Name: ___________________________
Pennkey: ___________________________

CIS 120 Midterm I
October 12, 2011

1	/25
2	/20
3	/10
4	/25
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.

1. **Program Design (25 points total)**

   An integer list `l1` is a *subsequence* of the integer list `l2` if all the elements of `l1` appear in the same order within the list `l2`, perhaps with other elements between them. For example, `[1;2;2;3]` is a subsequence of `[1;2;4;2;2;3]`. Use the four-step design methodology to implement a function called `subsequence` that determines whether one list is a subsequence of another.

(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 `subsequence` function as you might find it in a `.mli` file or module interface:

```
val subsequence : int list -> int list -> bool
```

(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 =
  true = (subsequence [1;2;2;3] [1;2;4;2;2;3])
;; run_test "comes from the problem description" test
```

Good answers:

```
(1)  true = subsequence [] [1;2;3]
```

Nil is a subsequence of any list.

```
(2)  false = subsequence [1;2;3] []
```

A non-empty list is not a subsequence of Nil

```
(3)  true = subsequence [1;2;3] [2;1;2;3]
```

A subsequence might not start at the head of `l2`

```
(4)  true = subsequence [1;2;] [1;2;]
```

A list is always a subsequence of itself.

(13 points) Step 4 is *implementing the program.* Fill in the body of the `subsequence` function to complete the design. Do *not* use any list library functions (such as `List.map`, `fold`, or `@`) to solve this problem.

```
let rec subsequence (l1:int list) (l2:int list) : bool =
  begin match (l1, l2) with
    | ([], _) -> true
    | (x::xs, y::ys) ->
        (x = y && subsequence xs ys)
        || subsequence l1 ys
    | (x::_, []) -> false
  end

(* or, without matching on tuples *)
let rec subsequence2 (l1:'a list) (l2:'a list ) : bool =
  begin match l1 with
    | [] -> true
    | x::xs ->
        begin match l2 with
          | [] -> false
          | y::ys -> ((x = y) && subsequence xs ys)
                    || subsequence l1 ys
        end
  end
```

2. List Processing (20 points)

Recall the definition of `fold` we saw in class:

```
let rec fold (combine : 'a -> 'b -> 'b) (base:'b) (l:'a list) : 'b =
 begin match l with
   | [] -> base
   | x::tl -> combine x (fold combine base tl)
 end
```

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

a.
```
let rec h (l:int list list) : int list =
  begin match l with
    | [] -> []
    | x::tl -> x@(h tl)  (* @ is the built-in 'append' function *)
  end
let r : int list = h [[1;2;3];[2];[3]]
```

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

b.
```
let rec g (f:'a -> bool) (l:'a list) : 'a list =
  begin match l with
    | [] -> []
    | x::tl -> if (f x) then x::x::(g f tl) else x::(g f tl)
  end
let r : int list = g (fun (x:int) -> x > 2) [1;2;3;4;5]
```

```
[1;2;3;3;4;4;5;5]
```

c.
```
let combine (x:int) (y:int) : int = x + y
let r : int = fold combine 0 [1;2;3;4]
```

```
10
```

d.
```
let m (l:int list) : int option =
  begin match l with
    | [] -> None
    | x::tl -> Some (fold min x tl)
  end
let r : int option = m [3;2;1;4;5]
```

```
Some 1
```

3. Types (10 points)

For each OCaml expression below, write down its type 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.

`int`	`3 + 7`
`int list list`	`[1]::[]`
ill typed	`[]::[1]`
`string -> unit`	`print_string`
ill typed	`print_string::print_int::[]`
`'a -> 'a`	`fun (x:'a) -> x`
`int`	`(fun (x:'a) -> x) (3 + 7)`
`int`	`print_string "hello"; 3 + 7`

`string`	`let x : int = 3 in`
	`let x : string = "hello" in`
	`x`

`'a option`	`None`
ill typed	`42 + (Some 120)`

4. Binary Trees and Binary Search Trees (20 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 elided the Empty nodes from these pictures.)

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

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

```
(b), (c), (e)
```

b. (8 points) Complete this definition of a function that returns the list of nodes of the given tree using an *in order* (left-to-right) tree traversal. For example, the in order traversal of tree `(a)` pictured above is `[1;2;0;3;4]`. You may use the `@` (list append) operator.

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

c. (8 points) Complete this definition of `tree_transform`, which converts an `'a tree` to a `'b tree` by applying a given function `f` to each node of the tree while retaining the tree's shape. (Note that this is analogous to the list `transform` function we saw in class.)

```
let rec tree_transform (f:'a -> 'b) (t: 'a tree) : 'b tree =
 begin match t with
   | Empty -> Empty
   | Node(lt, x, rt) -> Node(tree_transform f lt, f x, tree_transform f rt)
 end
```

d. (4 points) Suppose you know that `t` is an `int tree` that satisfies the binary search tree invariant. Which of the following properties must the function `f : int -> int` have so that `tree_transform f t` also satisfies the BST invariant?

- For every `x` and `y`, if `x <= y` then `(f x) <= (f y)`.
- $\boxed{\text{For every } x \text{ and } y\text{, if } x < y \text{ then } (f\ x) < (f\ y).}$
- For every `x` and `y`, if `x <= y` then `(f y) <= (f x)`
- For every `x` and `y`, if `x < y` then `(f y) < (f x)`.

5. Abstract Types (20 Points total)

Suppose that `S` is a module that implements this interface for the abstract type of sets (taken directly from homework 3):

```
module type Set = sig
  type 'a set

  val empty : 'a set
  val is_empty : 'a set -> bool
  val member : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val equal : 'a set -> 'a set -> bool
  val elements : 'a set -> 'a list
  val fromList : 'a list -> 'a set
  val setSize : 'a set -> int
end
```

(8 points) Indicate whether each of the following is true or false:

a. T [✓ F] A client of the set module could use the following program to determine whether `l` contains `3`:

```
let l : int list = ... (* construct a big int list*)
let answer : bool = S.member 3 l
```

b. T [✓ F] It is possible that the module `S` implements the type `'a set` internally by doing:

```
type 'a set = 'a option
```

c. [✓ T] F Given the interface above, implementing a client function

```
union : 'a S.set -> 'a S.set -> 'a S.set
```

that produces a set containing all of the elements of both of its inputs requires the use of the `S.elements` function.

d. [✓ T] F The `add` function may be defined recursively.

(12 points) Suppose that you have an application that uses the `setSize` operation extremely often—much more often than the `add` and `remove` operations, for example. You therefore need to implement a version of the `Set` module that makes calling `setSize` very efficient.

One way to do that is to use an internal representation of sets that stores the current size of the set along with the elements themselves. We can therefore use the following implementation of the `Set` interface, which represents sets as pairs with an invariant:

> INVARIANT: if the pair `(n, l)` represents a set $\{a_1, \ldots, a_n\}$, then `l` is a list with no duplicates containing the elements $a_1, \ldots, a_n$ in some order, and `n = length l`.

This invariant lets us implement the `setSize` function very efficiently by using the `fst` operation, which projects the first element of a pair (`snd` projects the second). However, we have to adapt the rest of the operations to maintain the new invariants. We have provided the `empty`, `is_empty`, and `add` functions. Complete the implementation of the `remove` function (which is used by `add`).

```
module SizeSet : Set = struct
  type 'a set = (int * 'a list)

  let empty : 'a set = (0, [])

  let is_empty (s:'a set) = (fst s = 0)

(* version 1: one recursive function *)
  let rec remove (x:'a) (s:'a set) : 'a set =
    begin match s with
      | (_,[]) -> (0,[])
      | (n,y::tl) ->
          let r = (n-1,tl) in
          if x=y then r else
            let (m,rest) = remove x r in
              (m+1,y::rest)
    end

(* version 2: helper functions *)
  let rec remove (x:'a) (s:'a set) : 'a set =
    let rec rem_list (l:'a list) : 'a list =
        begin match l with
          | [] -> []
          | h::tl -> if h = x then tl
                     else h::(rem_list tl)
        end
    in
    let elts = (snd s) in
    let size = (fst s) in
    let new_elts = rem_list (snd s) in
    if elts = new_elts then s
    else (size-1, new_elts)

  let rec add (x:'a) (s:'a set) : 'a set =
    let r = remove x s in
      (1+(fst r), x::(snd r))
  ... (* other operations omitted *)

  let setSize (s:'a set) : int = fst s
end
```