CIS 120e Midterm I Review

• This is a review for the midterm. These problems are intended to be indicative of the kind that might appear on the exam, though you should, of course, expect variations.

• Reminder: there will be at least one problem on the exam that is taken verbatim from the programming assignments.

• You will have 50 minutes to complete the actual exam.
1. List Recursion

For each of the following programs, write down the value computed for \( r \):

a. let \( l = [1;2;3] \)
   let rec \( \text{goo} \) (\( l: \text{int list} \)) (\( x: \text{int} \)) : \text{int list} =
   begin
   match \( l \) with
   | \([\]\) -> \([x]\)
   | \( h::t \) -> \( x::h::(\text{goo} \ t \ x) \)
   end
   let \( r = \text{goo} \ l \ 4 \)
   \([4;1;4;2;4;3;4]\)

b. let \( l = [1;2;3] \)
   let rec \( \text{too} \) (\( l: \text{int list} \)) (\( x: \text{int} \)) : \text{int list} =
   begin
   match \( l \) with
   | \([\]\) -> \([x]\)
   | \( h::t \) -> \( x::h::(\text{too} \ t \ h) \)
   end
   let \( r = \text{too} \ l \ 4 \)
   \([4;1;1;2;2;3;3]\)

c. let \( l = [1;2;3] \)
   let rec \( \text{bar} \) (\( l: \text{int list} \)) : \text{int} =
   begin
   match \( l \) with
   | \([\]\) -> failwith "\text{bar doesn’t work on \([\]\)}"
   | \( h::[\] \) -> \( h \)
   | \( h::t \) -> let \( x = \text{bar} \ t \) in
          if \( h > x \) then \( x \) else \( h \)
   end
   let \( r = \text{bar} \ l \)
   \(1\)
2. Generic programming

Implement a non-trivial function foo that has the following type. A function is non-trivial if, given different inputs, it yields different outputs. Hint: your function should be recursive.

\[
\text{foo: } \text{'a list} \to (\text{'a} \to \text{'b}) \to \text{'b list}
\]

Fill in the body of the function below

```ocaml
let rec foo (l:'a list) (f:'a -> 'b) : 'b list =
  begin match l with
    | [] -> []
    | h::t -> (f h)::(foo t f)
  end
```

3. Programming with trees

Suppose you want to generalize binary trees to “arbitrarily branching trees” so that each node can have 
zero or more children and such that nodes can have varying numbers of children.

a. Complete the following type definition so that 'a atree satisfies the description above.

```plaintext
type 'a atree =
  | AEmpty
  | ANode of 'a * ('a atree list)
```

b. Implement the is_leaf function that determines whether a node is a leaf (recall that a leaf is a node, all of whose children are empty). Hint: you may need an auxiliary function that you can define locally to is_leaf.

```plaintext
let is_leaf (t:'a atree) : bool =
  let rec all_empty (l:'a atree list) : bool =
    begin match l with
    | [] -> true
    | AEmpty::r -> all_empty r
    | _    -> false
    end in
  begin match t with
  | AEmpty -> false
  | AEmpty -> false
  | ANode (_, l) -> all_empty l
  end
```
c. Implement a function called `atree_fold` that is the “atree” analog of the `tree_fold` function; complete the following definition by adding the `combine`, `base` data, and `a atree` arguments and then filling in the body. Hint: the library function `List.map` might be useful.

```ocaml
let rec atree_fold (comb:'a -> ('b list) -> 'b) (base: 'b) (t:'a atree) : 'b =
begin match t with
| AEmpty -> base
| ANode (x, l) -> comb x (List.map (atree_fold comb base) l)
end
```
4. Types

For each OCaml expression below, give its type or write “ill typed” if there is a type error.

a. **int list** 3::4::5::[]

b. **int -> int** fun (x:int) -> x + x

c. **'a list option** Some []

d. **ill typed** "a"::3::[]

e. **int** !(ref 3)
5. Combining Collection Types

Circle the composite collection type that would be most appropriate for representing the data structure described in each of the following scenarios.

a. From a given room in an adventure game, it is possible to move in a number of directions (represented as strings), each leading to another room.
   i. (room,string) map
   ii. (string,item set) map
   iii. (string,room) map
   iv. string set

   Answer: iii

b. My favorite game of solitaire is played by laying out a row of 13 stacks of cards with four cards in each stack. The order of the stacks (and the cards in each stack) is important.
   i. card set set
   ii. (card,card list) map
   iii. card list
   iv. card list list

   Answer: iv

c. An address book can be used to find someone’s phone number given their first name and their last name.
   i. (string set,phone) map
   ii. ((string,string) map, phone) map
   iii. (phone,(string,string) map) map
   iv. (string,(string,phone) map) map

   Answer: iv

d. Every year, Sally takes her kids to the worldwide tattoo festival. These days they have their own tattoos to show off, but when they were young she used to play a game with them to keep them quiet: the goal was to find as many different tattoo configurations as possible among the passersby, where a “configuration” is the set of particular body parts that somebody has tattooed.
   i. (bodypart,bodypart set) map
   ii. (int, bodypart set) map
   iii. bodypart set set
   iv. bodypast list list

   Answer: iii
e. The CDDB database is used by iTunes and other music programs to look up the names of the songs on a CD, given the lengths of the tracks. (It is a little surprising that this works, but it turns out that knowing just the lengths of all the tracks, in order, is enough to uniquely identify pretty much any CD!) Since title information can be entered into the database several times (e.g., for international releases), some CDs are listed with several variants of the song names.

1. (int, string list list) map
2. (int list, string list set) map
3. (int,string) map list
4. (int, (int,string) map) map

Answer: ii
6. First-class Functions

Recall the usual definition of generic binary trees. Remember that the function fold_tree is the binary-tree analog of the list-function fold.

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

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

For each of the following functions, choose the combination of base (base) and combine-function (comb) arguments that should be given to fold_tree to implement the desired functionality.

**a. size** computes the total number of nodes in the tree.

```ocaml
let size (t:'a tree) : int = fold_tree comb base t
```

base should be: comb should be:

- []
- 0
- 1
- x

**b. height** computes the longest path from root to any leaf.

```ocaml
let height (t:'a tree) : int = fold_tree comb base t
```

base should be: comb should be:

- []
- 0
- 1
- x
c. in_order computes a list of elements using the *in-order* traversal of the tree.
   let in_order (t:'a tree) : 'a list = fold_tree comb base t
   base should be:  comb should be:
   • 0    • fun (l:'a list) (x:'a) (r:'a list) -> l @ r @ [x]
   • [x]  • fun (l:'a list) (x:'a) (r:'a list) -> l @ [x] @ r
   • []   • fun (l:'a list) (x:'a) (r:'a list) -> x :: (l @ r)
   • [[]] • fun (l:'a list) (x:'a) (r:'a list) -> l @ r

d. pre_order computes a list of elements using the *pre-order* traversal of the tree.
   let pre_order (t:'a tree) : 'a list = fold_tree comb base t
   base should be:  comb should be:
   • 0    • fun (l:'a list) (x:'a) (r:'a list) -> l @ r @ [x]
   • [x]  • fun (l:'a list) (x:'a) (r:'a list) -> l @ [x] @ r
   • []   • fun (l:'a list) (x:'a) (r:'a list) -> x :: (l @ r)
   • [[]] • fun (l:'a list) (x:'a) (r:'a list) -> l @ r

e. post_order computes a list of elements using the *post-order* traversal of the tree.
   let post_order (t:'a tree) : 'a list = fold_tree b comb t
   base should be:  comb should be:
   • 0    • fun (l:'a list) (x:'a) (r:'a list) -> l @ r @ [x]
   • [x]  • fun (l:'a list) (x:'a) (r:'a list) -> l @ [x] @ r
   • []   • fun (l:'a list) (x:'a) (r:'a list) -> x :: (l @ r)
   • [[]] • fun (l:'a list) (x:'a) (r:'a list) -> l @ r

10
7. Abstract Stack Machines

For each of the following programs, draw the state of the abstract stack machine (its workspace, stack and heap) at the point just after a binding for \( x \) has been pushed to the stack. Assume that the program has access to the usual generic binary tree datatype:

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

a. let y = 7 in
   let f (z:int) : int = z + z in
   let x = f y in
   x
   
   no answer available

b. let y = ref Empty in
   let z = ref (Node(!y, 3, Empty)) in
   let _ = y := !z in
   let x = y in
   x
   
   no answer available
```