Project 2: Recursive Data Structures
EECS 280 – Winter 2015
Due: Monday 2 February 2015, 11:55 PM

Introduction

This project will give you experience writing recursive functions that operate on recursively-defined data structures and mathematical abstractions. You will also gain practice testing your code.

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:

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

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

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

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

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

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

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.

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

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

/*
 * REQUIRES: fn must be associative.
 * EFFECTS:  return identity if list is empty.
 *           Otherwise, return the tail recursive equivalent of
 *           fn(list_first(list), accumulate(list_rest(list), fn, identity).
 *           Be sure to make your 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 and fn(identity, X) == X for any X.
 */
int accumulate(list_t list, int (*fn)(int, int), int identity);

/*
 * EFFECTS: returns the reverse of list
 *
 * For example: the reverse of ( 3 2 1 ) is ( 1 2 3 )
 */
list_t reverse(list_t list);

/*
 * EFFECTS: returns the list (first second)
 */
list_t append(list_t first, list_t second);

/*
 * 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_odd(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_odd to the list ( 4 1 3 0 )
 * you would get the list ( 4 0 )
 */
list_t filter_even(list_t list);

/*
 * EFFECTS: returns a new 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 filter(list_t list, bool (*fn)(int));
```

```
/*
 * REQUIRES: n >= 0
 *
 * EFFECTS: Returns a copy of the given list, but with the first n elements
 * rotated to the back of the new list.
 *
 * For example, rotate(( 1, 2, 3, 4, 5), 2) yields ( 3, 4, 5, 1, 2 )
 */
list_t rotate(list_t list, int n);

/*
 * REQUIRES: n >= 0 and 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 insert_list(list_t first, list_t second, int n);

/*
 * REQUIRES n >= 0 and list has at least n elements
 * EFFECTS: returns the list equal to list without its last n
 *          elements
 */
list_t chop(list_t list, int n);
```

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):

```
                 fib(3)
                /      \
          fib(2)    +   fib(1)
         /     \           |
     fib(0)  + fib(1)      1
        |        |
        0        1
```

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.

```
/*
 * 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 be recursive but need not be tail recursive
 */
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).
 * MUST be tail recursive
 * Hint: instead of starting at n and working down, start at
 * 0 and 1 and work *upwards*.
 */
int fib_tail(int n);
```

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:

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

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

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

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

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

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

// MODIFIES: cout
// EFFECTS: prints tree to cout.
//          ~~Note: this uses a non-intuitive, but easy-to-print~~
//             ~~format.~~
//          Now, as of Fall 2014, this function prints
//          the tree so that it ACTUALLY LOOKS LIKE A TREE!!!!1
void tree_print(const tree_t& tree);
```

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.

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

Tree Containment

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:

```
/*
 * EFFECTS: returns true if A is covered by B or if A is contained by either
 *          of B's subtrees.
 */
bool contained_by(tree_t A, tree_t B);
```

You need not explicitly write `covered_by`, but we recommend separating it as a helper function to simplify your solution.

In other words, the trees

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

Are contained by the tree

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

But this tree is not:

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

Tree Insert

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:

```
               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:

```
/*
 * 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: 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.
 */
tree_t insert_tree(int elt, tree_t tree);
```

Files

You can download the following files from CTools:

`p2.h`	the header file for the functions you must write
`p2-tests.cpp`	the file to which you must add test cases for the recursive functions you implement.
`test_helpers.h`	contains functions that will make writing your test cases easier
`test_helpers.cpp`	implementations of the functions in test_helpers.h
`recursive.h`	the list_t and tree_t interfaces that you will use to implement
`recursive.cpp`	the implementations of list_t and tree_t
`Makefile`	you will use this file to compile your code and prepare an archive that you will submit to the autograder

(The next three files contain public test cases)

- `simple_test.cpp`
- `simple_test.out.correct`
- `filter_test.cpp`
- `tree_insert_test.cpp`

(The next four files contain implementation details of `recursive.cpp`. Download them but don't worry about their contents. Do NOT directly use the functions in `Recursive_list.h` or `Binary_tree.h`. Use only the functions given in `recursive.h`)

- `Recursive_list.h`
- `Recursive_list.cpp`
- `Binary_tree.h`
- `Binary_tree.cpp`

There is also a `p2.tar.gz` archive that contains all the files for the project. You can extract this archive by running:

```
        tar -xzvf p2.tar.gz
```

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.

TIP: In order to make your program compile before you have implemented all the functions, you can write "function stubs." A function stub is a function that compiles, but doesn't do anything useful. For example:

```
int sum(list_t list)
```

```
{
    assert(false);
}
```

If you start out by writing stubs for all of the required functions, you won't have to wait until the very end for your code to compile.

Testing

To gain practice testing your code, you are required to test the recursive functions you implement (i.e. the functions listed in p2.h. You will *not* write tests for your static helper functions). Requirements:

- You must test each function at least once.
- You must add your tests to the main() function of the file p2-tests.cpp.
- Your tests must use assert(false) or return nonzero from main in order to show that a test has failed.
- You may print as much output as you like.

Protip: Write tests for the functions FIRST! (i.e. Write tests for sum(), and then implement sum()). It sounds like a pain, but you gain two important things by coding this way:

1. You avoid being under the illusion that your code works when it's actually full of bugs.
2. When you make changes to code that you wrote previously, you can re-run your test cases and immediately know if you broke something (yes, you will break things).

This practice is called **test-driven development.**

Compiling

Here is how to build your tests and the public tests into executable programs and run them:

```
make test
```

This will cause a program called 'make' to consult the Makefile for instructions on how to compile 'test,' and then run those instructions.
In the case of p2-tests, the following command will be run:

```
g++ -Wall -Werror -pedantic -O2 \
      p2-tests.cpp p2.cpp Recursive_list.cpp Binary_tree.cpp \
      recursive.cpp test_helpers.cpp -o p2-tests
```

Note that `test_helpers.cpp`, `recursive.cpp`, `Binary_tree.cpp`, and `Recursive_list.cpp` are part of the compile command. This is how the compiler gains access to the routines contained inside the files. DO NOT #include any .cpp files 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 –O2 flag (capital O, not zero). We need this option to turn on tail recursion optimization, but should not be used if you are compiling for debugging (use -g instead for debugging).

Handing in and grading

You will submit an archive called submit.tar.gz that contains p2.cpp, p2-tests.cpp, and group.txt. To create your archive, run the following command:

```
make tar
```

This will create a file called submit.tar.gz. You can upload your archive to the autograder on CTools.

Your project will be graded along these criteria:

1) Functional Correctness
2) Implementation Constraints
3) General Style
4) Presence of test cases

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. You must test each of the recursive functions. We will make sure your tests compile, but the contents will only be evaluated by hand-graders. **Take this opportunity to practice writing good tests. In future projects, we will evaluate your tests to see how well they reveal bugs in intentionally buggy solutions.**

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.