Lab - 01

Name : Ambalia Harshit
Roll No : - -
Subject : ACT(Advanced Computer Technology)
Date :

AIM : Convert suffix code to 3 address code.

```
# Shunting Yard algorithm, which is used to convert infix expressions to #
address code.
def infix_to_suffix(infix):
   # suffix and postfix is same.
   stack = ['#']
   answer = []
   precidence = {'#':0, '(':1, '+':2, '-':2, '*':3, '/':3, '^':4}
   alphabets = ['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']
   for j in infix:
       if(j=='('):
           stack.append(j)
       elif(j in alphabets):
           answer.append(j)
       elif(j==')'):
           while(stack[-1]!='('):
               p = stack.pop()
               answer.append(p)
           stack.pop()
       else:
           while(precidence[j]<=precidence[stack[-1]] and stack[-1]!='#'):
               p = stack.pop()
               answer.append(p)
           stack.append(j)
   while(stack):
       p = stack.pop()
       answer.append(p)
   answer.pop()
   return answer
```

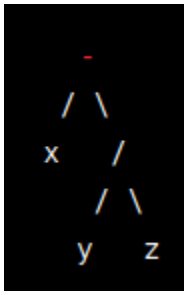
```
def get_operations(suffix):
   alphabetCap = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
   alphabets = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
   stak = []
   answer = []
   # assumptions = []
   for j in suffix:
       if(j in alphabets):
           stak.append(j)
       else:
           tmp1 = stak.pop()
           tmp2 = stak.pop()
           answer.append([j, tmp2, tmp1])
           temp = alphabetCap[0]
           alphabetCap.remove(temp)
           # assumptions.append([temp, answer[-1]])
           stak.append(temp)
   return answer
   # assumptions.pop()
   # print(assumptions)

def main():
   infix = list(input())
   suffix = infix_to_suffix(infix)
   print(get_operations(suffix))

if __name__ == "__main__":
   main()
```

Explanation :

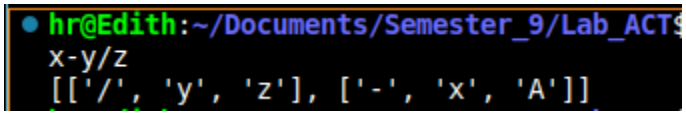
For the input : x - y / z, we can create the following graph/tree.



How are we gonna evaluate this tree/graph?

- After checking the precedence of operations, we can conclude that at first we have to perform (y / z). After will perform (-) with the previous result and (x).
- Will evaluate the graph from bottom-Up manner.

We can generate the computation tree like given, Using Code given. Where we first convert an infix equation to postfix/suffix equation, which will later be used to generate operations in sequence.



So here,

- We'll **first perform ['/', 'y', 'z']** and will **assign it to variable 'A'.**
- After that whatever value of 'A' we get, will **perform the second operation ['-', 'x', 'A']**
- Which will be the ultimate output for the given equation.
- So the generated output works the same as the graph we are getting.

Screen Shots :