CIS 120 Midterm I   October 12, 2012

Name (printed): ____________________________
Pennkey (login id): ____________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________________ Date: _________________________

1	/27
2	/20
3	/16
4	/17
5	/20
Total	/100

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 100 total points.
- There are 9 pages in this exam.
- Make sure your name and Pennkey (a.k.a. username) is on the top of this page.
- Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.

1. **Program Design (27 points total)**

   Use the four-step design methodology to implement a function called `intersperse` that, given a value `c` and a list of values, returns a new list in which `c` is placed between every pair of adjacent values of the original list. For example, `intersperse 0 [1;2;3]` should yield the list `[1;0;2;0;3]`.

   The function should be *generic* and work for any type of lists, not just lists of integers.

(0 points) Step 1 is *understanding the problem*. You don't have to write anything for this part—your answers below will demonstrate whether or not you succeeded with Step 1.

(3 points) Step 2 is *formalizing the interface*. Write down the *type* of the `intersect` function as you might find it in a `.mli` file or module interface:

```
val intersperse:
```

(12 points) Step 3 is *writing test cases*.

Complete the following tests with examples of the expected behavior. We have done the first one for you. Note that some test cases are better than others, and credit will be assigned accordingly: make sure your tests cover a sufficiently broad range of "interesting" inputs. Fill in the description string of the `run_test` function with a short explanation of why the test case is interesting.

i.
```
let test () : bool =
  [1;0;2;0;3] = (interspserse 0 [1;2;3])
;; run_test "comes from the problem description" test
```

ii.
```
let test () : bool =

  __________ = (intersperse __________ ____________________________)

;; run_test "______________________________________________________________" test
```

iii.
```
let test () : bool =

  __________ = (intersperse __________ ____________________________)

;; run_test "______________________________________________________________" test
```

iv.
```
let test () : bool =

  __________ = (intersperse __________ ____________________________)

;; run_test "______________________________________________________________" test
```

(12 points) Step 4 is *implementing the program*. Fill in the body of the `intersperse` function to complete the design. Do *not* use any list library functions (such as `fold`, or `@`) to solve this problem. If you would like to use a helper function in your answer, you must define it.

```
let rec intersperse (c:_______________) (l:_______________) : _______________ =
```

2. List Processing (20 points)

For each of the following programs, write the value computed for `r`:

a.
```
let rec h (l:int list) : int =
  begin match l with
    | [] -> 0
    | x::xs -> x * (h xs)
  end

let r : int = h [1;2;3]
```

b.
```
let rec g (l:'a list) : 'a list =
  begin match l with
    | [] -> []
    | [x] -> [x]
    | x::y::xs -> if x < y then x::(g (y::xs)) else y::(g (x::xs))
  end

let r : int list = g [1;3;2;0]
```

c.
```
let rec f (p: 'a -> bool) (l:'a list) : 'a list * 'a list =
  begin match l with
    | [] -> ([], [])
    | x::xs ->
        let (l,r) = f p xs in
          if p x then (x::l, r) else (l, x::r)
  end

let r : (int list * int list) = f (fun (x:int) -> x > 0) [0;1;2;-3;4]
```

The last two programs refer to the following definitions.

```
let rec transform (f: 'a -> 'b) (x: 'a list): 'b list =
 begin match x with
 | [] -> []
 | h :: t -> (f h) :: (transform f t)
 end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (x: 'a list): 'b =
 begin match x with
 | [] -> base
 | h :: t -> combine h (fold combine base t)
 end
```

d.
```
let k (x: 'a list) : 'a list =
  fold (fun (h:'a) (v:'a list) -> v @ [h]) [] x

let r : int list = k [1;3;2;4]
```

e.
```
let j (x : int list list) : int list =
  let transformer (l:int list) : int =
    fold (fun (x:int) (v:int) -> x + v) 0 l in
  transform transformer x

let r : int list = j [[1;2;3];[4;5];[]]
```

3. Types (16 points)

For each OCaml value or function definition below, fill in the blank where the type annotation could go or write "ill typed" if there is a type error. If an expression can have multiple types, give the most generic one. We have done the first one for you.

Some of these definitions refer to functions from the `Map1` module, which has the following **abstract** interface:

```
module type MAP =
  sig
    type ('a, 'b) map
    val empty : ('a, 'b) map
    val is_empty : ('a, 'b) map -> bool
    val mem : 'a -> ('a, 'b) map -> bool
    val find : 'a -> ('a, 'b) map -> 'b
    val add : 'a -> 'b -> ('a, 'b) map -> ('a, 'b) map
    val remove : 'a -> ('a, 'b) map -> ('a, 'b) map
    val from_list : ('a * 'b) list -> ('a, 'b) map
    val bindings : ('a, 'b) map -> ('a * 'b) list
  end
module Map1 : MAP = struct ... end

;; open Map1

let x : ______ (int,string) map ______________ = add 120 "is fun" empty


let a : ________________________________________ = (2::[])::[]

let b : ________________________________________ = 2 + "three"

let c : ________________________________________ = add 3 true empty

let d : ________________________________________ = add 3 true

let e : ________________________________________ = mem 3 [1;2;3]

let f : ________________________________________ = fun (g:int -> int) -> g 3

let g : ________________________________________ = fun (x:int) (y:int) -> x + y

let h : ________________________________________ = add 3 (from_list [(1,2)]) empty
```

4. Binary Search Trees (17 points)

Recall the definition of generic binary trees and the binary search tree `insert` function:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

let rec insert (t:'a tree) (n:'a) : 'a tree =
 begin match t with
  | Empty -> Node(Empty, n, Empty)
  | Node(lt, x, rt) ->
     if x = n then t
     else if n < x then Node (insert lt n, x, rt)
     else Node(lt, x, insert rt n)
 end
```

a. (5 points) Circle the trees that satisfy the *binary search tree invariant*. (Note that we have omitted the Empty nodes from these pictures.)

```
(a)          (b)         (c)        (d)       (e)

  4            2            2         2         2
 / \          / \          / \         \         \
2   5        5   6        5   6         5         5
     \          /              \       / \       / \
      6        4                4     4   6     4   4
```

b. (12 points) For each definition below, circle the letter of the tree above that it constructs or "none of the above".

```
let t1 : int tree =
 insert (Node(Node(Empty, 5 Empty), 2, Node(Empty, 6, Empty))) 4
```

(a) (b) (c) (d) (e) none of the above

```
let t2 : int tree =
 insert (insert (insert (insert Empty 4) 2) 5) 6
```

(a) (b) (c) (d) (e) none of the above

```
let t3 : int tree =
 insert (insert (insert (insert Empty 2) 5) 4) 6
```

(a) (b) (c) (d) (e) none of the above

```
let t4 : int tree =
 insert (insert (insert (insert Empty 5) 2) 4) 6
```

(a) (b) (c) (d) (e) none of the above

5. **Lists and Binary Trees (20 points)**

This problem uses the same datatype of trees as in Problem 4, but the trees are *not* binary search trees.

Consider how to compute the *sum* of the values at each *level* of an `int tree`. For example, given the tree `t` shown below, `level_sum t` computes the list `[3;4;5;4]`. Here, `3` is the value at the root of the tree, `4` is the sum of integers at level 1, `5` is the sum of values at level 2, and the last `4` is the sum of the values at level 3. In general, the i^{th} element of the list is the sum of values at the i^{th} level of the tree (starting at $i = 0$).

```
t : int tree =
Level 0:           3               -->   3
                  / \
                 /   \
Level 1:        2     2            -->   4   =   (2 + 2)
               / \   /
Level 2:      0   1 4              -->   5   =   (0 + 1 + 4)
                     \
Level 3:              4            -->   4
```

When thinking about how to implement `level_sum`, you created the following test code:

```
let leaf (i:int) : int tree = Node(Empty, i, Empty)

let t_left  : int tree = Node(leaf 0, 2, leaf 1)
let t_right : int tree = Node(Node(Empty, 4, leaf 4), 2, Empty)
let t       : int tree = Node(t_left, 3, t_right)

let test () : bool =
  (level_sum Empty) = []
;; run_test "level_sum Empty" test

let test () : bool =
  (level_sum t_left) = [2; 1]
;; run_test "level_sum left subtree" test

let test () : bool =
  (level_sum t_right) = [2; 4; 4]
;; run_test "level_sum right subtree" test

let test () : bool =
  (level_sum t) = [3; 4; 5; 4]
;; run_test "example from diagram" test
```

(Problem continues on next page.)

Implement the function `level_sum` by using the recursion pattern for binary trees.

Hints:

a. Decompose the problem into *two* functions: `level_sum` itself, and a helper function for use in *combining* the results of recursive calls to `level_sum`.

b. The `helper` function should take two `int list` values as inputs and produce an `int list`.

c. The test cases for `t_left` and `t_right` give the results of calling `level_sum` on the subtrees of `t`. Think about how to combine those results (using `helper`) to get to the answer for `level_sum t`.

```
let rec helper (l1:int list) (l2:int list) : int list =
```

```
let rec level_sum (t: int tree) : int list =
```