Name: ______________________________

Pennkey (letters, not numbers): ______________________________

CIS 120 Midterm I

February 15, 2012

1	/24
2	/20
3	/10
4	/25
5	/21
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 11 pages in this exam.
- Make sure your name and Pennkey (a.k.a. username) is on the top of this page.

1. **Program Design (24 points total)**

Suppose you have two **sorted** lists and would like to find out which elements they have in common.

Use the four step design methodology to implement a function called `intersect` that returns the elements that are contained in both lists. For example, the intersection of the lists [1;2;3;4] and [0;2;4;5] is the list [2;4].

(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.

> *When completing the steps below, consider the following:*
>
> - You may *assume* that the input lists are sorted and need not detect when they are not.
> - The function should be *generic* and work for any type of sorted lists, not just lists of integers.
> - Each input list *may* contain repeated elements. If an element is repeated in *both* lists, then it should be repeated in the output. If it appears only once in one list and is repeated in the other list, then it should appear only once in the output.

(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 intersect :
```

(9 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 =
  [2;4] = (intersect [1;2;3;4] [0;2;4;5])
;; run_test "comes from the problem description" test
```

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

  _________ = (intersect ________________ ________________)

;; run_test "__________________________________________________" test
```

iii.
```
let test () : bool =
  __________ = (intersect __________________ __________________)
;; run_test "________________________________________________________" test
```

iv.
```
let test () : bool =
  __________ = (intersect __________________ __________________)
;; run_test "________________________________________________________" test
```

(12 points) Step 4 is *implementing the program*. Fill in the body of the `intersect` 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 intersect (l1:________________) (l2:________________) : ________________ =
```

2. List Processing (20 points)

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

a.
```
let rec z (x:int list) : int list list =
  begin match x with
    | [] -> [ [] ]
    | _::t -> x :: (z t)
  end
let r : int list list = z [1;2;3]
```

b.
```
let rec g (f:'a -> 'a list) (x:'a list) : 'a list =
  begin match x with
    | [] -> []
    | h::t -> f h @ g f t
  end
let r : int list = g (fun (x:int) -> [x;x]) [1;2;3]
```

c.
```
let rec m (x:int option list) : int list =
  begin match x with
    | [] -> []
    | (Some y)::t -> y :: m t
    | None :: t -> m t
  end
let r : int list = m [Some 1; None; Some 2]
```

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 rec k (x: int list) : int list list =
  fold (fun (h:int) (v:int list list) -> x :: v) [] x
let r : int list list = k [1;2]
```

e.
```
let rec f (x : int list) : int list list =
    transform (fun (h:int) -> h :: x) x
let r : int list list = f [1;2]
```

3. **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. We have done the first one for you.

Some of these definitions refer to functions from the `Set1` module, which has the following **abstract** interface:

```
module type Set = sig
  type 'a set
  val empty : 'a set
  val is_empty : 'a set -> bool
  val mem :'a ->'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val union : 'a set -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set
  val equal : 'a set -> 'a set -> bool
  val elements : 'a set -> 'a list
end
module Set1 : Set = ...
;; open Set1
```

```
let x : ______ int set ______ = add 3 empty


let a : _________________________ = [2; "four"]

let b : _________________________ = 2 :: 4

let c : _________________________ = (2,4)

let d : _________________________ = add [3] empty

let e : _________________________ = add 3 [1;2;3]

let f : _________________________ = list_to_set [1;2;3]

let g : _________________________ = fun (x : int) -> x + 1

let h : _________________________ = (fun (x : int) -> x + 1) 10

let i : _________________________ = fun (f : int -> bool) -> f 3

let j : _________________________ = fun (x:'a set) -> add x empty
```

4. Binary Trees (25 points)

Recall the definition of generic binary trees:

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

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)

    3           1              3       3             2
   / \           \            /         \           / \
  2   4           2          2           2         1   3
 / \               \        /             \
1   0               4      1               1
                   /
                  3
```

b. (8 points) For each definition below, circle the letter of the tree above that it constructs or "none of the above".

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

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

```
let t2 : int tree =
  Node (Empty, 3, Node (Empty, 2, Node (Empty, 1, Empty)))
```

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

```
let t3 : int tree =
  Node (Empty, 1, Node (Empty, 2, Node (Empty, 3, Empty)))
```

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

```
let t4 : int tree =
  Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty))
```

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

c. (12 points) Complete this definition of a function that returns the *leaves* of the given tree from left-to-right. For example, calling leaves on tree `(a)` returns `[1;0;4]`. You may use the `@` operator (i.e. list append) in your solution.

```
let rec leaves (t:'a tree) : _______________ =
```

5. Binary Search Trees (21 points)

a. (9 points) Recall the *delete* function for binary search trees from class. (This function uses the same `tree` datatype from the previous problem.)

```
let rec tree_max (t:'a tree) : 'a =
 begin match t with
   | Empty -> failwith "tree_max called on empty tree"
   | Node(_,x,Empty) -> x
   | Node(_,_,rt) -> tree_max rt
 end

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

(This problem continues on the next page.)

Let `t` be the BST depicted below.

```
     6
    / \
   4   7
  / \   \
 2   5   8
```

For each separate call to delete *with this tree*, draw the result:

- `delete t 2`

- `delete t 7`

- `delete t 6`

b. (12 points) Implement `bst_filter`. The `bst_filter` function applies a given predicate to each element in an input tree to see if it should be included in the output. (This function is analogous to the list `filter` function from homework four.)

For example below, filtering the tree on the left with a predicate for even numbers results in the tree on the right:

```
      6                                         6
     / \       bst_filter is_even              / \
    4   7          -------->                  4   8
   / \   \                                   /
  2   5   8                                 2
```

Below, complete the definition, including the types of `pred` and the result type of the function. In your implementation, you **must** use the BST `delete` function.

```
let rec bst_filter (pred: ________________) (t : 'a tree) : ____________ =
```