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

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.

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

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

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

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

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

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

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)`
- ...

(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 =
```

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

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.

(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.

(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.

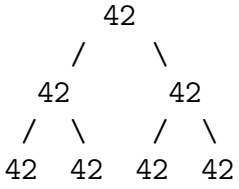
4. Trees and Modules [30 pts]

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:



(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.

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`.

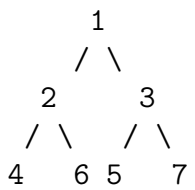
(d) [4 pts] The designer of `get` intended it to behave like an array indexing operation. Assume the following representation type:

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

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

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

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`.

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

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