CSE 373 -- Spring 1999 -- Assignment #4

(*due Friday, April 30*)

1. **Exercise 4.8**

2. **Given an implementation of a general integer tree implementation that uses "first child" and "sibling" pointers, write a function `PrintTreePostorder(IntTree)` that prints out the tree's values in a postorder manner.**

```
typedef struct _IntTreeNode() {
          int data;
          struct _IntTreeNode *sibling;
          struct _IntTreeNode *first_child;
        } IntTreeNode;

typedef IntTreeNode *IntTree;
```

   (*C++ people:* note that C++ doesn't really give you much of an advantage for this problem, so just write a C routine...)

3. **Assume that we have a hash table with 5 slots, and are using the standard integer hash function *hash(x) = x mod tablesize* to store integers in the slots. Draw the table that results from the insertion sequence: 5923, 7222, 6108, 4917, 1234.**
   (a) Using separate chaining (inserting at the end of the list)
   (b) Using linear probing
   (c) Using quadratic probing
   (d) Rehashing your answer to part (b) once the load factor reaches 1.0 (grow the table to the next prime number greater than twice the current table size and rehash the numbers in the order they're stored in the table)

4. **In lecture we discussed using a hash table to store a set. Assume that I have two such hash table sets of integers, each the same size, both of which use the same hash function. Both of the hash tables use separate chaining to resolve collisions. Describe (in words, not code) a linear-time algorithm for creating a new hash table that is the union of the two sets. Then describe how you would find the intersection of the two sets.**

5. **Programming Assignment (due Monday, May 10)**

   The Goal: This program is designed to give you experience using hash tables and trees by implementing yet another realistic program. The book's ADT code will need slightly more modification than on the previous programming assignment, so be sure to start early.

   The Assignment: In this assignment, you will implement an interpreter for a very simple language named *Binary Expression Language*, or BEL++ for short (the "++" is just so we can charge our customers more money). An interpreted language is one in which the user types expressions and the language processes them one by one. This is in contrast with a compiled language (such as C or C++) in which source code is fed into a compiler in order to produce an executable that implements the program. Thus your program will read a BEL++ statement from the console, process it, and then read the next BEL++ statement.

BEL++ Syntax: BEL++ allows users to define variables, to print out a variable's definition, and to evaluate variables. Each of these commands is represented by a single character in BEL++: **d**efine, **p**rint, and **e**valuate. There is also a fourth command: **q**uit that allows the user to get out of the interpreter. Each BEL++ statement consists of a single character and its arguments. Syntax is of the following form:

d <*varname*> **=** <*definition*>
p <*varname*>
e <*varname*>
q

A variable's name (*varname*) can be any combination of letters and the underscore symbol. Thus "`x`", "`nextval`", and "`brads_special_variable`" are all legal variable names.

A variable's definition (*definition*) can either be a floating point value, another variable name, or a mathematical expression involving floating point values and other variable names. For simplicity (trust me on this one), the mathematical expressions will be given in prefix notation; even though it makes things harder for our users, it makes things easier for us. The mathematical operators that we'll accept are `+`, `-`, `*`, `/`, and `^` (exponentiation). Thus, statements that set up a few definitions might be:

```
d pi      = 3.14159
d two_pi  = * 2 pi
d x       = 5
d y       = x
d len_sq  = + ^ x 2 ^ y 2
```

Prefix Notation: If you're not familiar with prefix notation, it places the arithmetic operator first, and follows it with its two arguments (where the arguments may themselves be prefix expressions). Thus, the definition of `two_pi` above is 2*pi, whereas `len_sq` is defined to be (x^2) + (y^2). Note that one nice thing about prefix notation is that no parenthesis are needed to indicate order of evaluation. For example, (x-y)-z would be "- - x y z", whereas x-(y-z) would be "- x - y z". If you think of an expression in tree form, prefix notation corresponds to printing the tree using a preorder traversal (note the subtle foreshadowing...).

BEL++ Semantics: Your program should not evaluate variables when they are defined, but merely store the definition away for later evaluation. Printing a variable will simply print out the same definition that the user provided for it. Evaluating it will have your program compute the value of the variable and print out a single floating point answer. For example, the following commands might be executed after the definitions above:

```
p pi
3.14159
e pi
3.14159
p two_pi
* 2 pi
e two_pi
6.28318
```

```
p len_sq
+ ^ x 2 ^ y 2
e len_sq
50
d x = 2
e len_sq
8
d y = 1
e len_sq
5
```

Note that since we're storing a variable's definition rather than evaluating it when it's defined, we can get different answers if we change the variables that are used in its definition. For example, in the examples above, `len_sq` is defined in terms of `x` and `y`, and `y` is originally defined in terms of `x`. When we first evaluate `len_sq`, it uses the original definitions of `x` and `y` and computes $5^2 + 5^2$. Then, by redefining `x`, the next evaluation of `len_sq` will compute $2^2 + 2^2$, since `x` is now 2 and `y` is still defined to be `x`. Then, we redefine `y` so that it is no longer is defined in terms of `x`, causing the final evaluation of `len_sq` to be $2^2 + 1^2$.

Implementation: Your BEL++ interpreter should store each variable's definition in a binary tree (note -- this is *not* a binary search tree, but simply a tree in which each node has two children). All of the variable definitions should be stored in a hash table so that variables can be looked up easily for evaluation and printing.

At a high level, your program should work as follows. There will be a main loop that reads a command from the console and then evalutes it according to the command type:

- If it's a definition, read the definition into a binary tree (this should be an elegant recursive function). Store the definition using a hash table where the variable name serves as the key. If the variable has already been defined, replace the previous definition with the new one.

- If it's a print request, look the variable up in the hash table and print out its definition using a recursive traversal routine on the binary tree.

- If it's an evaluation request, evaluate the binary tree. This should be an elegant recursive function. If the evaluation of a definition refers to a variable that hasn't been defined, print out an appropriate error message.

- If it's the quit command, exit the program.

This gives you the basic algorithm that your code should use. Details that you will need to figure out will be (i) what you will store in the nodes of your binary tree, (ii) how your hash table will be organized and used, (iii) what operations your binary tree will need to support.

ADTs Weiss' ADTs will not be quite as helpful for this assignment as they were for the previous one. He provides a binary search tree implementation, but since the only thing our tree will share is the binary property, you'll want to get rid of most of its operations and replace them with your own (starting from scratch may be just as easy). His hash table implementations will be a bit more useful, since it has the basic functionality we need and handles conflicts. The main issue you'll need to deal with are the issues we discussed in class Wednesday -- you'll be hashing a string (the variable name), but also need to store a binary tree. Should you send these two things in as a record and have the hash table pick out the name field? Or change the interface to take a (key,data) pair? Or something else entirely?

Helpful C functions C's `atof()` function takes a char * as an argument and returns a double (#include <stdlib.h>). This can be helpful if you read in or store floating point values as strings and need to convert them to their numeric values. C's `pow()` function takes two doubles and returns the result of the first taken to the power of the second as a double (#include <math.h>).

A Word of Advice: At first glance, this program may seem a bit overwhelming. However, it can be implemented using a small number of elegant recursive functions. Spend a fair amount of time sketching out how these functions will work before diving into the implementation -- a bit of planning will save a lot of wrestling with code.

Check the course web for sample input for your program, and further hints for how to implement the program in a modular, straightforward manner.