Project 2: Recursive Data Structures
EECS 280 – Fall 2013
Due: Wednesday 2 October 2013, 11:55 PM

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
```

List Interface

The file recursive.h 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 non-empty 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.

List Processing Procedures

Given this list_t interface, you will write the list processing procedures below, adhering to the following constraints and guidelines:

- 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
- No static or global variables
- If you define any helper functions, be sure to declare them "static", so that they are not visible outside your program file. See the appendix for more information about tail recursion and helper functions.

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. You may call any of these functions in the implementation of another.

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

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

int accumulate(list_t list, int (*fn)(int, int), int identity);
    /*
    // REQUIRES: fn must be associative.
```

```
    // EFFECTS: return identity if list is empty
    //          return fn(list_first(list),
    //          accumulate(list_rest(list), fn, identity))
    //          otherwise. Be sure to make above code tail-recursive!
    //
    // 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);
    //
    //  The "identity" argument is typically the value for which
    //  fn(X, identity) == X, for any X.
    */

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)
    */

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 ( 4 1 3 0 )
    // you would get the list ( 1 3 )
    */

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
   //
   // For example, if you applied filter_even to the list ( 4 1 3 0 )
   // you would get the list ( 4 0 )
   */

list_t filter(list_t list, bool (*fn)(int));
    /*
    // EFFECTS: returns a list containing precisely the elements of
```

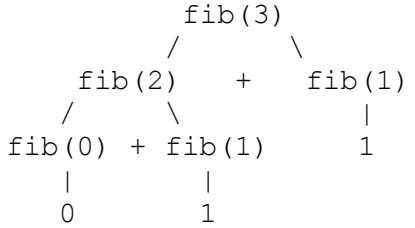
```
    //          list for which the predicate fn() evaluates to true,
    //          in the order in which they appeared in list.
    */

list_t rotate(list_t list, unsigned int n);
    /*
    // EFFECTS: returns a list equal to the original list with the
    //          first n elements moved to the end of the list.
    //
    // For example, rotate(( 1, 2, 3, 4), 2) yields ( 3, 4, 1, 2 )
    */

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

Fibonacci numbers

Not all recursive definitions are necessarily linear-recursive. For example, consider the Fibonacci numbers:

```
fib(0) = 0;
fib(1) = 1;
fib(n) = fib(n-1) + fib(n-2);     (n > 1)
```

This is called a "tree-recursive" definition; the definition of fib(N) refers to fib() twice. You can see why this is so by drawing a picture of evaluating fib(3):



The call pattern forms a tree.

Implement Fibonacci Recursively and Tail-Recursively

You are to write two versions of fib(), as follows. The first version should be written recursively, in this tree pattern. It should not be tail-recursive (and so it should not call the second version!). The second version must be tail-recursive and is tricky, but we've supplied a hint.

```
int fib(int n);
    /*
    // REQUIRES: n >= 0
    // EFFECTS: computes the Nth Fibonacci number
    //          fib(0) = 0
    //          fib(1) = 1
    //          fib(n) = fib(n-1) + fib(n-2) for (n>1).
    // This must not be tail recursive
    */

int fib_tail(int n);
    /*
    // REQUIRES: n >= 0
    // EFFECTS: computes the Nth Fibonacci number
    //          fib(0) = 0
    //          fib(1) = 1
    //          fib(n) = fib(n-1) + fib(n-2) for (n>1).
    // MUST be tail recursive
    // Hint: instead of starting at n and working down, start with
    // 0 and 1 and work *upwards*.
    */
```

Binary Trees

The Fibonacci numbers appear to be tree-recursive, but can be computed in a way that is linear-recursive. This is not true for all tree-recursive problems. For example, consider the following definition of a binary tree:

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.

Tree Interface

The file recursive.h 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.
```

Tree Processing Procedures

There are four functions you are to write for binary trees. These must be recursive, but do not need to be tail-recursive. You may not use any looping structures.

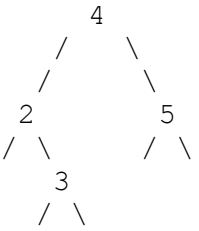
```
int tree_sum(tree_t tree);
    // EFFECTS: returns the sum of all elements in the tree,
    //          zero if the tree is empty



list_t traversal(tree_t tree);
    /* MODIFIES:
    // EFFECTS: returns the elements of tree in a list using an
    //          in-order traversal. An in-order traversal yields a
    //          list with the "left most" element first, then the
    //          second-left-most, and so on, with the right-most
    //          element last.
    //
    //          for example, the tree:
    //
    //                           4
    //                         /   \
    //                        /     \
```

```
//                          2         5
//                         / \       / \
//                            3
//                           / \
// would return the list
//
//        ( 2 3 4 5 )
//
// An empty tree would print as:
//
//        ( )
//
*/
```

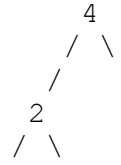
We can define a special relation between trees "is covered by" as follows:

- 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.

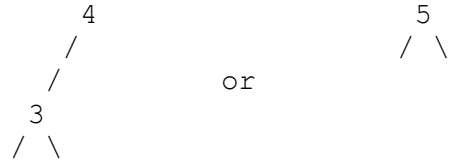
For example, the tree:



Covers the tree:



But not the trees:



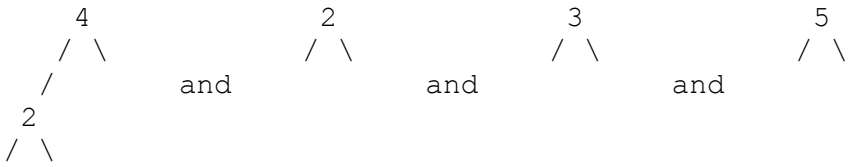
In light of this definition, write the following function:

```
bool contained_by(tree_t A, tree_t B);
       /* EFFECTS: returns true if A is covered by B, or A is covered
       // by any complete subtree of B.
```

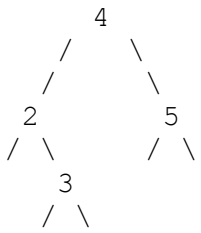
```
        */
```

You need not explicitly write `covered_by`, but we recommend it, as it is likely to make your solution simpler overall to separate it.

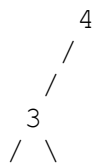
In other words, the trees



Are contained by the tree



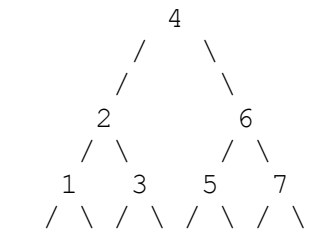
But this tree is not:

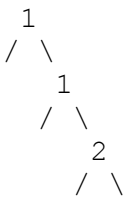


There exists a special kind of binary tree, called the sorted binary tree. A sorted binary tree is well-formed if:

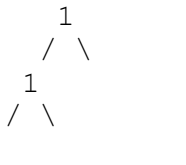
1. It is a well-formed binary tree and
2. One of the following is true:
    a. The tree is empty
    b. The left subtree is a sorted binary tree, and all elements in the left subtree are strictly less than the top element of the tree.
    - AND -
    The right subtree is a sorted binary tree, and all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

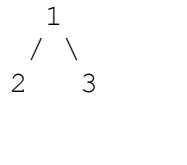
For example, the following are all well-formed sorted binary trees:

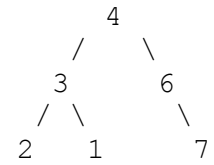




While the following are not:







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 the tree:
    //                            4
    //                          /   \
    //                         /     \
    //                        2       5
    //                       / \     / \
    //                      1   3
    //                     / \ / \
    //
```

```
// Hint: an in-order traversal of a sorted binary tree is always
//       a sorted list, and there is only one unique location for
//       any element to be inserted.
*/
```

Files

There are several files installed in the directory: /afs/umich.edu/class/eecs280/proj2

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

Coding Specifics

You should put **all** of the functions **you** write in a single file, called p2.cpp. 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, static variables, or reference arguments. DO NOT INCLUDE a main function in your p2.cpp file. You can think of p2.cpp as providing a library of functions that other programs might use, just as recursive.cpp does. We will provide a main function when using your code as a library to test your functions.

Testing

To test your code, you should create a family of test case programs that exercise these functions. We have placed a handful of test cases in /afs/umich.edu/class/eecs280/proj2. Here is a simple one to get you started:

simple_test.cpp

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    #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;
    }
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Here is how to build this test case, called simple_test.cpp, into a program, given your own implementation of p2.cpp and our implementation of recursive.cpp:

```
g++ -O1 -pedantic -Wall -Werror -o simple_test simple_test.cpp p2.cpp
recursive.cpp
```

Note that `recursive.cpp` is part of the compile line. This is how the compiler gains access to the routines contained inside the file. DO NOT #include `recursive.cpp` in your `p2.cpp` file! If you do, your submission will not compile with our test cases and you will receive little to no points for the project.

Also note the –O1 flag (capital O, not zero), just as we used in Project 1. It detects uninitialized variables, but should not be used if you are compiling for debugging. When you run it, 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 provided two more test cases for you to try out. These two are self-verifying (they tell you if they succeeded or failed).

Handing in and grading

You should hand in your program, p2.cpp, via submit280 before the deadline. See CTools for more information on submit280. 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.

Appendix

Tail Recursion

Remember: a tail-recursive function is one in which the recursive call happens absent **any** pending computation in the caller. For example, the following is a tail-recursive implementation of factorial:

```
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!
{
  return 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.

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.

Helper Functions

The tail-recursive version of factorial requires a helper function. This is common, but not always necessary. 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.