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 goo (l:int list) (x:int) : int list =
      begin match l with
         | [] -> [x]
         | h::t -> x::h::(goo t x)
      end
   let \( r = \) goo \( l 4 \)

b. let \( l = [1;2;3] \)
   let rec too (l:int list) (x:int) : int list =
      begin match l with
         | [] -> [x]
         | h::t -> x::h::(too t h)
      end
   let \( r = \) too \( l 4 \)
c. let \( l = [1;2;3] \)
    let rec bar (l:int list) : int =
        begin match l with
            | [] -> failwith "bar doesn’t work on []"
            | h::[] -> h
            | h::t -> let x = bar t in
                    if h > x then x else h
        end
    let r = bar l
2. Generic programming

Implement a non-trivial function \texttt{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.

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

Fill in the body of the function below

\[
\text{let rec foo (l:'a list) (f:'a -> 'b) : 'b list =}
\]
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.

   type 'a atree =

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.

   let is_leaf (t:'a atree) : bool =
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
```
4. Types
For each OCaml expression below, give its type or write “ill typed” if there is a type error.

a. ___________________________ 3::4::5::[]

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

c. ___________________________ Some []

d. ___________________________ "a":3::[]

e. ___________________________ !(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

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

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

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

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.
i. \((\text{int, string list list})\) map 
ii. \((\text{int list, string list set})\) map 
iii. \((\text{int, string})\) map list 
iv. \((\text{int, (int, string) map})\) map
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.

```ml
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.

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

base should be:  comb should be:

  • []      • fun (l:int) (_,:'a) (r:int) -> 1 + max l r
  • 0      • fun (l:int) (_,:'a) (r:int) -> 1 + l + r
  • 1      • fun (l:int) (x:int) (r:int) -> x + l + r
  • x      • fun (l:int) (_,:'a) (r:int) -> l + r
```

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

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

base should be:  comb should be:

  • []      • fun (l:int) (_,:'a) (r:int) -> 1 + max l r
  • 0      • fun (l:int) (_,:'a) (r:int) -> 1 + l + r
  • 1      • fun (l:int) (x:int) (r:int) -> x + l + r
  • x      • fun (l:int) (_,:'a) (r:int) -> l + r
```
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
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
```

```ocaml
12
```
b. let y = ref Empty in
   let z = ref (Node(!y, 3, Empty)) in
   let _ = y := !z in
   let x = y in
     x