# Project Two: Recursion

**Updated June 9, 2012. See GREEN HIGHLIGHTED part for updating.**

## Introduction

This project will give you experience writing recursive functions that operate on recursively-defined data structures and mathematical abstractions.

## Lists

A "list" is a sequence of zero or more numbers in no particular order. A list is well formed if:

a) It is the empty list, or
b) It is an integer followed by a well-formed list.

A list is an example of a linear-recursive structure: it is "recursive" because the definition refers to itself. It is "linear" because there is only one such reference.

Here are some examples of well-formed lists:

```
( 1 2 3 4 ) // a list of four elements
( 1 2 4 ) // a list of three elements
( ) // a list of zero elements--the empty list
```

The file recursive.h posted on Sakai defines the type "list_t" and the following operations on lists:

```
bool list_isEmpty(list_t list);
// EFFECTS: returns true if list is empty, false otherwise

list_t list_make();
// EFFECTS: returns an empty list.

list_t list_make(int elt, list_t list);
// EFFECTS: given the list (list) make a new list consisting of
// the new element followed by the elements of the
// original list.

int list_first(list_t list);
// REQUIRES: list is not empty
// EFFECTS: returns the first element of list

list_t list_rest(list_t list);
// REQUIRES: list is not empty
// EFFECTS: returns the list containing all but the first
// element of list

void list_print(list_t list);
// MODIFIES: cout
// EFFECTS: prints list to cout.
```

Note: `list_first` and `list_rest` are both partial functions; their EFFECTS clauses are only valid for nonempty lists. To help you in writing your code, these functions actually check to see if their lists are empty or not--if they are passed an empty list, they fail gracefully by warning you and exiting; if you are running your program under the debugger, it will stop at the exit point. Note that such checking is not required! It would be perfectly acceptable to write these in such a way that they fail quite ungracefully if passed empty lists. Note also that `list_make` is an overloaded function - if called with no arguments, it produces an empty list. If called with an element and a list, it combines them.

Given this list_t interface, you will write the following list processing procedures. Each of these procedures **must be tail recursive**. For full credit, your routines must provide the correct result **and** provide an implementation that is tail-recursive. In writing these functions, you may use only recursion and selection. You are **NOT** allowed to use goto, for, while, or do-while, nor are you allowed to use global variables.

Remember: a tail-recursive function is one in which the recursive call happens **absent any** pending computation in the caller. Besides this, we have another requirement: if the implementation of your target function `tar_func()` needs a helper function `help_func()` to solve the problem tail-recursively, then you should write your target function in the following way:

```
<return_type> tar_func( … )
{
    return help_func( … );
}
```

In other words, your target function only has one statement, which is a `return` statement. The `return` statement gets the return result **directly** from `help_func()`. Notice that the following implementation is illegal, since the return result is not obtained from `help_func()` directly:

```
<return_type> tar_func( … )
{
    return another_func(…, help_func( … ), …);
}
```

However, it is legal to use other function calls as the formal parameters of `help_func()`. For example, the following implementation is allowed:

```
<return_type> tar_func( … )
{
    return help_func(…, another_func( … ), …);
}
```

As a concrete example, the following implementation of factorial is a legal tail-recursive implementation:

```
static int factorial_helper(int n, int result)
// REQUIRES: n >= 0
// EFFECTS: computes result * n!
{
     if (!n) {
          return result;
     }
     else {
          return factorial_helper(n-1, n*result);
     }
}

int factorial_tail(int n)
// REQUIRES: n >= 0
// EFFECTS: computes n!
{
     factorial_helper(n, 1);
}
```

Notice that the return value from the recursive call to factorial_helper() is only returned again---it is not used in any local computation, nor are there any steps left after the recursive call.

The tail-recursive version of factorial uses a helper function. If you define any helper functions, be sure to declare them "**static**", so that they are **not visible** outside your program file. This will prevent any name conflicts in case you give a function the same name as one in the test harness. The function factorial_helper, above, is defined as a static function.

As another example, the following is *not* tail-recursive:

```
int factorial(int n)
// REQUIRES: n >= 0
// EFFECTS: computes n!
{
     if (!n) {
          return 1;
     }
     else {
          return (n * factorial(n-1));
     }
}
```

Notice that the return value of the recursive call to factorial is used in a computation in the caller -- namely, it is multiplied by n.

**Here are the functions you are to implement. There are several of them, but many of them are similar to one another, and the longest is at most tens of lines of code, including support functions.**

```
int size(list_t list);
/*
// EFFECTS: Returns the number of elements in "list".
//          Returns zero if "list" is empty.
*/

int getNth(list_t list, int n);
/*
// REQUIRES: 1. "list" is non-empty.
//           2. n <= the number of elements in "list".
```

```
//
// EFFECTS: Returns the n-th element in list. n begins
//          from one.
//
// For example, getNth((1,2,3), 1) gives the first
// element, which is 1.
*/

int sum(list_t list);
/*
// EFFECTS: Returns the sum of each element in "list".
//          Returns zero if "list" is empty.
*/

int product(list_t list);
/*
// EFFECTS: Returns the product of each element in "list".
//          Returns one if "list" is empty.
*/

int accumulate(list_t list, int (*fn)(int, int), int base);
/*
// REQUIRES: "fn" must be associative.
//
// EFFECTS: Returns "base" if "list" is empty.
//          Returns fn(list_first(list),
//                     accumulate(list_rest(list),
//                                fn, base) otherwise.
//
// For example, if you have the following function:
//
//           int add(int x, int y);
//
// Then the following invocation returns the sum of all elements:
//
//           accumulate(list, add, 0);
*/

list_t reverse(list_t list);
/*
// EFFECTS: Returns the reverse of "list".
```

```
//
// For example: the reverse of ( 3 2 1 ) is ( 1 2 3 )
*/

list_t append(list_t first, list_t second);
/*
// EFFECTS: Returns the list (first second).
//
// For example: append(( 2 4 6 ), ( 1 3 )) gives
// the list ( 2 4 6 1 3 ).
*/

list_t filter_odd(list_t list);
/*
// EFFECTS: Returns a new list containing only the elements of the
//          original "list" which are odd in value,
//          in the order in which they appeared in list.
//
// For example, if you applied filter_odd to the list ( 3 4 1 5 6 )
// you would get the list ( 3 1 5 ).
*/

list_t filter_even(list_t list);
/*
// EFFECTS: Returns a new list containing only the elements of the
//          original "list" which are even in value,
//          in the order in which they appeared in list.
*/

list_t filter(list_t list, bool (*fn)(int));
/*
// EFFECTS: Returns a list containing precisely the elements of "list"
//          for which the predicate fn() evaluates to true, in the
//          order in which they appeared in list.
//
// For example, if predicate bool odd(int a) returns true if a is odd,
// then the function filter(list, odd) has the same behavior as the
// function filter_odd(list).
*/
```

```
list_t insert_list(list_t first, list_t second, unsigned int n);
/*
// REQUIRES: n <= the number of elements in "first".
//
// EFFECTS: Returns a list comprising the first n elements of
//          "first", followed by all elements of "second",
//           followed by any remaining elements of "first".
//
//     For example: insert (( 1 2 3 ), ( 4 5 6 ), 2)
//            is  ( 1 2 4 5 6 3 ).
*/

list_t removeNth(list_t list, unsigned int n);
/*
// REQUIRES: 1. "list" is non-empty.
//           2. n <= the number of elements in "list".
//
// EFFECTS: Returns the list with the n-th element of "list" removed.
//          n starts from one.
//
// For example, removeNth((1, 2, 3), 1) removes the first element,
// giving the list (2, 3); removeNth((1, 2, 3), 2) removes the
// second element, giving the list (1, 3).
*/

list_t chop(list_t list, unsigned int n);
/*
// REQUIRES: "list" has at least n elements.
//
// EFFECTS: Returns the list equal to "list" without its last n
//          elements.
*/
```

## Binary Trees

A binary tree is another fundamental data structure we will use in this project. A binary tree is well formed if:

a) It is the empty tree, or
b) It consists of an integer element, plus two children, called the left subtree and the right subtree, each of which is a well-formed binary tree.

Additionally, we say a binary tree is a "leaf" if and only if both of its children are the EMPTY TREE.

The file recursive.h on Sakai defines the type "tree_t" and the following operations on trees:

```
bool tree_isEmpty(tree_t tree);
// EFFECTS: returns true if tree is empty, false otherwise

tree_t tree_make();
// EFFECTS: creates an empty tree.

tree_t tree_make(int elt, tree_t left, tree_t right);
// EFFECTS: creates a new tree, with elt as it's element, left as
// its left subtree, and right as its right subtree

int tree_elt(tree_t tree);
// REQUIRES: tree is not empty
// EFFECTS: returns the element at the top of tree.

tree_t tree_left(tree_t tree);
// REQUIRES: tree is not empty
// EFFECTS: returns the left subtree of tree

tree_t tree_right(tree_t tree);
// REQUIRES: tree is not empty
// EFFECTS: returns the right subtree of tree

void tree_print(tree_t tree);
// MODIFIES: cout
// EFFECTS: prints tree to cout.
// Note: this uses a non-intuitive, but easy-to-print
// format.
```

There are several functions you are to write for binary trees. These must be recursive, and cannot use any looping structures. They do not need to be tail-recursive.

```
int tree_sum(tree_t tree);
/*
// EFFECTS: Returns the sum of all elements in "tree".
//          Returns zero if "tree" is empty.
*/

bool tree_search(tree_t tree, int key);
/*
// EFFECTS: Returns true if and only if there exists an element in
//          "tree" whose value is "key".
*/

int depth(tree_t tree);
/*
// EFFECTS: Returns the depth of "tree", which equals the number of
//          layers of nodes in the tree.
//          Returns zero if "tree" is empty.
//
// For example, the tree
//
//                           4
//                         /   \
//                        /      \
//                       2        5
//                      / \      / \
//                         3        8
//                        / \      / \
//                       6   7
//                      / \ / \
//
// has depth 4.
// The element 4 is on the first layer.
// The elements 2 and 5 are on the second layer.
// The elements 3 and 8 are on the third layer.
// The elements 6 and 7 are on the fourth layer.
//
*/
```

```
int tree_min(tree_t tree);
/*
// REQUIRES: "tree" is non-empty.
// EFFECTS: Returns the smallest element in "tree".
*/

list_t traversal(tree_t tree);
/*
// EFFECTS: Returns the elements of "tree" in a list using an
//          in-order traversal. An in-order traversal prints
//          the "left most" element first, then the second-left-most,
//          and so on, until the right-most element is printed.
//
//          For any root element, all the elements of its left subtree
//          are considered as on the left of that root element and
//          all the elements of its right subtree are considered as
//          on the right of that root element.
//
// For example, the tree:
//
//                           4
//                         /   \
//                        /      \
//                       2        5
//                      / \      / \
//                         3
//                        / \
//
// would return the list
//
//       ( 2 3 4 5 )
//
// An empty tree would print as:
//
//       ( )
//
*/

bool tree_hasPathSum(tree_t tree, int sum);
/*
// EFFECTS: Returns true if and only if "tree" has a root-to-leaf
//          path such that adding all elements along the path
//          equals "sum".
//
// A root-to-leaf path is a sequence of elements in a tree starting
// with the root element and proceeding downward to a leaf (an element
// with no children).
//
// An empty tree has no root-to-leaf path.
//
// For example, the tree:
//
//                           4
//                         /   \
//                        /     \
//                       1       5
//                      / \     / \
//                     3   6
//                    / \ / \
//
// has three root-to-leaf paths: 4->1->3, 4->1->6 and 4->5.
// Given sum = 9, the path 4->5 has the sum 9, so the function
// should return true. If sum = 10, since no path has the sum 10,
// the function should return false.
//
*/

tree_t mirror(tree_t tree);
/*
// EFFECTS: Returns a new tree so that it is the mirror image
//          of the original tree.
//
// For example, the mirror image of the tree
//
//       4
//      / \
//     2   5
//    / \
//   1   3
//
// is
//
```

```
//       4
//      / \
//     5   2
//        / \
//       3   1
*/
```

We can define a special relation between trees "is covered by" as follows:

- An empty tree is covered by all trees.
- The empty tree covers only other empty trees.
- For any two non-empty trees, A and B, A is covered by B if and only if the top-most elements of A and B are equal, the left subtree of A is covered by the left subtree of B, and the right subtree of A is covered by the right subtree of B.

For example, the tree:

```
        4
       / \
      /   \
     2     5
    / \   / \
       3
      / \
```

covers the tree:

```
         4
        / \
       /
      2
     / \
```

but not the trees:

```
          4                   5
         /                   / \
        /           or
       3
      / \
```

In light of this definition, write the following function:

```
bool covered_by(tree_t A, tree_t B);
/*
// EFFECTS: Returns true if tree A is covered by tree B.
*/
```

With the definition of “covered by”, we can define a relation “contained by”. A tree A is contained by a tree B if

- A is covered by B, or,
- A is covered by any complete subtree of B.

For example, the trees

```
          4                      5
         / \                    / \
        /             and
       2
      / \
```

are contained by the tree

```
         4
        / \
       /   \
      2     5
     / \   / \
        3
       / \
```

but this tree is not:

```
         4
        /
       /
      3
     / \
```

Please write a function implementing the relation “contained by”:

```
bool contained_by(tree_t A, tree_t B);
```

```
/*
// EFFECTS: Returns true if tree A is covered by tree B
//          or any complete subtree of B.
*/
```

There exists a special kind of binary tree, called the sorted binary tree. A sorted binary tree is well-formed if:

1. It is a well-formed binary tree **and**
2. One of the following is true:
   a) The tree is empty.
   b) The left subtree is a sorted binary tree, and any elements in the left subtree are strictly less than the root element of the tree.
   c) The right subtree is a sorted binary tree, and any elements in the right subtree are greater than or equal to the root element of the tree.

For example, the following are all well-formed sorted binary trees:

```
               4                 1
             /   \              / \
            /     \                1
           2       6              / \
          / \     / \                2
         1   3   5   7              / \
        / \ / \ / \ / \
```

While the following are not:

```
        1           1            4
       / \         / \          / \
      1           2   3        3   6
     / \                      / \   \
                             2   1   7
```

You are to write the following function for creating sorted binary trees:

```
tree_t insert_tree(int elt, tree_t tree);
/*
// REQUIRES: "tree" is a sorted binary tree.
//
```

```
// EFFECTS: Returns a new tree with elt inserted at a leaf such that
//          the resulting tree is also a sorted binary tree.
//
// For example, inserting 1 into the tree:
//
//                           4
//                         /   \
//                        /     \
//                       2       5
//                      / \     / \
//                         3
//                        / \
//
// would yield
//                           4
//                         /   \
//                        /     \
//                       2       5
//                      / \     / \
//                     1   3
//                    / \ / \
//
// Hint: There is only one unique position for any element to be
// inserted.
*/
```

## Files

There are several files located in the Project-Two folder of our Sakai Resources:

| | |
|---|---|
| `p2.h` | The header file for the functions you must write |
| `recursive.h` | The list_t and tree_t interfaces |
| `recursive.C` | The implementations of list_t and tree_t. |

You should copy the above files into your working directory. **DO NOT modify these files!** You should put **all** of the functions you write in a single file, called **p2.C (exactly like this!)**. You may use only the C++ standard and iostream libraries, and no others. You may use assert() if you wish, but you do not need to. You may **not** use global variables. You can think of p2.C as providing a library of functions that other programs might use, just as recursive.C does.

## Compiling and Testing

To test your code, you should create a family of test case programs that exercise these functions. Here is a simple illustration to get you started:

```
#include <iostream>
#include "recursive.h"
#include "p2.h"
using namespace std;
int main()
{
    int i;
    list_t listA;
    list_t listB;
    listA = list_make();
    listB = list_make();
    for (i = 5; i>0; i--) {
        listA = list_make(i, listA);
        listB = list_make(i+10, listB);
    }
    list_print(listA);
    cout << endl;
    list_print(listB);
    cout << endl;
    list_print(reverse(listA));
    cout << endl;
    list_print(append(listA, listB));
    cout << endl;
}
```

Compile the above test code with recursive.C and your p2.C. You should write a makefile to do the compiling. When you run the program in Linux, this is the output you should see:

```
./simple_test
( 1 2 3 4 5 )
( 11 12 13 14 15 )
( 5 4 3 2 1 )
( 1 2 3 4 5 11 12 13 14 15 )
```

We have placed two test cases `simple_test.C` and `treeins_test.C` in the Project-Two folder on Saki for you to try out. These two are self-verifying---they tell you if they succeeded or failed.

## Submitting and Due Date

You only need to submit your source code file p2.C (**name it exactly like this!**). The source code file should be submitted via the Assignment tool on Sakai. The due date is 11:59 pm on June 20th, 2012.

## Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

An example of Functional Correctness is whether or not your reverse function reverses a list properly in all cases. An example of an Implementation Constraint is whether reverse() is tail-recursive. General Style speaks to the cleanliness and readability of your code.