```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov  4 20:38:37 2023

@author: LAKSHMIPRIYA Anil
"""

# ============================================================================
# PA - 3
#
# Building Expression Tree
# ============================================================================

import helper
import operator as op
from graphviz import Digraph
import matplotlib.pyplot as plt
from PIL import Image
import io


# Dictionary storing the precedence values of operators
opr_precedence_dict = {'^': 3,
                       '*': 2,
                       '/': 2,
                       '%': 2,
                       '+': 1,
                       '-': 1}


# Function to check if the given symbol is a valid operand
def is_operand(token_symbol):
    cond_1 = ord('a') <= ord(token_symbol) <= ord('z')
    cond_2 = ord('A') <= ord(token_symbol) <= ord('Z')
    cond_3 = ord('0') <= ord(token_symbol) <= ord('9')
    if cond_1 or cond_2 or cond_3:
        return True
    return False


# Function to check if the given symbol is a valid operator
def is_operator(token_symbol):
    if token_symbol in {'^', '*', '/', '%', '+', '-'}:
        return True
    return False


# Function to check if the association rule for the given symbol is left-to-right
def is_left_to_right_associative(operator):
    if operator != '^':
        return True
    else:
        return False


# Function to check if the precedence of the first operator is greater than the second operator
def is_precedence_greater(operator1, operator2):
    # To Debug
    # print(f'Inside greater - top => {operator1}:', opr_precedence_dict.get(operator1), f'\tcurr => {operator2}:', opr_precedence_dict.get(operator2
    if opr_precedence_dict.get(operator1) > opr_precedence_dict.get(operator2):
        return True
    else:
        return False


# Function to check if the precedence of two operators are equal
def is_precedence_equal(operator1, operator2):
    # To Debug
    # print(f'Inside equal - top => {operator1}:', opr_precedence_dict.get(operator1), f'\tcurr => {operator2}:', opr_precedence_dict.get(operator2))
    if opr_precedence_dict.get(operator1) == opr_precedence_dict.get(operator2):
        return True
    else:
        return False



def build_expression_tree(expression):
    '''
    Implement this function to build an expression tree from the infix expression
    Return the root of the expression tree created
```

```
'''

'''
Parameters
----------
expression : A string
    Represents the infix expression.

Returns
-------
root : An instance of class Node
    Represents the root of the expression tree.

Strategy
-------
  Convert the infix expression into an expression tree directly using two stacks
    - Character stack stores operators and opening barackets
    - Node stack stores the operands and the updated root of the expression tree during its construction

'''

token = expression.replace(' ', '')     # Remove spaces from input infix expression
token_len = len(token)

is_valid_token = True
parenthesis_cnt = 0                     # Count variable to keep track of brackets

# Character stack to store non-operands
C_stack = helper.LinkedListStack()

# Node stack to store operands and components of expression tree
N_stack = helper.LinkedListStack()

for char_index, char in enumerate(token):
    try:
        if is_operand(char):            # Check if the character is an operand
            # Verify the syntax to check if there are no consecutive operands
            if char_index < token_len - 1:
                if is_operand(token[char_index + 1]):
                    raise Exception('Exception raised - Invalid infix syntax - Repeated operands')

            '''
            Rule - If an operand is encountered, push it onto the Node stack.
            '''
            operand_node = char
            N_stack.push(operand_node)

            # To Debug
            '''print('\ninput = ', char)
            print('C_stack = ')
            C_stack.printStack()
            print('\nN_stack = ')
            N_stack.printStack()
            print('\n-----------------------------------------------------')'''

        elif char == '(':
            '''
            If the incoming symbol is '(', then push it onto the Character stack.
            '''
            C_stack.push(char)
            parenthesis_cnt += 1

            # To Debug
            '''print('\ninput = ', char)
            print('C_stack = ')
            C_stack.printStack()
            print('\nN_stack = ')
            N_stack.printStack()
            print('\n-----------------------------------------------------')'''


        elif char == ')':                   # ((2 + 3) + 4))   ; 2 + 3) + 4
            '''
            Rule - If the incoming symbol is ')', then
                    i) Pop the operator from the Character stack
                    ii) Pop the topmost element from Node stack as the first operand
                    iii) Pop the new topmost element from the Node stack as the second operand.
                    iv) Assign the first operand as the right child of the operator node.
                    v) Assign the second operand as the left child of the operator node.
                    vi) Push the operator node onto the Node stack
                    vii) Repeat till the Character stack becomes empty or '(' is encountered.
            '''
```

```
        while not C_stack.isEmpty() and  C_stack.top() != '(' :
            popped_operator = C_stack.pop()
            operand_1 = N_stack.pop()
            operand_2 = N_stack.pop()

            popped_operator.right = operand_1
            popped_operator.left = operand_2

            N_stack.push_node(popped_operator)

        # Raise exception since '(' was not encountered
        if C_stack.isEmpty():
            raise Exception('Exception raised - Invalid infix syntax - Unbalanced parenthesis - missing opening bracket')
        else:
            # Removing the opening bracket from the stack
            opening_parenthesis = C_stack.pop()
            parenthesis_cnt -= 1

            # To Debug
            '''print('\ninput = ', char)
            print('C_stack = ')
            C_stack.printStack()
            print('\nN_stack = ')
            N_stack.printStack()
            print('\n----------------------------------------------------')'''


    elif is_operator(char):            # Check if the character is an operator

    # Verify that the first symbol in infix expression is not an operator.
        if char_index == 0:
            raise Exception('Exception raised - Invalid infix syntax - Expression cannot start with operator')

        # Verify that the last symbol in infix expression is not an operator.
        if char_index == token_len - 1:
            raise Exception('Exception raised - Invalid infix syntax - Expression cannot end with operator')

        # Verify the syntax to confirm that there are no consecutive operators.
        if char_index < len(token) - 1:
            if is_operator(token[char_index + 1]):
                raise Exception('Exception raised - Invalid infix syntax - Repeated operators')

        while not C_stack.isEmpty():
            '''
            Rule - If the top of the Character stack is an opening bracket,
                    then push the operator onto the Character stack stack.
            '''
            if C_stack.top() == '(':
                break       # The code for pushing the operator is outside the while loop


            '''
            Rule -

            If operator at the top of Character stack has a greater precedence than the current operator, then
                    i) Pop the operator from the Character stack
                    ii) Pop the topmost element from Node stack as the first operand
                    iii) Pop the new topmost element from the Node stack as the second operand.
                    iv) Assign the first operand as the right child of the operator node.
                    v) Assign the second operand as the left child of the operator node.
                    vi) Push the operator node onto the Node stack
                    vii) Check for precedence again for (current operator, new operator at the top of the Character stack).
                    viii) If the new operator at the top of Character stack still has a greater precedence than the current operator,
                            viii.a) Go to step i)
            '''
            condition1 = is_precedence_greater(C_stack.top(), char)

            # ------------------------------

            '''
            Rule -

            1) If the current operator and the operator at top of the Character stack have the same precedence,
                then check their associativity.

            2) If the associativity of the operators is left to right, then
                    i) Pop the operator from the Character stack
                    ii) Pop the topmost element from Node stack as the first operand
                    iii) Pop the new topmost element from the Node stack as the second operand.
                    iv) Assign the first operand as the right child of the operator node.
                    v) Assign the second operand as the left child of the operator node.
```

```
                        vi) Push the operator node onto the Node stack
                        vii) Go to step 1)
                '''

                condition2 = is_precedence_equal(C_stack.top(), char)
                condition3 = is_left_to_right_associative(char)

                if (condition1 or (condition2 and condition3)):
                    #print('HERE1')
                    popped_operator = C_stack.pop()
                    operand_1 = N_stack.pop()
                    operand_2 = N_stack.pop()

                    popped_operator.right = operand_1
                    popped_operator.left = operand_2

                    N_stack.push_node(popped_operator)
                else:
                    #print('Here2 - precedence lower or right-associative')
                    break

            # Outside the while loop
            '''
            Rule -

            If the Character stack is empty, then push the current operator onto it.

            (OR)

            If operator at the top of the Character stack has a lesser precedence than the current operator,
                then push the current operator onto the Character stack.

            (OR)

            If the associativity of the operators is right to Left,
                then simply push the  operator onto the Character stack.

            (OR)

            For cases where either the greater or equal precedence conditions were satisfied,
                push the current operator onto the Character stack after perfrming the necessary steps corresponding to the codition.
            '''

            C_stack.push(char)
            # To Debug
            '''print('Here3')
            print('\ninput = ', char)
            print('C_stack = ')
            C_stack.printStack()
            print('\nN_stack = ')
            N_stack.printStack()
            print(f'\nN_stack top() = {N_stack.top()}')
            print('\n-----------------------------------------------------')'''

        else:
            raise Exception('Exception raised - Invalid symbol in infix token')

    except Exception as e:
        print(e)
        is_valid_token = False
        break


# Check if any '(' were left unbalanced
if is_valid_token and parenthesis_cnt != 0:
    try:
        raise Exception('Exception raised - Invalid infix syntax - Unbalanced parenthesis - missing closing bracket')
    except Exception as e:
        print(e)
        is_valid_token = False


if is_valid_token:
    # Popping the remaining operators from the Character stack
    #print('\n\nHere4')
    while not C_stack.isEmpty():
        popped_operator = C_stack.pop()
        operand_1 = N_stack.pop()
        operand_2 = N_stack.pop()

        popped_operator.right = operand_1
        popped_operator.left = operand_2
```

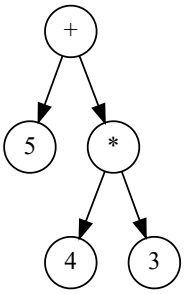
```
            N_stack.push_node(popped_operator)

            # To debug
            '''print(f'\n opeartor =  {popped_operator.val}, operand_1 = {operand_1.val}, operand_2 = {operand_2.val}')
            print('C_stack = ')
            C_stack.printStack()
            print('\nN_stack = ')
            N_stack.printStack()
            dot = visualize(N_stack.top_node())
            display(dot)
            print('\n----------------------------------------------------')'''

        root = N_stack.pop()    # Pop the root of the expression tree.

        C_stack.push(char)
        # To Debug
        '''print('\nC_stack = ')
        C_stack.printStack()
        print('\nN_stack = ')
        N_stack.printStack()
        print('\n----------------------------------------------------')'''
        return root
    return None




# Dictionary to store the arithmetic operations corresponding to the string operators
operators = {
    '+' : op.add,
    '-' : op.sub,
    '*' : op.mul,
    '/' : op.truediv,
    '%' : op.mod,
    '^' : op.pow,
}


def evaluate_expression_tree(root):
    '''
    Implement this function to evaluate the expression tree
    Takes the root of the expression tree as the input and returns the result
    '''


    '''
    Parameters
    ----------
    root : An instance of class Node
        Represents the root of the expression tree.

    Returns
    -------
    int/double
        The evaluated value of the expression tree.

    Strategy
    -------
      Inorder traversal of the expression tree
        - If an operand is encountered, return its value
        - If an operator is encountered,
            - Solve the left subtree of the operator
            - Solve the right subtree of the operator
            - Apply the operator to the result of the evaluation of the left and right subtrees
            - Return the final evaluated result
    '''

    # If the tree for which the root is given is empty, then return None
    if root is None:
        return None

    # If the current node is a leaf node (operand)
    if root.left is None and root.right is None:
        return int(root.val)

    # Recursively evaluate the left subtree
    left_sum = evaluate_expression_tree(root.left)

    # Recursively evaluate the right subtree
    right_sum = evaluate_expression_tree(root.right)
```

```
    if not left_sum or not right_sum:
        return None
    else:
        try:
            result = operators[root.val](left_sum, right_sum)
            return result
        except ZeroDivisionError as ze:
            print('\nException Caught - ZeroDivisionError: ', ze)
            return None


def visualize(root, node = None):
        # Visualize the tree using graphviz

        # Recursively add nodes and edges
        def add_nodes_edges(root, dot = None):
            col = "black"
            if (node != None and root.val == node):
                    col = "green"
            if dot is None:
                dot = Digraph()   # Create Graphviz Digraph

                dot.node(name=str(root), label=str(root.val),
                         color = col, shape="circle",
                         fixedsize="True", width="0.4")
            col = "black"

            # Add nodes recursively
            if root.left:
                if (node != None and root.left.val == node):
                    col = "green"
                dot.node(name=str(root.left), label=str(root.left.val),
                     color = col, shape="circle",
                     fixedsize="True", width="0.4")
                dot.edge(str(root), str(root.left))
                dot = add_nodes_edges(root.left, dot=dot)

            if root.right:
                if (node != None and root.right.val == node):
                    col = "red"
                dot.node(name=str(root.right), label=str(root.right.val),
                     color = col, shape="circle",
                     fixedsize="True", width="0.4")
                dot.edge(str(root), str(root.right))
                dot = add_nodes_edges(root.right, dot=dot)

            return dot
        return add_nodes_edges(root)



def show_expression_tree2(root, filename, dot_list):
  if root != None:
      print('Root:', root.val)
      dot3 = visualize(root)
      dot3.render(filename, format='png')
      dot_list.append(f'{filename}.png')
  return dot_list

def show_expression_tree(root, dot_list):
  if root != None:
      print('Root:', root.val)
      dot = visualize(root)
      dot_list.append(dot)
  return dot_list

def auto_subplot_layout(total_subplots):
    if total_subplots <= 0:
        return 1, 1
    elif total_subplots == 1:
        return 1, 1
    elif total_subplots == 2:
        return 1, 2
    else:
        # Arrange in a square grid
        rows = int(total_subplots**0.5)
        cols = (total_subplots + rows - 1) // rows
        return rows, cols


if __name__ == "__main__":
```

```
# ============================================================================
#     Given Test Case
# ============================================================================

    infix_expression = '5+4*3'
    root = build_expression_tree(infix_expression)
    result = evaluate_expression_tree(root)
    print('Root: ', root.val, '\tResult: ', result)
    dot = visualize(root)
    display(dot)


    print('-----------------------------------------------------\n')

# ============================================================================
#     Custom Test Cases
# ============================================================================
    dot_list = []
    res_list = []
    inp_list = []

    infix_expression1 = '8-4/2*3+1'
    root1 = build_expression_tree(infix_expression1)
    dot_list = show_expression_tree(root1, dot_list)
    result1 = evaluate_expression_tree(root1)
    print("Result:", result1)
    if result1:
      res_list.append(result1)
      inp_list.append(infix_expression1)

    print('-----------------------------------------------------')

    infix_expression2 = '4 ^ 3 / 5 * 6 + 2'
    root2 = build_expression_tree(infix_expression2)
    show_expression_tree(root2, dot_list)
    result2 = evaluate_expression_tree(root2)
    print("Result:", result2)
    if result2:
      res_list.append(result2)
      inp_list.append(infix_expression2)

    print('-----------------------------------------------------')

    infix_expression3 = '6+1-9/3+2^4^2'
    root3 = build_expression_tree(infix_expression3)
    dot_list = show_expression_tree(root3, dot_list)
    result3 = evaluate_expression_tree(root3)
    print("Result:", result3)
    if result3:
      res_list.append(result3)
      inp_list.append(infix_expression3)

    print('-----------------------------------------------------')

    infix_expression4 = '2 + 3 @ 4'
    root4 = build_expression_tree(infix_expression4)
    dot_list = show_expression_tree(root4, dot_list)
    result4 = evaluate_expression_tree(root4)
    print("Result:", result4)
    if result4:
      res_list.append(result4)
      inp_list.append(infix_expression4)

    print('-----------------------------------------------------')

    infix_expression5 = '((5 % 3) + 4'
    root5 = build_expression_tree(infix_expression5)
    dot_list = show_expression_tree(root5, dot_list)
    result5 = evaluate_expression_tree(root5)
    print("Result:", result5)
    if result5:
      res_list.append(result5)
      inp_list.append(infix_expression5)

    print('-----------------------------------------------------')

    # Set up the subplots
    n = len(dot_list)
    fig, axes = plt.subplots(1, n, figsize=(n * 5, 5))  # You can adjust the figure size

    # Plot each Digraph in a subplot
```

```
# Plot each Digraph in a subplot
for i, digraph in enumerate(dot_list):
    fig.suptitle = 'Subplots of Expression Trees'
    ax = axes[i]
    ax.set_title(f'{inp_list[i]}\n eval = {res_list[i]}')
    ax.axis('off')  # Hide axis

    # Render Digraph to PNG and display it in the subplot
    img_data = digraph.pipe(format='png')
    img = Image.open(io.BytesIO(img_data))
    ax.imshow(img)

plt.tight_layout()
plt.show()
```

```
Root:  +        Result:  17
```




```
-------------------------------------------------------

Root: +
Result: 3.0
-------------------------------------------------------
Root: +
Result: 78.80000000000001
-------------------------------------------------------
Root: +
Result: 65540.0
-------------------------------------------------------
Exception raised - Invalid symbol in infix token
Result: None
-------------------------------------------------------
Exception raised - Invalid infix syntax - Unbalanced parenthesis - missing closing bracket
Result: None
-------------------------------------------------------
```