CIS 120 Midterm I   October 4, 2013

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	/20
2	/10
3	/16
4	/14
5	/20
6	/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.
- 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. List Processing (20 points)

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

a.

```
let x : int = 42
let f (y : int) : int = y + x
let x : int = 120

let r : int = f 0
```

Answer: `r =`

b.

```
let rec f (l : 'a list) : ('a list * 'a list) =
 begin match l with
 | [] -> ([], [])
 | [x] -> (l , [])
 | u::v::w ->
    let (y,z) : ('a list * 'a list) = f w in
    (u::y, v::z)
 end

let r : ('a list * 'a list) = f [1;2;3;4;5]
```

Answer: `r =`

c.

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

let r : int list = f [(fun x -> - x); (fun x -> x * (x + 1))] 6
```

Answer: `r =`

d.

```
type foo = {mutable bar : int}
let f (x : foo) (y : foo) : int * int * int =
 let z = x in
 x.bar <- y.bar + 1;
 (x.bar, y.bar, z.bar)

let r : int * int * int = f {bar = 0} {bar = 0}
```

Answer: `r =`

2. **Types (10 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. Recall that the `@` operator appends two lists together in OCaml. We have done the first one for you. Consider the definitions to be below the following code:

```
module type MAP = sig
  type ('a * 'b) map
  val fromList : ('a * 'b) list -> ('a * 'b) map
end

module LMap : MAP = struct
  type ('a * 'b) map = ('a * 'b) list
  let fromList (l : ('a * 'b) list) = l
end

open LMap;;
```

```
let x : ____________ int list ________________________ = [2 + 2]

let a : ______________________________________________ = 42 ^ " 42"

let b : ______________________________________________ = [42] :: [[]]

let c : ______________________________________________ = [42] :: [42]

let d : ______________________________________________ = [("cis", 120)]

let e : ______________________________________________ = fromList [(120, 42)]

let f : ______________________________________________ =
  fromList ([("benjamin",42)] @ [("pierce", 120)])

let g : ______________________________________________ =
  fromList [("benjamin", "cis"); ("pierce", 120)]

let h : ______________________________________________ =
  fromList [(120,fromList [("benjamin", 42)])]

let i : ______________________________________________ = (fun f -> f 42)

let j : ______________________________________________ = (fun x -> x + "foo")
```

3. Binary Trees and Binary Search Trees (16 points)

Recall the definition of generic binary trees and the BST `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. Circle the trees that satisfy the *binary search tree invariant*. (Note that we have omitted the `Empty` nodes from these pictures, to reduce clutter.)

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

 1             4           4        1          4
  \           / \         / \        \          \
   2         2   6       1   6        3          5
    \       / \         / \   \      / \          \
     3     1   5       0   2   8    2   4          5
      \                                             \
       4                                             6
```

b. For each definition below, circle the letter of the tree that it constructs or “none of the above”.

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

 1              4            4           1           4
  \            / \          / \           \           \
   2          2   6        1   6           3           5
    \        / \          / \   \         / \           \
     3      1   5        0   2   8       2   4           5
      \                                                   \
       4                                                   6
```

```
let t1 : int tree =
 Node(Empty, 1, Node(Node(Empty,2,Empty), 3, Node (Empty,4,Empty)))
```

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

```
let t2 : int tree =
 insert (insert (insert (insert Empty 1) 2) 3) 4
```

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

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

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

```
let t4 : int tree =
 insert (Node ((insert (insert (insert Empty 2) 1) 5), 4, Empty) 6
```

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

4. Modules (14 points)

For this question, suppose we've written the following definition of a module signature `MOD` and two modules `ModOne` and `ModTwo` conforming to the signature `MOD`:

```
module type MOD = sig
  type t
  val x : t
  val f : t -> int
end

module ModOne : MOD = struct
  type t = bool
  let x = false
  let f (b:t) : int = if b then 1 else 0
end

module ModTwo : MOD = struct
  type t = int
  let x = 41
  let y = 16
  let f (b:t) : int = b + 1
end
```

a. Does the following module definition typecheck? If not, *briefly* (one sentence) explain why not.

```
module MyMod3 : MOD = struct
  type t = int
  let x = 16
end
```

b. Does the following client code typecheck? If so, what value does it print? If not, briefly explain why not.

```
;; print_int (ModOne.f ModOne.x)
```

c. Does the following client code typecheck? If so, what value does it print? If not, briefly explain why not.

```
;; print_int (ModOne.f true)
```

d. Does the following client code typecheck? If so, what value does it print? If not, briefly explain why not.

```
;; print_int (ModTwo.f ModTwo.y)
```

e. Does the following client code typecheck? If so, what value does it print? If not, briefly explain why not.

```
;; print_int (ModOne.f ModTwo.x)
```

f. Does the following client code typecheck? If so, what value does it print? If not, briefly explain why not.

```
;; print_int (ModOne.f ModOne.x + ModTwo.f ModTwo.x)
```

g. List *all* the values that could possibly be printed when we run the following client code (for all possible ways of filling in the body of z).

```
let z : ModTwo.t = ...
;; print_int (ModTwo.f z)
```

5. Program Design (20 points)

Use the four-step design methodology to implement a function called `rotations` that computes all the cyclic permutation of a list, ie. given a list $[v_1, v_2, \ldots, v_n]$ of n values (where $n \geq 0$), returns the list

$$[[v_1; v_2; \ldots; v_n]; [v_2; \ldots; v_n; v_1]; [v_n; v_1; \ldots; v_{n-1}]]$$

also of length n.

For example, `rotations [1;2;3;4]` should yield the list:

```
[ [1;2;3;4]; [2;3;4;1]; [3;4;1;2]; [4;1;2;3] ]
```

a. 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.

b. Step 2 is *formalizing the interface*. Write down the *type* of the `rotations` function as you might find it in a `.mli` file or module interface.

```
val rotations:
```

c. Step 3 is *writing test cases*. Complete the following three tests with the expected behavior. We have done the first one for you, based on the problem description.

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" input numbers and lists. Fill in the description string of the `run_test` function with a short explanation of *why* the test case is interesting. Your description should not just restate the test case, e.g. "rotations [1;2;3]".

i.
```
let test () : bool =
  rotations [1;2;3;4] = [ [1;2;3;4]; [2;3;4;1]; [3;4;1;2]; [4;1;2;3] ]
;; run_test "rotations on normal list" test
```

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

  (rotations ________________________________) = __________________

;; run_test "______________________________________________________" test
```

iii.
```
let test () : bool =
  (rotations ________________________________) = ____________________
;; run_test "____________________________________________________" test
```

d. Step 4 is *implementing the program*. Fill in the body of the `rotations` function to complete the design. You can use `@` or any of the higher order functions in the appendix in your answer.
Hint : You can also define an auxilliary function.

```
let rotations (l : ______________________) : ________________________ =
```

6. Higher-Order Functions (20 points)

In this problem you will be guided into coding a function `cartesian`, which computes the so-called *cartesian product* of two lists `l1` and `l2` — that is, the list of all tuples (a, b) where a comes from `l1` and b comes from `l2`.

You should use a single higher-order function — one of `transform`*,* `filter`*, or* `fold` *— in your solution to each of the subproblems (except the last).*

For reference, the definitions of these functions are given in the appendix.

a. Write a function `make_product_list` that, given a list `l` and a value `a`, returns a list of tuples whose first element is `a` and whose second elements ranges over all the elements of `l`. For example, `make_product_list [3;4;5] 1 = [(1,3);(1,4);(1,5)]`.

Your answer should be a call of a single higher-order function with some argument

```
let make_product_list (l : 'b list) (a : 'a): ('a * 'b) list =
```

b. Write a function `cartesian_helper` that given a list `[a1;a2;...;an]` and another `[b1;b2;...;bm]`, where $m, n \geq 0$, produces the following list of lists:

```
[ [(a1,b1); (a1,b2); ... ; (a1,bm)];
  [(a2,b1); (a2,b2); ... ; (a2,bm)];
                ...
  [(an,b1); (an,b2); ... ; (an,bm)] ]
```

For example,

```
cartesian_helper [1;2] [3;4] = [[(1, 3); (1, 4)]; [(2, 3); (2, 4)]].
```

Hint: You should use a higher-order function and `make_product_list`.

```
let cartesian_helper (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list list =
```

c. Write a function `concat` that given a list of lists produces a single list that is the concatenation of all the elements of the original list. For example:

```
concat [[(1,3);(1,4)];[(2,3);(2,4)]] = [(1,3);(1,4);(2,3);(2,4)]
```

Again, you answer should be a call to a single higher-order function with some argument.

```
let concat (l : 'a list list) : 'a list =
```

d. Finally, use the functions you defined above to write the function `cartesian`, defined in the beginning of this problem. For example,

```
cartesian [1;2] [3;4;5] = [(1,3);(1,4);(1,5);(2,3);(2,4);(2,5)]
```

Hint: Do not use any higher-order functions or recursion here. A simple combination of the functions you have coded above should be enough!

```
let cartesian (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =
```

Appendix

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

let rec filter (f: 'a -> bool) (l: 'a list) : 'a list =
 begin match l with
   | [] -> []
   | h::t -> if f h then h :: filter f t else filter f t
 end
```