Postfix Calculator

Introduction

In this lab, the goal is to create a calculator capable of evaluating mathematical expressions. The process consists of two fundamental steps: firstly, converting an infix expression to a post expression, and secondly, evaluating the resultant postfix expression.

In simple terms, an infix expression is the conventional way we write mathematical expressions, where operators like +,-,*, and / are placed between the operands (e.g., 3 + 4 * 5). On the other hand, a postfix expression, also known as Reverse Polish Notation (RPN), places operators after the operands (e.g., 3 4 5 * +). The significant advantage of postfix notation is the elimination of parenthesis and the ambiguity regarding the order of operations, making it a streamlined approach for computer evaluation.

Here's a breakdown of the task at hand:

Converter Class:

- This class is responsible for tacking an infix expression as a string input and converting it to a postfix expression.
- Example: The infix expression is "3 + 4 * 5" should be converted to "3 4 5 * +".

PostfixCalculator Class:

- This class is in charge of evaluating the postfix expression generated by the Converter class.
- Example: Given the postfix expression "3 4 5 * +" , the PostfixCalculator class should evaluate and return the result 23.

For this project, both the **Converter** and **Calculator** classes should utilize the **ArrayStack** implementation to manage operands and operators using the stack data structure (pages 229-230). The use of any pre-built classes from the Java library such as ArrayList is not allowed.

By completing this lab, you will have a working calculator that abides by the principles of data structure, specifically the stack, and understand the mechanics behind expression evaluation in computing.

Implementation and Design

<u>Converter class</u> : This class converts an infix expression to a postfix expression.

1. **Object Instantiation.**
    - Upon instantiation, the **‘Converter’** class accepts a ‘**String**’ representing the user-generated infix expression, which should be stored as an instance variable.
2. **Tokenization.**
    - Initially, tokenize the input string into a list of tokens representing operators, operands, and parentheses using the provided parser method in ‘**ParserHelper.java**’.
3. **[Method] Infix to Postfix Expression Conversion.**
    - Define a method named ‘**toPostFix()**’ which will convert the infix expression to a postfix expression.
    - The resulting postfix expression should be a ‘**String**’, with each operator and operand separated by spaces, ready for evaluation by the ‘**PostfixCalculator**’.
4. **Conversion Algorithm.**
    - Employ a stack to facilitate the conversion process.
    - Create an empty stack and empty output string.
    - Read all tokens from left to right:
        i. For an operand, append it to the output string.
        ii. For an operator, check the top of the stack to ensure lower precedence. If the token has higher precedence, push it onto the stack. If it has lower precedence, pop operators from the stack, appending them to the output string until you encounter a lower precedence operator or the stack is empty.
        iii. For an open parenthesis, push it onto the stack.
        iv. For a closed parenthesis, pop and append all operators from the stack to the output string until an open parenthesis is encountered. Discard both parentheses as postfix expressions don’t require them.

<u>PostfixCalculator class</u> : This class evaluates a postfix expression.

1. **^ Operator.**
    - Utilize ‘**Math.pow(x, y)**’ to evaluate ‘**x^y**’.
2. **[Method] Postfix Expression Evaluation.**
    - In postfix notation, operators are placed after the operands.
    - Multiple operators are evaluated from left to right, each operator being applied to the two preceding operands.

- If you encounter difficulties understanding the evaluation algorithm, refer to the postfix notation definition in the textbook.

The implementation details and any helper methods needed within the classes are at your discretion.

The task entails the analysis and evaluation of mathematical expressions provided as input. These expressions consist of integers and are operated upon by a set of arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). It's a given that the expressions will be well-formed, meaning they will not lack operands for any operator. For instance, an incomplete expression like "3 + " will not be encountered. Below are some exemplary inputs to be used for testing:

- 3+4*5/6
- (300+23)*(43-21)/(84+7)
- (4+8)*(6-5)/((3-2)*(2+2))
- 2^3*(5+(3+2)^2)-40/2+2^2-1

The project does not assume expressions starting with a negative sign or exceptions such as division by zero, but we encourage handling these cases.

Example output

```
type your infix expression: 3+4*5/6
converted to postfix: 3 4 5 * 6 / +
answer is 6.33

type your infix expression: (300+23)*(43-21)/(84+7)
converted to postfix: 300 23 + 43 21 - * 84 7 + /
answer is 78.09

type your infix expression: (4+8)*(6-5)/((3-2)*(2+2))
converted to postfix: 4 8 + 6 5 - * 3 2 - 2 2 + * /
answer is 3.00

type your infix expression: 2^3*(5+(3+2)^2)-40/2+2^2-1
converted to postfix: 2 3 ^ 5 3 2 + 2 ^ + * 40 2 / - 2 2 ^ + 1 -
answer is 223.00
```