CIS 120 Midterm I February 21, 2014

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	/16
2	/20
3	/16
4	/12
5	/12
6	/12
7	/12
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. **Program Design (16 points)**

Use the four-step design methodology to implement a function called `trim` that, when given an integer n and a list x, returns the list with all occurrences of n removed *from the beginning of the list*.

For example, `trim 1 [1;1;2;1]` should yield the list `[2;1]`.

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 `trim` function as you might find it in a `.mli` file or module interface.

```
val trim: ______________________________________________
```

c. Step 3 is *writing test cases*. Complete the following 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.

i.
```
let test () : bool =
  trim 1 [1;1;2;1] = [2;1]
;; run_test "given from the problem description" test
```

ii.
```
let test () : bool =
   (trim ________________________________) = __________________
;; run_test "________________________________________________________" test
```

iii.
```
let test () : bool =
   (trim ________________________________) = ____________________
;; run_test "________________________________________________________" test
```

d. Step 4 is *implementing the program*.

```
let rec trim (n : ______________) (x : ____________________) : _______________ =
```

2. Types (20 points)

For each OCaml value below, fill in the blank where the type annotation could go or write "ill typed" if there is a type error. Recall that the `@` operator appends two lists together in OCaml. We have done the first one for you.

```
let x : ____________ int list _______________ = [ 120 ]


let a : ______________________________________ = 1 :: [2]

let b : ______________________________________ = 1 :: [[]]

let c : ______________________________________ = [(1,true)]

let d : ______________________________________ = (1, true)

let e : ______________________________________ = [1; true]

let f : ______________________________________ = [[]] @ [[1]]

let g : ______________________________________ = [1] @ []

let h : ______________________________________ = (fun x -> fun y -> x + y) 3

let i : ______________________________________ = [fun x -> x + 1; fun x -> x - 1]

let j : ______________________________________ = Some 3
```

3. Datatypes and Trees (16 points)

Consider the following definition of trees with integers stored only at the leaves:

```
type leafy_tree =
  | Leaf of int
  | Branch of leafy_tree * leafy_tree
```

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

a.
```
let rec f (t : leafy_tree) : int =
  begin match t with
  | Leaf x -> x
  | Branch (l, r) -> max (f l) (f r)
  end
let r : int = f (Branch (Leaf 1, Branch (Leaf 3, Leaf 2)))
```

Answer: `r =`

b.
```
let rec g (y : int) (t : leafy_tree) : leafy_tree =
  begin match t with
  | Leaf x -> Leaf y
  | Branch (l, r) -> Branch (g y l, g y r)
  end
let r : leafy_tree = g 4 (Branch (Leaf 1, Leaf 2))
```

Answer: `r =`

c.
```
let rec h (y : int) (t : leafy_tree) : leafy_tree =
  begin match t with
  | Leaf x -> Leaf (y + x)
  | Branch (l, r) -> h y l
  end
let r : leafy_tree = h 4 (Branch (Leaf 1, Leaf 2))
```

Answer: `r =`

d.
```
let rec j (t : leafy_tree) : int list =
  begin match t with
  | Leaf x -> [x]
  | Branch (l, r) -> (j l) @ (j r)
  end
let r : int list = j (Branch (Leaf 1, Branch (Leaf 3, Leaf 2)))
```

Answer: `r =`

4. Higher-order function patterns (12 points)

Recall the functions `transform` and `fold` discussed in lecture and used in HW04:

```
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
```

The following recursive functions have been given for you. Rewrite each of them using either `transform` or `fold`.

a.
```
let rec member (elt : int) (x : int list) : bool =
   begin match x with
   | [] -> false
   | h :: t -> h = elt || member elt t
   end
```

```
let member (elt : int) (x : int list) : bool =
```

b.
```
let rec add_ancestor_labels_list (rs: tree list): labeled_tree list =
  begin match rs with
    | [] -> []
    | h :: t -> add_ancestor_labels h :: add_ancestor_labels_list t
  end

     let add_ancestor_labels_list (rs : tree list) : labeled_tree list =
```

5. **Modules and Abstract types (12 points)**

Consider the following module definition

```
module M = struct
  type t = int
  let zero : t = 0
  let incr (x : t) : t = x + 1
  let to_int (x: t) : int = x
  let from_int (x : int) : t = x
end
```

and the following invariant that the module designer would like to maintain

A value of type `M.t` *is never negative.*

Evaluate whether each of the following signatures for `M` could be used to maintain this invariant.

a.
```
type t
val zero  : t
val incr  : t -> t
val to_int : t -> int
val from_int : bool -> t
```

Circle one:

i. This interface prevents all clients from breaking the invariant

ii. A client could break the invariant if `M` used this interface

iii. This interface doesn't match `M` (it would cause a compilation error)

b.
```
type t
val zero : t
val incr : t -> t
val to_int : t -> int
```

Circle one:

i. This interface prevents all clients from breaking the invariant

ii. A client could break the invariant if `M` used this interface

iii. This interface doesn't match `M` (it would cause a compilation error)

c.
```
type t
val zero  : t
val incr  : t -> t
val to_int : t -> int
val from_int : int -> t
```

Circle one:

i. This interface prevents all clients from breaking the invariant

ii. A client could break the invariant if `M` used this interface

iii. This interface doesn't match `M` (it would cause a compilation error)

6. Binary Search Trees (12 points)

Circle either T (true) or F (false) for each statement about binary search trees below. Below, `insert` and `delete` refer to the BST functions that we discussed in class. For reference, these functions appear in the appendix.

a. T F An `Empty` tree satisfies the BST invariant.

b. T F The tree below satisfies the BST invariant.

```
      5
     / \
    2   7
   / \
  1   6
```

c. T F The tree below satisfies the BST invariant.

```
      3
     / \
    2   7
   /
  2
```

d. T F Suppose we are given the following tree `t` that does **not** satisfy the BST invariant:

```
    2
   / \
  5   3
```

Then the expression `(insert (delete 5 t) 5)` will return a tree that satisfies the BST invariant (i.e. a tree that is a BST).

e. T F If you `insert` a number `n` into a BST `t`, and then `delete` `n` from the result, then the resulting tree will always have exactly the same shape and same elements as `t`.

f. T F If you `delete` a number `n` from a BST `t`, and then `insert` `n` into the result, then the final tree will always have exactly the same shape and same elements as `t`.

7. More Binary Search Trees (12 points)

Recall the type of *generic binary search trees*:

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

Implement a function called `scs`, short for *smallest containing subtree*. This function should, when given two values that appear in a binary search tree, return the smallest subtree that contains both of those values.

For example, given the tree

```
t1 =     4
        / \
       2   5
      / \
     1   3
```

the smallest containing subtree of `1` and `3` is

```
t2 =     2
        / \
       1   3
```

Likewise, the smallest subtree of `t1` containing `1` and `2` is also `t2`. On the other hand, the smallest subtree of `t1` that contains both `2` and `5` is the whole tree.

You should assume that the input tree is a binary search tree, that both input values are contained within the tree, and that the first argument is smaller than the second. Your solution does *not* need to detect whether any of these assumptions are violated.

Your implementation *must* take advantage of the binary search tree invariant and must work for *generic* binary search trees. **You may not use any auxiliary functions in your solution, such as `lookup`, `insert`, or `delete`.**

(You may use this page for your implementation of `scs` if needed.)

Appendix - BSTs containing integers

```
type tree =
   | Empty
   | Node of tree * int * tree

let rec lookup (t:tree) (n:int) : bool =
  begin match t with
   | Empty -> false
   | Node(lt, x, rt) ->
      if x = n then true
      else if n < x then lookup lt n
      else lookup rt n
  end

let rec insert (t:int tree) (n:int) : int 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

let rec tree_max (t:tree) : int =
  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 (n:int) (t:tree) : tree =
  begin match t with
   | Empty -> Empty
   | Node(lt,x,rt) ->
      if x = n then
       begin match (lt,rt) with
         | (Empty, Empty) -> Empty
         | (Node _, Empty) -> lt
         | (Empty, Node _) -> rt
         | _ -> let m = tree_max lt in
            Node(delete m lt, m, rt)
       end
      else
       if n < x then Node(delete n lt, x, rt)
       else Node(lt, x, delete n rt)
  end
```