CSCI 241, Fall 2012

Assignment #1: Expression Trees (30pts)

Summary

- Goal: This assignment is worth 10% of the grade for the course. In this assignment you will implement some functions for building and traversing trees. Please show some independence in completing this assignment, and continue to follow good design practices!
- Due time: **end of day (11:59pm), Friday, Oct 12.**
- Hand in: submit your file(s) via the **Assignment 1 Submission** link on the Blackboard under "Assignments". A Readme file is not required but you may include one if there's anything you want to tell me.
- Read the following specs carefully!

Introduction

A binary tree provides a natural way to represent arithmetic expressions. Here we consider simple forms of expression trees formed from integers and the 5 binary operators +, -, *, / and ^ (power). The internal nodes of an expression tree contain any one of the binary arithmetic operators. And the leaves of an expression are operands. Go to lecture slides for examples of expression trees.

Input

We will input arithmetic expression as text string. In this assignment we assume the expression input to your program appear as a form of infix notation, fully parenthesized, and provided on a single line. Thus 5+2*4 is not a legal input; it must be written as (5+(2*4)). This saves you from having to deal with operator precedence rules.

Some examples of valid expressions:
25, (15+8), ((7+9)*16), ((45+4)/(3*12)), ((9+(4^28))*(14+13))

A text expression is "fully parenthesized" when each arithmetic operator and its operands are surrounded by their own (unique) pair of parentheses. Spaces in these text expressions do not matter. E.g. (9   +5) is the same as (9+5)

Output (15pts)

1. Turn the input arithmetic expression into an expression tree. Output the prefix, infix, and postfix notations via preorder, inorder and postorder traversal, respectively, on the expression tree.
2. Evaluate the resulting expression tree and output the value.

Sample output for the given expression: ((3+4)*((5^6)/7)) as follows:
Output:

1. Prefix: * + 3 4 / ^ 5 6 7
   Infix: ((3+ 4)*(5^6)/7))
   Postfix: 3 4 + 5 6 ^ 7 / *
2. Value: 15625.000000000002

Implementations (15pts):

1. Start by writing a generic binary tree package (gen_tree.ads, gen_tree.adb) using the linked representation.
2. You should also write a TreeExpression package (tree_expression.ads, tree_expression.adb) that implements the conversion to an expression tree from a given expression, a tree expression evaluation, etc.
3. Resources you can use and modify:
    - Sample codes on http://users.cis.fiu.edu/~weiss/ada.html
    - Cite properly.
4. Comment on your submission properly. Name your files with specified names. Put all the .ads and .adb files involved in your implementation into a zip file and submit a single zip file. Feel free to reuse the stack and queue packages in your implementation.
5. Try to integrate the modularity design in your code.
6. When encountering an invalid expression such as (4 +) or ((4 +3) (2 +5)), you should throw an InvalidExpressionException.

Extra Credit (10 pts)

Fully parenthesized expressions make things easier for you, but harder for the user. The point of this extra credit challenge is to deal with strings that are not necessarily fully parenthesized. As before, you want to convert back and forth between strings and expression trees.

Rename your old TreeExpression package (and its file) to FullParenTreeExpression. Now write a new TreeExpression package. The new package should behave differently in two ways:

1. The new infix() function should convert a tree to a string that has as few parentheses as possible. For example, (new TreeExpression("((5+3)*4)")).infix() should return the string "(5+3)*4, and (new TreeExpression("((5*3)+4)")).infix() should return just 5*3+4, since that has the same meaning, given the usual rules about operator precedence ("order of operations"). Also, ((5-4)-3) should come out as 5-4-3, but (5-(4-3)) shouldn't!
2. Its constructor should be able to create a tree from expression strings even if they are not fully parenthesized. Again, you must follow the usual rules about operator precedence. To find out how to do this, search the web to find a good discussion of “recursive descent”. Some discussions specifically talk about parsing arithmetic expressions, which is an easy case.

If you only handle one of the two bullet points above, you will still get extra credit; more if you handle both.