CS 3110 – Data Structures and Functional Programming
Example Problems for Prelim 1 based on Spring 2014

Exercise 1: OCaml Programming (10 points)

Fill in each box with a value that causes the entire expression to evaluate to `42`.

(a)
```
let zardoz = List.fold_left [          ] 0 in
    zardoz [1; 2; 3; 4; 5; 6] + zardoz [6; 5; 4; 3; 2; 1]
```

(b)
```
let yolt f x = f (f x) in
    yolt ((+) 20) [          ]
```

(c)
```
let rec tarski o =
    match o with
    | None ->
      tarski (Some 3110)
    | Some x ->
      if x > 20 then x
      else tarski (Some (x + x + x)) in
  tarski [          ]
```

(d) `type point3 = {x : int; y : int; z : int}`

```
let descartes r = r.x * r.y in
```

```
descartes [          ]
```

⋆ **(e)**
```
let haskell f (x,y) = f x y in
let rec peano x y = match y with
  | 0 -> x
  | _ -> peano (x + 1) (y-1) in
```

```
haskell (*) (haskell peano (1, 2), [          ] )
```

Exercise 2: Types (10 points)

Write down the types of each of the following OCaml expression. For full credit, you should write the most general type possible. For example, given `(fun x -> x)` you should write `'a -> 'a` rather than `int -> int` or `string -> string`.

(a) `(fun _ -> 42)`

(b) `((=) 3)`

(c) `let g x y = Some (x + y) in g`

(d) `let h l = List.fold_right (fun x a -> x::a) l l in h`

⋆ **(e)** `let rec l x = l x in l`

Exercise 3: Fold it! (15 points)

Implement each of the following functions on lists using `fold_left` or `fold_right`. You may assume that the `List` module has already been opened in the current scope. Note that you must *not* use recursion in your solutions.

(a) Write a function that computes the length of a list:

```
let length (l:'a list) : int =

  fold_
```

(b) Write a function that given a list `[a0; a1; ...; an]` produces a list `[f a0; f a1; ...; f an]`:

```
let map (f:'a -> 'b) (l:'a list) : 'b list =
  fold_
```

(c) Write a function that returns `None` when applied to the empty list, and `Some x` when applied to a non-empty list, where `x` is the last element of the list:

```
let last (l:'a list) : 'a option =
  fold_
```

(d) Write a function that reverses the elements of a list:

```
let reverse (l:'a list) : 'a list =
  fold_
```

⋆ **(e)** Write function that computes the run-length encoding of a list—that is, a list of integer-element pairs indicating the number of times elements appear consecutively in the input. For example:

```
  encode ['a'; 'a'; 'b'; 'c'; 'c'; 'c'; 'a'] =
         [(2,'a'); (1,'b'); (3, 'c'); (1; 'a')]
```

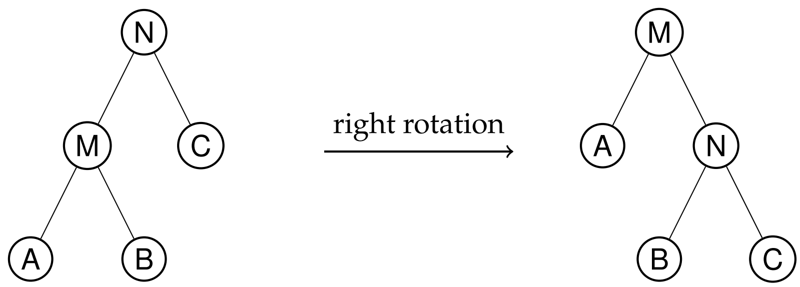
```
let encode (l:'a list) : (int * 'a list) =
  fold_
```

Exercise 4: Rotations (10 points)

One way to represent a binary tree is using the datatype:

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

In some applications (for example, balanced binary search trees) it is necessarily to rotate a tree around the root. In general, a rotation rearranges the structure of the tree, but maintains the order of the leaves according to an in-order traversal.



A rotation should only affect a trees whose left subtree is a node and not a leaf. Rotating all other trees behaves like the identity.

Write a function `right_rotate` that rotates a binary tree around the root:

```
let right_rotate (t:'a tree) : 'a tree =
```

Exercise 5: Higher-Order Programming (15 points)

We've seen that the concept of a fold can be generalized from lists to other data types. For example, the type of the fold function for the `'a tree` type defined in the last exercise is as follows:

```
fold : 'b -> ('b -> 'a -> 'b -> 'b) -> 'a tree -> 'b
```

where

```
fold init f Leaf = init
```

and

```
fold init f (Node (l,x,r)) = f (fold init f l) x (fold init f r)
```

Somewhat surprisingly, we can represent data types such as trees *just* using fold functions:

```
type 'a fold_tree = 'b -> ('b -> 'a -> 'b -> 'b) -> 'b
```

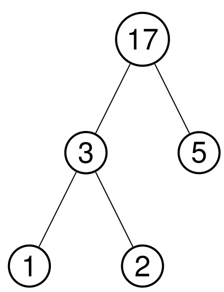
With this representation using higher-order functions, a leaf is represented by a constant function that returns the initial value,

```
let mkleaf = (fun init f -> init)
```

while a node with value `x`, left subtree `l`, and right subtree `r` is represented by a function that first supplies `f` and `init` to the left and right subtrees, and then combines the results with `x` using `f`:

```
let mknode l x r = (fun init f -> f (l init f) x (r init f))
```

For example, we can represent the following tree



as

```
let example_tree : int fold_tree =
  (fun init f ->
    let l = f (f init 1 init) 3 (f init 2 init) in
    let r = f init 5 init in
    f l 17 r)
```

As a final example, to compute the size of a `fold_tree`, we can use the following function:

```
let size (t: 'a fold_tree) : int =
  t 0 (fun l x r -> l + 1 + r)

# size example_tree;;
- : int = 5
```

(a) Write a function that computes the height of a `fold_tree`. By convention, the height of a leaf node should be considered `0`.

```
# height example_tree;;
- : int = 3
```

```
let height (t:'a fold_tree) : int =
```

(b) Write a function that produces an optional value containing the largest value in a `fold_tree`, or `None` if the tree is empty.

```
# max example_tree;;
- : int = Some 17
```

```
let max (t:'a fold_tree) : int option =
```

(c) Write a function that returns a list containing the values in a `fold_tree` according to an in-order traversal:

```
# in_order example_tree;;
- : int list = [1; 3; 2; 17; 5]

let in_order (t:'a fold_tree) : 'a list =
```

⋆ **(d)** Write a function that tests if a `fold_tree` represents a binary search tree. Recall that a binary search tree has the property that at each node, all values in the left subtree are less than or equal to the value at the node, and all values in the right subtree are greater than the value at the node:

```
# is_bst example_tree;;
- : bool = false

let is_bst (t:'a fold_tree) : bool =
```

Exercise 6: Modules and Functors (10 points)

Your problem set is due in a few hours. Your partner has given you an interface for a tree module he's writing.

```
module type TREE = sig
  type 'a tree

  (* [add x t] adds an element [x] to the tree [t] *)
  val add : 'a tree -> 'a -> 'a tree

  (* [root t] returns [Some x] if the tree [t] is a non-empty and has
   * root [x] and [None] otherwise. *)
  val root : 'a tree -> 'a option

  (* [remove x t] removes element [x] from the tree [t] *)
  val remove : 'a tree -> 'a -> 'a tree

  (* [member x t] returns [true] if [x] is an element of
   * tree [t] and [false] otherwise *)
  val member : 'a tree -> 'a -> bool

  (* [size t] returns the number of elements in the tree [t] *)
  val size : 'a tree -> int
end
```

Unfortunately, you do not have access to the module implementation, but a week ago he sent you his first draft:

```
module Tree : TREE = struct
  (** root is t.(0);
      left of t.(i) is t.(2i);
      right of t.(i) is index t.(2i+1) *)
  type 'a tree = 'a array

  (* TODO: finish *)
end
```

To finish the assignment, you need a function

```
exists : ('a -> bool) -> 'a Tree.tree -> bool
```

such that `exists p t` traverses a `Tree.t t` and returns true if and only if the predicate `p` returns `true` on *some* node. Unfortunately, your partner is taking their 2110 prelim and cannot be reached.

Write the `exists` function inside of your own module `M` using only the `TREE` interface, or briefly explain why this task is impossible.

```
module M = functor(Tree:TREE) ->
struct
  let rec exists (p:'a -> bool) (t:'a Tree.tree) =
```

```
end
```