CS 3110 Fall 2016
Sample problems for Prelim 1 based on Fall 2015

Solutions

Please note that the actual Fall 2015 Prelim 1 was longer than this selection of problems. We have attempted to remove any problems that were about topics not covered this semester, but it's possible we missed some.

1. True–False [20 pts]

   Label the following statements with either "true" (T) or "false" (F). Correct answers receive two points, blank or omitted answers receive 1 point, and incorrect answers receive zero points.

   (a) `let f x y = x+y` and `let f = fun x -> fun y -> x+y` are semantically equivalent.

   True

   (b) `x::x::_` is a pattern that matches any list in which the first two elements are the same.

   False

   (c) `11 + (Some 11)` evaluates to `Some 22`.

   False

   (d) A polymorphic variant type must be defined before it can be used.

   False

   (e) If type `t` is abstract in signature `S`, then a structure with type `S` is not permitted to define `t`.

   False

   (f) A functor takes a signature as input and produces a structure as output.

   False

   (g) A glass-box test is based on the implementation details of a module.

   True

2. Higher-order functions [10 pts]

   (a) [5 pts] Write a function `n_times` such that `n_times f n x` applies `f` to `x` a total of `n` times. That is,

   - `n_times f 0 x` yields `x`

- `n_times f 1 x` yields `f x`
- `n_times f 2 x` yields `f (f x)`
- ...

Answer:

```
let rec n_times f n x =
  if n=0 then x
  else f(n_times f (n-1) x)
```

(b) [5 pts] A *section* is a binary operator that has been partially applied to one argument, which could be either the left or the right argument. For example,

- the function `knightify` that prepends `"Sir "` to a string would be a *left section* of the string concatenation operator; and
- the function `half` that divides a `float` by `2.0` would be a *right section* of the floating-point division operator.

A direct implementation of those two sections could be as follows:

```
let knightify s = "Sir " ^ s
let half x = x /. 2.0
```

Consider writing two functions, `secl` and `secr`, that create the left and right section, respectively, of an operator. We could use those to reimplement our two examples:

```
let knightify = secl "Sir " (^)
let half = secr (/.) 2.0
```

Complete the following definitions of `secl` and `secr`. You may not change the provided code.

```
let secl (a: 'a) (f: 'a -> 'b -> 'c) : 'b -> 'c =
```

Answer:

```
  f a
  (* would also work: fun (b: 'b) -> f a b *)
```

```
let secr (f: 'a -> 'b -> 'c) (b: 'b) : 'a -> 'c =
```

Answer:

```
  fun (a: 'a) -> f a b
```

3. Lists [20 pts]

(a) [12 pts] Consider writing a function `dup: 'a list -> 'a list`, such that `dup lst` duplicates each element of `lst`. For example, given the list `[a;b;c]`,

produce the list `[a;a;b;b;c;c]`. Duplicates must remain in the original order: `dup [a;b;c]` is not `[a;b;c;a;b;c]`.

Implement this function in three ways:

i. `dup_rec`: recursively, without using any `List` module functions;

ii. `dup_fold`: using `List.fold_left` or `List.fold_right`, but no other `List` module functions nor the `rec` keyword; and

iii. `dup_lib`: using any combination of `List` module functions, but excluding any `fold` functions and the `rec` keyword.

The Appendix provides the names and types of many `List` module functions.

Neither time nor space efficiency is a concern. Your solutions do not need to be tail recursive.

Answer:

```
let rec dup_rec = function
  | [] -> []
  | h::t -> h::h::dup_rec t

let dup_fold lst =
  List.fold_right (fun elt acc -> elt::elt::acc) lst []

let dup_lib lst =
  lst
  |> List.map (fun elt -> [elt;elt])
  |> List.flatten
```

(b) [3 pts] For each of your three implementations above, explain why the implementation is or is not tail recursive. Note that functions in the Appendix are tail recursive unless specified otherwise.

Answer:

None of those three is tail recursive.

i. Still has work to do after the recursive call: prepending the elements onto the returned list.

ii. `List.fold_right` *is not tail recursive.*

iii. Neither `List.flatten` *nor* `List.map` *is tail recursive.*

(c) [5 pts] Write a function `remove_dups: 'a list -> 'a list` such that `remove_dups xs` is all the unique elements of `xs`. The order of elements in the returned list does not matter. For example, `remove_dups [1;2;3;1;4;3;4;5;1]` could be `[1;2;3;4;5]`, or `[5;4;3;2;1]`, etc. You may implement the function in any way you see fit.

Answer:

```
(* [diff xs z] is the list [xs] with all
 * occurrences of [z] removed. *)
let rec diff xs = function
  | [] -> xs
  | y::ys' -> diff (List.filter (fun x -> x <> y)
                                  xs) ys'

let rec remove_dups = function
  | [] -> []
  | x::xs' -> x :: (remove_dups (diff xs' [x]))

(* OR *)

let remove_dups lst =
  let srt = List.sort compare lst in
  List.fold_left (fun acc elt -> match acc with
      | [] -> [elt]
      | h::_ when elt = h -> acc
      | h::_ -> elt::acc) [] srt

(* OR *)

let remove_dups lst =
  List.fold_left
    (fun acc elt ->
      if (List.mem elt acc) then acc else elt::acc)
    [] lst
```

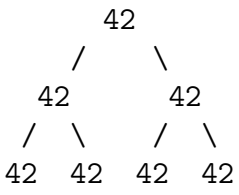
4. Trees and Modules [30 pts]

   Here is a variant that represents binary trees:

```
type 'a bintree =
  | Nil
  | Node of 'a * 'a bintree * 'a bintree
```

   A *perfect binary tree* is a binary tree in which all nodes have two children—except the leaves (i.e., `Nil`), which never have any children—and all leaf nodes are at the same depth in the tree.

   (a) [4 pts] Write a function `perfect : int -> 'a -> 'a bintree` such that `perfect i x` is the perfect binary tree with $2^i - 1$ nodes, where each node contains the value `x`. For example, `perfect 3 42` would produce a value representing the following tree:



Answer: Why not Node(x Nil Nil)

```
let rec perfect d x =
  if d = 0 then Nil
  else Node(x, perfect (d-1) x, perfect (d-1) x)
```

(b) [4 pts] Write a function `is_perfect : 'a bintree -> bool` such that `is_perfect t` returns `true` if and only if `t` is a perfect binary tree. For full credit your solution must run in time that is linear in the number of nodes in the tree.

Text

Answer:

```
(* omitted from these solutions *)
```

A *functional array* is a persistent data structure that maps a finite range of integers to elements. Here is an interface for functional arrays:

```
module type FUN_ARRAY = sig
  (* The type of an array. Given an array [a], we
     write [a[i]] to denote the element at index [i]. *)
  type 'a t

  (* Raised when an operation is asked to access an
     element at an index that is not bound in an array. *)
  exception OutOfBounds

  (* [make i x] creates an array with indices
     ranging from 1 to (2^i)-1, inclusive,
     all of which map to the element [x].
     (^) denotes integer exponentiation here. *)
  val make : int -> 'a -> 'a t

  (* [get i a] is an element of [a]. *)
  val get : int -> 'a t -> 'a

  (* [set i x a] is an array [b] such that [b[i]=x] and
     [b[k] = a[k]] for all [i <> k].
     Raises [OutOfBounds] if [i] is an index not bound
     in [a]. *)
  val set : int -> 'a -> 'a t -> 'a t
end
```

(c) [3 pts] What does it mean that a functional array is *persistent*? Your answer should incorporate the type of `set`.

Answer:
A persistent aka functional data structure is not mutated by update operations. Instead, those operations create new copies of the data structure, and old copies stay around—i.e., they persist. We can see that in the type of `set`*, because it takes an old data structure as its final argument and returns a new data structure.*

(d) [4 pts] The designer of `get` intended it to behave like an array indexing operation. Assume the following representation type:

```
type 'a t = 'a list
```

Write an implementation of `get` that satisfies the documented specification but is not what the designer intended.

Answer:

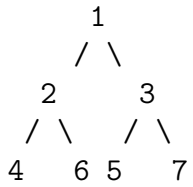
```
let get i a = List.hd a
```

(e) [3 pts] Write a good specification for `get`.

Answer:

```
(* [get i a] is [a[i]].
   Raises [OutOfBounds] if [i] is an index not bound
   in [a]. *)
```

You will now implement `FUN_ARRAY` using `'a bintree` as the representation type. The idea is that the tree must be perfect, and that each node in the tree represents an index, and the value stored at that node is the element mapped by the index. To figure out which node represents index `k`, start at the root and repeatedly divide `k` by 2 until it is reduced to 1. Each time the remainder equals 0, move to the left subtree; if the remainder equals 1; move to the right. For example, index 6 is reached by left, right:



Let's assume that the definitions of `'a bintree` and `perfect` are in scope. Here is a start at implementing a module for functional arrays:

```
module FunArray : FUN_ARRAY = struct
  type 'a t = 'a bintree
  exception OutOfBounds
  let make = failwith "TODO"
  let get  = failwith "TODO"
```

```
  let set  = failwith "TODO"
end
```

(f) [2 pts] Write a comment documenting the representation invariant (if any) for `'a FunArray.t`.

Answer:

QUESTION

```
(* RI:  The tree must be perfect. *)
```

Note that anything about how the tree represents an array (e.g., where values must be according to their index) properly belongs in the AF, not the RI.

(g) [2 pts] Implement `FunArray.make`.

Answer:

```
let make = perfect
```

(h) [8 pts] Implement `FunArray.set`.

Answer:

```
(* omitted from these solutions *)
```

```
let preorder t = foldtree [] (fun x l r -> [x] @ l @ r) t
let rec foldtree init op = function
  | Leaf -> init
  | Node (v,l,r) -> op v (foldtree init op l) (foldtree init op r)


val get : int -> 'a t -> 'a
val set : int -> 'a -> 'a t -> 'a t
let set i x tree =
if x = 1 then .. match against tree if leaf node insert otherwise overwrite and return
else if x mod 2 = 1...
else ...
```

```
(* [set i x a] is an array [b] such that [b[i]=x] and [b[k] = a[k]] for all [i <> k].
Raises [OutOfBounds] if [i] is an index not bound in [a]. *)
```