Project 5

Expression Tree

Introduction

An expression tree is a binary tree in which each non-leaf node contains a binary operator and each leaf node contains a number or a variable. The tree represents an algebraic expression. At each non-leaf node, the left and right subtrees themselves represent algebraic expressions. The left subtree represents the left operand of the operator at the node and the right subtree represents its right operand. The structure of the tree reflects the order of operation.

The Program Interface

The program begins by presenting the user with a Menu of possible actions.

```
MENU

(I) Enter an expression
(D) Draw expression tree
(E) Evaluate expression
(Q) Quit
```

The user enters one of the letters `I`, `D`, `E`, or `Q` and hits Return. I will provide you with the `Menu` class. It is very easy to use. If the user enters `I`, then the program prompts him to enter an algebraic expression written in infix notation consisting of nonnegative numbers, single-letter variables, the operations of addition, subtraction, multiplication, and division, and left and right parentheses as needed. After this expression has been entered, the user may draw the expression tree, evaluate the expression, enter another expression, or quit the program. If the user enters `E` and if the expression contains any variables, then the user will be prompted to enter values for the variables, after which the value of the expression will be displayed.

The Expression Tree

This program will involve many classes. The expression tree will be derived from the `BinaryTree` class. The nodes will represent the tokens of the algebraic expression, but there are five kinds of tokens in the expression: operators, numbers, variables, left parentheses, and right parentheses. Three of these kinds (operators, numbers, and variables) will be stored in the expression tree. Thus, we will have the `Token` class as a base class for the `Operator` class, the `Number` class, the `Variable` class, the `LParen` class, and the `RParen` class.

The `Token` class and its subclasses are described in the documents `Token Class`, `Operator Class`, `Number Class`, `Variable Class`, `LParen Class`, and `RParen Class`.

The algorithm used to process the expression will require stacks and queues, which, in turn, will require various list classes. In other words, this project will incorporate just about every class that we have developed in this course.

The `ExpressionTree` Class

Although the `ExpressionTree` class is publicly derived from the `BinaryTree` class, it is not a template class. The data stored in its nodes are pointers to `Token`s. Thus, more specifically it is derived from the `BinaryTree<Token*>` class. Its data members and member functions are described in the document `ExpressionTree Class`.

The Algorithm

I have written the functions `toPostfix()` and `getToken()`. The `toPostfix()` function calls `getToken()` repeatedly to read all of the tokens of the infix expression. It returns a queue of `Token` pointers in postfix order, ready to be processed by the function `toExprTree()`.

The function `toExprTree()` should dequeue the `Token` pointers one by one. When a pointer to a `Number` token or a `Variable` token is dequeued, an `ExpressionTree` is constructed that contains only the pointer to that token as its root node. That `ExpressionTree` is pushed onto a stack.

When a pointer to an `Operator` token is dequeued, the top two `ExpressionTree`s are popped off the stack. An `ExpressionTree` is constructed with the pointer to the operator as its root and the two `ExpressionTree`s as its left and right subtrees. This tree is then pushed onto the stack.

After the last token pointer is dequeued and processed in this manner, the stack should contain a single expression tree. It should be popped and assigned to `*this`.

The typeid Operator

C++ provides an operator that can be used to determine the type of object that a pointer points to. If `p` is a pointer to an object, then the expression `typeid(*p)` will return the type

of object that `p` points to. For example, if we want to know whether `p` points to an `int`, we would write the following `if` statement.

```
if (typeid(*p) == typeid(int))
```

You should use the `typeid` operator when you dequeue a token pointer from the queue in order to decide how to process it.

The List of Variables

Variables may appear in any leaf node of the expression tree. To evaluate the expression, we must prompt the user to enter a value for each variable that appears. That means we must know what the variables are. Also, we must be careful not to prompt the user more than once for the same variable. The solution is to build a list of the `Variable`s that appear in the tree. You should traverse the tree, searching for `Variable` nodes. When one is found, check to see whether it is already in the list of variables. If not, then add it to the list. When the traversal is finished, each variable should appear in the list exactly once. Then use the list to prompt the user to enter a value for each variable. When the tree is evaluated, the value of each variable will be substituted for that variable.

When you are finished, turn in *all* of the files necessary to compile your program, include the various list, stack, and queue header files. You work is due by midnight, Monday, May 1.