Project 4: A Stack Machine to Evaluate Expressions

Problem

A common problem in computer science and programming is to convert information from one format into another format that is more convenient to deal with. One such classical computer science problem is to read infix mathematical expressions, convert them to postfix, and then evaluate the postfix expressions. The most common solutions involve using a stack to help with the conversion.

Another common problem is to read someone else's pseudocode and implement an algorithm to match. You practice that skill with this project as well.

For this project, will write your own Stack ADT, then use the stack to do the conversion based on pseudocode given below. Input will be from a text file, and output will be written to a text file.

You may look at the Runestone and possibly use it—but your code must pass the test cases we give you WITHOUT MODIFICATION—so you have to write your own code to solve the problem.

Stack ADT (stack.py)

You will implement a Stack ADT (class Stack) that supports the following operations:

- `push(item)`: push an item onto the stack. Size increases by 1.
- `pop()`: remove the top item from the stack and return it. Raise an `IndexError` if the stack is empty.
- `top()`: return the item on top of the stack without removing it. Raise an `IndexError` if the stack is empty.
- `size()`: return the number of items on the stack.
- `clear()`: empty the stack.

Pseudocode For Main Program (main.py)

Pseudocode for main program:

1. Open file *data.txt*
2. Read an infix expression from the file
3. Display the infix expression
4. Call function *in2post(expr)* which you write *in2post()* takes an infix expression as an input and returns an equivalent postfix expression as a string. If the expression

is not valid, raise a SyntaxError. If the parameter expr is not a string, raise a ValueError.

5. Display the postfix expression
6. Call function *eval_postfix(expr)* which you write *eval_postfix()* takes a postfix string as input and returns a number. If the expression is not valid, raise a SyntaxError.
7. display the result of *eval_postfix()*

Output must match the format shown in Figure 1 below.

What to Submit

Submit in Canvas as `project4.zip`:

1. `main.py` → your main application code.
2. `stack.py` → your BST implementation that WILL be tested by the test code.

Grading (100 points)

pylint will be run on stack.py. Expected minimum score is 8.5.

Score is the sum of:

- Percentage of test cases passed x 80 points
- min(Coding style score/8.5, 1) x 20 points
- Possible adjustment due to physical inspection of the code, to make sure you actually implemented things as directed.

```
infix: 4
postfix:  4
answer: 4.0

infix: 5  +7
postfix:  5 7 +
answer: 12.0

infix: 7*5
postfix:  7 5 *
answer: 35.0

infix: (5-3)
postfix:  5 3 -
answer: 2.0

infix: 5/5
postfix:  5 5 /
answer: 1.0

infix: 8*5+3
postfix:  8 5 * 3 +
answer: 43.0

infix: 8*(5+3)
postfix:  8 5 3 + *
answer: 64.0

infix: 8+3*5-7
postfix:  8 3 5 * + 7 -
answer: 16.0

infix: (8+3)*(5-6)
postfix:  8 3 + 5 6 - *
answer: -11.0

infix: ((8+3)*(2-7))
postfix:  8 3 + 2 7 - *
answer: -55.0

infix: ((8+3)*2)-7
postfix:  8 3 + 2 * 7 -
answer: 15.0

infix: (8*5)+((3-2)-7*3)
postfix:  8 5 * 3 2 - 7 3 * - +
answer: 20.0

infix: ((8*5+3)-7)-(5*3)
postfix:  8 5 * 3 + 7 - 5 3 * -
answer: 21.0

infix: 7*9+7-5*6+3-4
postfix:  7 9 * 7 + 5 6 * - 3 + 4 -
answer: 39.0
```

Figure 1. Example Program Output

Evaluate a Postfix Expression

1. Initialize a stack
2. If next input is a number:
    Read the next input and push it onto the stack
    else:
        Read the next character, which is an operator symbol
        Use top and pop to get the two numbers off the top of the stack/
    Combine these two numbers with the operation
        Push the result onto the stack
3. Goto #2 while there is more of the expression to read
4. There should be one element on the stack, which is the result. Return this.

Infix to Postfix Pseudocode

1. Initialize stack to hold operation symbols and parenthesis

2. if the next input is a left parenthesis:
    Read the left parenthesis and push it onto the stack
  else if the next input is a number or operand:
    Read the operand (or number) and write it to the output
  else if the next input is an operator:
    while (stack is not empty AND
        stack's top is not left parenthesis AND
        stack's top is an operation with equal or higher precedence than the next input symbol):
        Print the stack's top
        Pop the stack's top
        Push the next operation symbol onto the stack
  else:
    Read and discard the next input symbol (should be a right parenthesis)
    Print the top operation and pop it
    while stack's top is not a left parenthesis:
        Print next symbol on stack and pop stack
    Pop and discard the last left parenthesis
  3. Goto #2 while there is more of the expression to read
  4. Print and pop any remaining operations on the stack
    There should be no remaining left parentheses