Name: ___________________________

Pennkey (letters, not numbers): ___________________________

CIS 120 Midterm I

February 15, 2012

SOLUTIONS

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 : 'a list -> 'a list -> 'a list
```

Grading Scheme.

- *-1* `int list -> int list -> int list` *instead of generic type*
- *-1 any argument/result is "list" instead of "'a list"*
- *-2 function decl instead of type (i.e.* `(x: 'a list) (y : 'a list) : 'a`*)*

(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
```

Good answers:

```
(1)  [] = intersect [] [1;2;3]
```

Intersect with 1st nil is nil

(2) [1;1;2] = intersect [1;1;2;3] [1;1;2]

Duplicate elements in output

(3) [1;2;3] = intersect [1;2;3] [1;2;3]

A list intersected with itself is the same list.

(4) [] = intersect [1;3;5] [4;6]

Empty intersect of non-empty lists

Grading Scheme. 3 points per test case. We deducted one point for each of the following errors

- *wrong answer to test*
- *not "interesting"*
- *poor or no description (i.e. description just states what the test case is, not why it was interesting.)*
- *an input lists is not sorted*

(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:int list) (l2:int list) : int list =
 begin match (l1,l2) with
  | ([],_) -> []
  | (_,[]) -> []
  | (h1::t1,h2::t2) ->
     if h1 == h2
     then h1 :: intersect t1 t2
     else if h1 < h2 then intersect t1 l2
          else intersect l1 t2
 end
```

Grading scheme:

- *no deduction for minor syntax errors*
- *-4 No "merge" behavior (i.e. not testing h1 ¡ h2)*
- *-4 Incorrect base cases*
- *-4 Not recursing/pattern matching on both lists*
- *various other errors at discretion*

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]
```

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

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

```
[1;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]
```

```
[[1;2];[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]
```

```
[[1;1;2];[2;1;2]]
```

Grading scheme, each answer worth four points:

- *no deduction for minor syntax*
- *-1 Each missing values from list*
- *-1 Each extra value in list*
- *-2 Extra list structure in answer*
- *Other errors at discretion*

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 : ____ ill typed ______ = [2; "four"]

let b : ____ ill typed ______ = 2 :: 4

let c : ____ int * int ______ = (2,4)

let d : ____ int list set ___ = add [3] empty

let e : ___ ill typed _______ = add 3 [1;2;3]

let f : _____ int set _______ = list_to_set [1;2;3]

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

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

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

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

Grading scheme: 1 point per answer. `int tuple` *not accepted for part (c) as tuples need to specify the types of both components.*

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
```

Answer: (b), (c), (e)

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 Answer: (c)

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

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

Answer: (d)

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

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

Answer: 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

Answer: (e)

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) : 'a list =
  begin match t with
    | Empty -> []
    | Node (Empty, x, Empty) -> [x]
    | Node(lt, x, rt) -> (leaves lt) @ (leaves rt)
  end
```

Grading scheme:

- *no deduction for minor syntax errors*
- *-1 each error in interface*
- *-2 major syntax error*
- *-2 wrong result for Empty*
- *-2 returns wrong type of answer (tree instead of list)*
- *-6 similar function implemented correctly (i.e. inorder)*
- *other errors at discretion*

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`
  Answer:

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

- `delete t 7`
  Answer:

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

- `delete t 6`
  Answer:

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

Grading scheme: three points per answer. No partial credit.

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 : 'a -> bool) (t : 'a tree) : 'a tree =
  begin match t with
  | Empty -> Empty
  | Node (lt, x, rt) ->
     if pred x
     then Node (bst_filter pred lt, x, bst_filter pred rt)
     else bst_filter pred (delete t x)
  end
```

or

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

Grading scheme:

- *no deduction for minor syntax errors*
- *-1 point (each) interface types*
- *-1 omitting pred from recursive call*
- *-2 not testing pred x*
- *-2 not calling delete correctly*
- *-2 not returning a tree*
- *-3 not constructing new node correctly*
- *other errors at discretion*