Lab 4 Report Template

1. Lab exercises

3.1 Lab1.E1

3.1.1 describe how you think to solve the problem.

I firstly to contribute a python class named Stack and write relevant methods in this class. Then I write a method to transfer the infix expression to postfix, which use tow stack to store characters. Also, there is a method to check the parentheses matching of the infix expression. The last step, I written a method to calculate the result of the postfix expression.

3.1.2 main code of program, include annotations.

The main algorithm could be a function, with input and output arguments, and the arguments should be clearly explained.

```python
class Stack(object):
    # 初始化栈为空列表
    def __init__(self):
        self.__items = []

    def __str__(self):
        return str(self.__items)

    # 判断栈是否为空，返回布尔值
    def is_empty(self):
        return self.__items == []

    # 返回栈的大小
    def size(self):
        return len(self.__items)

    # 返回栈顶元素
    def top(self):
        return self.__items[self.size()-1]

    # 把新的元素堆进栈里面（压栈）
    def push(self, item):
        self.__items.append(item)

    # 把栈顶元素丢出去（出栈……）
    def pop(self,):
        return self.__items.pop()
```

```python
def parentheses_match(exp):
    exper = exp.split()
    # print(exper)
    stack1 = Stack()
    stack2 = Stack()
    # 创建两个栈
    for dd in exper:
        if dd == "(":
            stack1.push(dd)
            # 如果表达式里面有(，则入栈1
        if dd == ")":
            if stack1.size() == 0:
                return False
                # 如果表达式里面有)，但栈1里面没有(，则返回错误
            else:
                stack2.push(dd)
                if stack1.top() == "(":
                    stack1.pop()
                    stack2.pop()
                # 如果表达式里有),且栈1里面有(,则都入栈再出栈
    if stack1.size() == 0 and stack2.size() == 0:
        # 当且仅当两个栈最后都为空时，返回 真
        return True
    else:
        return False
```

```python
def infix_to_postfix(exp):
    exper = exp.split()
    stack1 = Stack()
    stack2 = Stack()
    for dd in exper:
        if dd not in ["/", "+", "-", "*", "%", "**", "(", ")"]:
            # 如果字符不是运算符，则入栈2
            stack2.push(dd)
        if dd in["/", "+", "-", "*", "%", "**", "(", ")"]:
            if stack1.is_empty() or dd == "(" or stack1.top() == "(":
                stack1.push(dd)
                # 若字符为括号或者栈1为空或者仅有一个左括号，字符入栈
            elif dd in ["/", "+", "-", "*", "%", "**"]:
                # 若字符为运算符，比较它与栈1顶元素的计算优先级
                if compare(dd, stack1.top()):
                    # 若运算符高级，入栈1
                    stack1.push(dd)
                else:
                    # 若运算符低级，栈1元素出栈推入栈2，直到运算符相对栈1顶元素高级，再将运算符推入栈2
                    p = 1
                    while p == 1:
                        stack2.push(stack1.pop())
                        if compare(dd, stack1.top()):
                            p = 0
                            stack1.push(dd)
            elif dd == ")":
                # 遇到右括号，栈1元素出栈推入栈2，直到遇到左括号，过程中忽略两个括号
                m = 1
                while m == 1:
                    stack2.push(stack1.pop())
                    if stack1.top() == "(":
                        stack1.pop()
                        m = 0
    ll = stack1.size()
    for z in range(0, ll):
        stack2.push(stack1.pop())
    # 最后将 栈1全部推入栈2
    nn = stack2.size()
    string = ""
    for s in range(0, nn):
        string = str(stack2.pop())+" "+string
    # 栈2倒表达成字符串，输出
    return string
```

```python
def score(a):
    # 为运算符赋分
    if a in ["(", ")"]:
        return 0
    if a in ["+", "-"]:
        return 1
    if a in ["/", "*", "%"]:
        return 2
    if a == "**":
        return 3


def compare(a, b):
    # 比较运算符优先级
    sa = score(a)
    sb = score(b)
    return sa > sb
```

```python
def calculate_postfix(post_exp):
    list_exp = post_exp.split()
    # print(list_exp)
    stack = Stack()
    # 创建空栈
    for i in range(0, len(list_exp)):
        # 当遇到计算符
        if list_exp[i] in ["/", "+", "-", "*", "%", "**"]:
            above = stack.pop()
            # 取栈顶元素
            under = stack.pop()
            # 取栈顶第二个元素
            small_exp = under+list_exp[i]+above
            # 获得新栈顶元素的简单表达式
            stack.push(str(eval(small_exp)))
            # 计算新元素并入栈
        else:
            stack.push(list_exp[i])
            # 数字元素入栈
    return stack.top()

    # 返回最后剩下的元素
```

3.1.2 show some screenshot of the running results with test data.

Test:

```
infix_exper = "( 1 + ( 2 + 3 ) * 4 - 5 % 5 + ( 2 + 7 ) ** 2 )"
print(parentheses_match(infix_exper))
postfix_exper = infix_to_postfix(infix_exper)
print(postfix_exper)
print(eval(infix_exper))
print(calculate_postfix(postfix_exper))
```

Output:

```
True
1 2 3 + 4 * + 5 5 % - 2 7 + 2 ** +
102
102
```

The eval method gives the correct result of the index expression.

3.2 Lab2.E2

For some simple index expression like " 1 + 2 * 3 ", I can split the expression into 2 queues to store the number and the operational character alone to change it into the postfix. However, I cannot finish the problems with parentheses. So I think that the queue is unfit for the task to transfer the infix expression into postfit.