement} \rangle ::= \langle \text{statement} \rangle \backslash '*' \langle \text{term} \rangle \mid \langle \text{statement} \rangle \backslash '/' \langle \text{term} \rangle \mid \langle \text{statement} \rangle \backslash \% \langle \text{term} \rangle \mid$
- $\langle \text{term} \rangle ::= \langle \text{term} \rangle '+' \langle \text{var} \rangle \mid \langle \text{term} \rangle '-' \langle \text{var} \rangle \mid \langle \text{var} \rangle$
- $\langle \text{var} \rangle ::= [\langle \text{letter} \rangle \mid \langle \text{number} \rangle]$
- $\langle \text{letter} \rangle ::= [A-Z \mid a-z]$
- $\langle \text{num} \rangle ::= [0-9]$
- All top down parsers can work with this CFG because they all build the parse tree from the top down and then you read left to right
- This CFG I based off of the one from the PLC textbook. I believe this will work because in the textbook they build the derivation from the top down and it works and precedence is maintained I beleive

4. (10 points) What features of the compilation process allow us to determine the reference environment for any at any given line of code in the program. Answer this question for both dynamic and static scoping? Does the type of scoping change this answer? Explain why?

- The symbol table is the part of the compilation process that will help us determine the scope of a variable. This is because "it stores information about the occurence of various entities such as variable names, function names, objects, classes, interfaces, etc" from Tutorials Point. Found here:  
[https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_symbol\\_table.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm)
- And this makes perfect sense because if you look at the graphic found on the same page, the symbol table also holds information on the scope of the variables. So no matter what scope we have, the symbol table will know exactly where the scope is and what the reference environment is. This would not change given the type of scope. The symbol table knows everything about a variable



5. (10 points) Detail how you would go about adding reserved words into the problem where you are designing your own lexical analyzer? How would you have to change your code? What would you have to add to let users choose a reserve word word as an identifier?

- If I wanted to add reserved words to my lexical analyzer from problem 1 the way I would go about is that before passing in my word into all of the functions, I would first check to see if the word is in a dictionary or hash-map of reserved words. If it is then we do not pass that word into all of the functions.
- This way we very quickly and easily find if a word is a reserved word and that way it is also not mistaken as some other type of word
- This also means that we can very easily just add words to our reserved words dictionary if we ever need to
- And if we wanted to let users use a reserved word as an identifier than I would put an if statement in my code from question 1 to be able to identify that this reservedWord with a '\*' attached to the end will not be used as a reserved word and instead will be used as an identifier.
- Something like this, this is far from done but you get the gist of it:

```

1 | incomingWord = 'string*'
2 | reserved_words = {
3 |     'string':'string',
4 | }
5 | if(incomingWord[-1] == '*'):
6 |     lexicalAnalyzer(incomingWord[:-1])
7 | else:
8 |     # check if word is in reserved_words
9 |     # if so then it doesn't go to lexical analyzer

```

6. (20 points) Write a recursive decent algorithm for a java while statement, a Java if statement, an logical/mathematical expression based on the rules you created in your lexical analyzer, and an mathematical assignment statement, where statement may be an empty function. Supply the EBNF rule for each.

- I looked at the [java docs](#) to base my answer off of. It kind of was helpful but its kind of weird because its very 'generic'. Like when it calls for a statement it just calls for <statement> but in the bottom it says that the statement must be type boolean. So like why not have <booleanStatement>. You are writing and maintaining one of the most used languages why not make it easier on everyone who is looking at your EBNF/CFG
- The EBNF for a while statement is something like this:
  - <whileStatement> := "while" + "(" + <booleanTypeExpression> + ")" [<statement>]
  - You would have the statement in [] because the statement following the while is optional because if you look at the java docs you can see that that statement can be a "StatementWithoutTrailingSubstatement" which can then be an "EmptyStatement"
  - What this means is that the statement can be nothing so for us it means it can be optional

**Statement:**

[StatementWithoutTrailingSubstatement](#)  
[LabeledStatement](#)  
[IfThenStatement](#)  
[IfThenElseStatement](#)  
[WhileStatement](#)  
[ForStatement](#)

**StatementNoShortIf:**

[StatementWithoutTrailingSubstatement](#)  
[LabeledStatementNoShortIf](#)  
[IfThenElseStatementNoShortIf](#)  
[WhileStatementNoShortIf](#)  
[ForStatementNoShortIf](#)

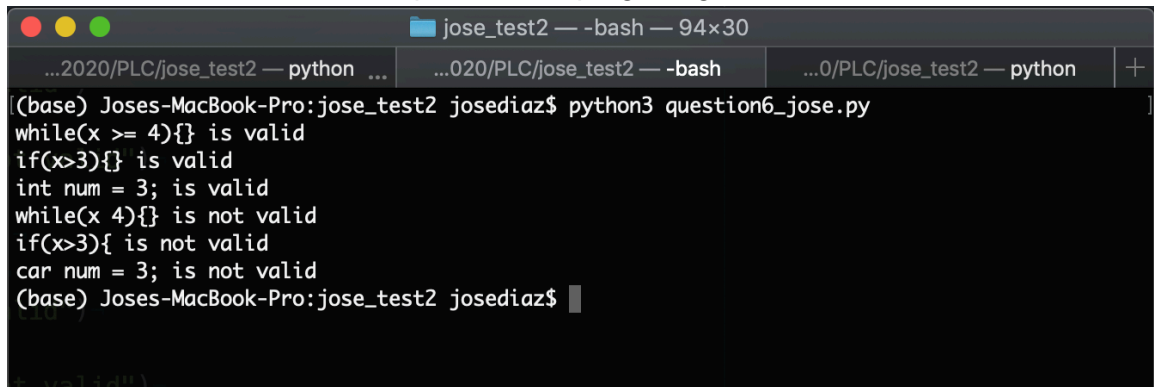
**StatementWithoutTrailingSubstatement:**

[Block](#)  
[EmptyStatement](#)  
[ExpressionStatement](#)  
[AssertStatement](#)  
[SwitchStatement](#)  
[DoStatement](#)  
[BreakStatement](#)  
[ContinueStatement](#)  
[ReturnStatement](#)  
[SynchronizedStatement](#)  
[ThrowStatement](#)  
[TryStatement](#)  
[YieldStatement](#)

- - The EBNF for an if statement is something like this:
    - `<ifStatement> := "if" + "(" + <booleanTypeExpression> + ")" [ <statement> ]`
    - The same reasoning as with the while, the docs say that the `<statement>` can be

an empty statement meaning that for us the statement can be optional

- The EBNF for a logical/mathematical expression and assignment looks something like this:
- I put them both together because they are both statements, just slightly different
- if we base it off of how I did number one then we can only do the basic operators and assigning one variable to another
  - `<Statement> := <var> <op> [<var>]`
  - `<op> := ['/' '*' '%' '+' '-' '=']`
  - `<var> := <id>`
  - `<id> := <letter>{0-9}`
  - `<letter> := [A-Z | a-z]{<\letter>}`
- Obviously if the operator is an equal sign then we will be doing an assignment statement but just to clarify.
- For this problem I did it very similarly to how I did question 1
- Here is a screenshot of the output that this program gives



```
(base) Joses-MacBook-Pro:jose_test2 josediaz$ python3 question6_jose.py
while(x >= 4){} is valid
if(x>3){} is valid
int num = 3; is valid
while(x 4){} is not valid
if(x>3){ is not valid
car num = 3; is not valid
(base) Joses-MacBook-Pro:jose_test2 josediaz$
```

- `<valid>`

```
1
2 types = {
3     'byte': 'byte',
4     'short': 'short',
5     'int': 'int',
6     'long': 'long',
7     'float': 'float',
8     'double': 'double',
9     'boolean': 'boolean',
10    'char': 'char',
11 }
12
13 def startChecker(arr):
14     haveWhile = False
15     haveIf = False
```



```

16     # print(arr[:2])
17     # print(arr[:5])
18     if(arr[:5] == 'while'):
19         # print('we have while at the start')
20         haveWhile = True
21         return True and checkBool(arr[5:])
22     elif(arr[:2] == 'if'):
23         # print('we have an if')
24         haveIf = True
25         return True and checkBool(arr[2:])
26     else:
27         # lets find if we have an =
28         equalSign = arr.find('=')
29         if(equalSign != -1):
30             # need to see if we have variable type and a value and
31             return True and checkVar(arr, equalSign)
32         else:
33             return False
34     return False
35
36 def checkBool(end):
37     parenCount = 0
38     curlyCount = 0
39     startBool = False
40     totalBool = False
41     for char in end:
42         if(char == '('):
43             parenCount += 1
44         elif(char == ')'):
45             parenCount -= 1
46         elif(char == '{'):
47             curlyCount += 1
48         elif(char == '}'):
49             curlyCount -= 1
50         elif(char == '>' or char == '<' or char == '='):
51             startBool = True
52         if(startBool and char == '='):
53             totalBool = True
54             startBool = False
55     return (startBool or totalBool) and (curlyCount == 0) and (parenCount == 0)
56
57 def checkVar(arr, equalSign):
58     done = False
59     #print(arr[equalSign+1: ])
60     wordsBefore = arr[:equalSign].split()

```

```

61     #print(wordsBefore)
62     if(wordsBefore[0] in types.values()):
63         for char in arr[equalSign+1:]:
64             #print(char)
65             if(done):
66                 return False
67             if(char == ';'):
68                 done = True
69             elif(char == '='):
70                 return False
71             else:
72                 pass
73     if(done):
74         return True
75     return False
76
77 def main():
78     validStrings = ['while(x >= 4){}', 'if(x>3){}', 'int num = 3;']
79     nonvalidStrings = ['while(x 4){}', 'if(x>3){', 'car num = 3;']
80
81     for string in validStrings:
82         if(startChecker(string)):
83             print(string + " is valid")
84         else:
85             print(string + " is not valid")
86
87     for string in nonvalidStrings:
88         if(startChecker(string)):
89             print(string + " is valid")
90         else:
91             print(string + " is not valid")
92
93 if __name__ == '__main__':
94     main()

```

7. (10 points) Given the natural constraints of an RDA explain how you would go about the creation of a Statement function in your RDA that would allow statement to either be a while statement, an if statement or an assignment statement.

- So to check to see if a statement is either a while, if, or an assignment you would have to check for each case.
- So first if it is a while we could check for a 'while' keyword, if we find it then we know it is a while and we must have some sort of '(' + + ')'
- Then if we do not find a 'while' keyword we would check to see if we have an 'if'

keyword. If we do, then again we would check to have '(' + + ')'

- And then if we want to know if it is an assignment statement then we must assume that none of the keywords were found and instead we have a variable name that is valid, not a float or a reserved word or anything like that, followed by a '=' and then a valid value. This is so that we do not have something like 'var = @##\$' which would be totally incorrect
- And eventually if we want to know what type of variable it is we would take whatever value it has and run it through our lexical analyzer

8. (10 points) Perl allows both static and a kind of dynamic scoping. Write a Perl program that uses both and clearly shows the difference in effect of the two. Explain clearly the difference between the dynamic scoping described in this chapter and that implemented in Perl.

- In my program you can see that when we dynamically scope we are grabbing the value from the closest place the variable was declared.
- When we statically scope we are actually grabbing the value from the parent function
- In my program I also explain in the comments

```

1
2 # so from geeks for geeks in order to get dynamically scoped variable
3 # my keyword defines a statically scoped local variable
4 # and local defines dynamically scoped local variable
5 #
6 # link: https://www.geeksforgeeks.org/static-and-dynamic-scoping/
7
8 print "Hello World!\n";
9 $mainVar = 100;
10 $mainVar2 = 100;
11
12 sub showVar1
13 {
14     return $mainVar;
15 }
16 sub dynamic
17 {
18     # use local which gives us dynamically scoped var
19     # here since we are dynamic we are grabbing from the closest place
20     local $mainVar = 1;
21     return showVar1();
22 }
23
24 sub showVar2
25 {
26     return $mainVar2;
27 }
28 sub static
29 {
30     # use my which gives us the statically scoped var
31     # here instead of grabbing what is closest to us we are grabbing from the
32     my $mainVar2 = 1;
33     return showVar2();
34 }
35
36 print dynamic()." ----- DYNAMIC\n";
37 print static()." ----- STATIC\n";
38 print "---- Jose Diaz ----";

```

- Output is also in txt file and here

```
1 | Hello World!  
2 | 1 ----- DYNAMIC  
3 | 100 ----- STATIC  
4 | ---- Jose Diaz ----
```