CSE 326: Data Structures
Assignment #2
April 1, 1996
due: Wednesday, April 10

Programming problem: implement infix-to-postfix and infix-to-prefix converters via expression trees. Here are the details:

1. Implement the abstract data type `BinaryTree` as a `C++` class. Be sure to include the following operations, which you will need to solve part (2):

   (a) `NewTree(info, left, right)`, which takes the data `info` (either an integer or a character) and two `BinaryTrees` `left` and `right`, and returns a new `BinaryTree` whose root has data field `info`, and whose left and right subtrees are `left` and `right`, respectively.

   (b) `Info(T)`, which returns the data field of the `BinaryTree T`.

   (c) `LeftSubtree(T)`, which returns the left subtree of the `BinaryTree T`.

   (d) `RightSubtree(T)`, which returns the right subtree of the `BinaryTree T`.

   (e) `IsEmpty(T)`, which returns `TRUE` if the `BinaryTree T` is empty, and `FALSE` otherwise.

   You may also include any other operations that you think reasonable for this abstract data type or necessary to solve part (2). You may choose any representation of binary trees that you find most effective.

2. A "fully parenthesized formula" is defined as follows:

   (a) Any nonnegative integer is a fully parenthesized formula.

   (b) If A and B are fully parenthesized formulas and $\bullet$ is an element of the set $\{+, -, *\}$, then

   $$(A \bullet B)$$

   is a fully parenthesized formula.

   For example, $(((1+6)*(7-3))+(5*(2+(31-13))))$ is a fully parenthesized formula, but $(1+6*2)$ and $1+(6*2)$ and $6*2)$ are not.

   Design and implement a recursive function `InfixToTree` that reads a fully parenthesized formula and outputs the binary expression tree corresponding to the formula. Then implement two recursive functions `TreeToPrefix` and `TreeToPostfix` that input a binary expression tree and print respectively the prefix and postfix expressions represented by the tree. (A *prefix expression* is either a number, or the concatenation,

in this order, of an operator and two prefix expressions. For example, "$-, 20, +, 2, 3$" is the prefix expression corresponding to the postfix expression "$20, 2, 3, +, -$".)

For all three of these functions, use the `BinaryTree` abstract data type that you implemented in part (1), but be sure that you are only interacting with it via its exported operations. That is, if I substituted some other correct implementation of `BinaryTree` in part (1), your procedures in part (2) should still work.

Finally, put these functions together to produce two functions `InfixToPrefix` and `InfixToPostfix` that do what the names suggest. Test these conversion procedures on the examples given above, as well as other good test examples. Details on how to turn in your program will follow later this week.

Hints: The body of `InfixToTree` mirrors the definition of "fully parenthesized formula" above. In particular, the recursive part does the following, reporting an error if the input is not what it expects:

- Read (.
- Recursively call `InfixToTree` to read the left operand and return the left subtree.
- Read the operator.
- Recursively call `InfixToTree` to read the right operand and return the right subtree.
- Read).

Since `InfixToTree` works recursively, in general it will not consume the entire input formula. How much will it consume? Don't forget to check for the error of having more input than your recursive procedure expects; you may assume that the formula ends with a newline character.

You may want to write a function `ReadLexeme` that reads and returns the next complete "token" (called a "lexeme") from the input stream; this token is either an integer, a character such as '(' or '+', or the newline character. You should ignore spaces and tabs.