COMP 116 Data Structures

Lab #8

In this lab, we will practice using stacks to evaluate mathematical expressions. You can use the stack implementation from the Standard Template Library, you do not have to implement your own.

Before we start, let's examine two forms of notations that can be used to represent mathematical expressions.
The **infix notation** is the notation that you are already familiar with. The operator is between the two operands, there is a priority among the operations to remove ambiguity concerning the order in which the operators should be applied, and parentheses can be used to override these priorities. This notation typically looks as follows:

$$A * (B + C) * D + E$$

Postfix notation eliminates the need for parentheses and there are no levels of precedence to learn between operations. The operator is always placed after the two operands to which it is supposed to apply (these operands could be the result of application of previous operators). For example, the postfix notation of the infix expression above would be:

$$A\ B\ C + *\ D\ *\ E\ +$$

Postfix expressions are evaluated from left to right, resolving each operator using the two values that precede it every time.
Giving the values $A = 5$, $B = 2$, $C = 8$, $D = 3$ and $E = 9$, we could evaluate the expression above as follows:

$$\begin{aligned} 5\ 2\ 8\ + * 3\ *\ 9 + &= 5\ 10\ *\ 3\ *\ 9\ +\ \text{(resolving 2 8 +)} \\ &= 50\ 3\ *\ 9\ +\ \text{(resolving 5 10 *)} \\ &= 150\ 9\ +\ \text{(resolving 50 3 *)} \\ &= 159 \end{aligned}$$

For this lab, you will write two functions, a function called `infix2postfix` that takes a string as input and returns a string, and a function called `postfixEval` that takes a string as input an returns an integer. For simplicity of processing, we will require that the strings containing expressions in infix or postfix notation should be a sequence of "words", where each word is either an integer, a parenthesis (open or close), or one of the four operators $+$, $-$, $*$ or $/$.

The function `infix2postfix` will require a stack as follows:

1. Read the "words" one at a time.

2. If the word is a number, pass it directly to the output.
3. If the word is a left parenthesis, push a left parenthesis to the stack.
4. If the word is a right parenthesis, pop words from the stack and pass them to the output until you pop a left parenthesis (the parentheses should not be passed to the output). If no left parenthesis is found on the stack, throw a runtime error because the infix expression is incorrect.
5. If the word is an operator, pop words from the stack and pass them to the output until the top of the stack is either a left parenthesis or an operator of strictly lower priority. The operator should then be pushed on the stack.
6. If there are no longer any words, pop every operator from the stack and pass them to the output. If you find a left parenthesis left on the stack, throw a runtime error because the infix expression is incorrect.

I will write a few examples of expressions on the board to help you test your program.

________ Show me the result when you are done.

Then, write the function `postfixEval`. You should be able to figure out how to write that one on your own. If you do it without a stack, your solution is probably wrong.
Feel free to ask me to check your algorithm before writing it down.

________ Show me the result when you are done.

When you are done, write your name on the sheet and hand it to the lab instructor.

Name: ____