EECS 268: Spring 2009
Laboratory 9: Binary Trees and Evaluating Simple Algebraic Expressions

Due: 11:59 pm the day before your lab meets the week of May 4th

Lecture Topics: Binary Tree Construction and Traversal, Evaluating Prefix and Postfix Expressions
Lab Topics: Template-based Data Structures, Design Patterns

1. Introduction

In this lab, you will use Binary Trees and Stacks to parse, store, and evaluate simple algebraic expressions involving single-character operands and the four operators: +, -, *, /. You will read the expression in postfix form, and output both the result of the expression, as well as the original expression in prefix, infix, and postfix notation.

2. Getting Started

1. Read the postfix expression, one character at a time. Build the binary tree as you proceed. (The algorithm will be very similar to the stack-based postfix evaluation algorithm we studied earlier. Operands get pushed; operators cause two operands to be popped, and a result pushed back on. You will need to think through exactly what the stack item type should be.)

2. Read the symbol table, one symbol at a time (each symbol includes both the identifier and the dollar amount). Use the map data structure provided in the C++ STL to associate the symbol with its value.

3. If you detect an error in the input postfix expression or symbol table, print as informative of a message as you can and continue to the next transaction.

4. If the expression is valid, traverse the tree you built in (i) preorder, (ii) inorder, and (iii) postorder. Simply output the characters in the correct order. Place them all on one line without spaces between the characters. For inorder, place parentheses around subexpressions to correctly capture evaluation order.

5. Evaluate the tree using the "symbol table" information provided. Print your answer to the output file. To do this, you will need to use a specific traversal scheme along with a "visit" function object that actually evaluates the result. This routine will also require a stack, but the stack item type it needs is different from that of the stack in step #1 above. Hence you will need to make a template-based version of stack code you have written for previous lab assignments.

3. Data Structures and Design Patterns

3.1. Data Structures

Stack: This lab requires the use of a template-based Stack. This will require you to convert the Stack implementation you used in previous labs to a template-based data structure. Unlike the typedef-based Stack used in previous labs, the template-based Stack will enable you to use a single Stack implementation with multiple data types within your program. This Stack will then be used both in the construction of a Binary Tree representing the given algebraic expression, as well as in the evaluation of this expression.

Binary Tree: This lab will also require a Binary Tree. (Note: You will *not* need to provide a template-based implementation for this Binary Tree.) The primary functionality of this Binary Tree can be divided into Construction and Traversal:

- Construction: You will be constructing the Binary Tree in a piecewise fashion, one node at a time, as you process the given algebraic expression. The resulting Binary Tree will therefore be constructed in a bottom-up fashion. This will require your Binary Tree implementation to include the ability to attach an existing Binary Tree as either the left or right child of a new Binary Tree. (An example of this can be seen in the `attachLeftSubtree` and `attachRightSubtree` functions provided in Carrano's Binary Tree implementation.)

- Traversal: Your Binary Tree will need to implement the preorder, inorder, and postorder traversals. These traversals will then be used both for outputting the algebraic expression, as well as for evaluating the expression. In order to re-use the traversal algorithms for both applications, we will be applying the "Visitor" design pattern. More information on the "Visitor" pattern can be found in the next section.

Map: Finally, this lab will require the use of the map data structure provided in the C++ STL. This data structure is used to associate a key with a value, and it will be used in this lab to associate each symbol in the given algebraic expression with its corresponding value.

3.2. Design Patterns

Visitor: The Visitor design pattern allows for the separation of traversal logic and application logic. In other words, the Visitor pattern allows us to write a single traversal algorithm (e.g., preorder, inorder, postorder), and then to re-use that traversal algorithm for many purposes, by passing the "application logic" into the visitor function as either a function pointer or a function object. The Visitor function will then visit each node of the data structure in turn, passing that node to the provided "application logic" via the given function pointer or function object.

Function Object: A function object is like any other C++ object, except that it also overloads the parenthesis operator (), allowing it to be used like a function. For example, if the "Print" class overloaded the parenthesis operator, then every instance of the "Print" would be a function object which could then be called as if it were a function (i.e., `Print p; p();`) Like all overloaded operators, this is equivalent to calling the overloaded parenthesis operator function explicitly (i.e., `Print p; p.operator()();`)

4. Input & Output

Your program will be expected to read input from the first file specified on the command line and to write output to the second file specified on the command line. Your program should display an error message and exit immediately if the expected command-line parameters are not provided. Following is an example of how your program will be run from the command line:

```
$ ./main input.txt output.txt
```

Input will be a series of algebraic expression and symbol table pairs. For each pair, you will output both the result of the algebraic expression, as well as the original expression in prefix, infix, and postfix notation.

4.1. Sample Input

Following is an example of a valid input file:

```
ab+dX-*s/
a 3.25 X 2.00 s 9.99 b 25.90 d 6.00

UU*u-
u 7999.00 U 95.49
```

4.2. Sample Output

Following is an example of the expected output for the previous input file:

```
Prefix: /*+ab-dXs
Infix: (((a+b)*(d-X))/s)
Postfix: ab+dX-*s/
Result: 11.67

Prefix: -*UUu
Infix: ((U*U)-u)
Postfix: UU*u-
Result: 1119.34
```

5. Grading Criteria

20	**Reading the postfix expression and building the binary tree**
10	**Reading the symbol table and using the STL map**
25	**Evaluating the binary tree to get a numeric result**
20	**Display of expressions using tree traversals (including infix parentheses)**
10	**Error detection in prefix expression and symbol table**
15	**Programming style and documentation (internal and javadoc)**
100	**Total**