CSE 130, Spring 2012
Instructor: Ranjit Jhala

Name/ID _____

Midterm Exam

Instructions: read these first!

Do not open the exam, turn it over, or look inside until you are told to begin.

Switch off cell phones and other potentially noisy devices.

Write your *full name* on the line at the top of this page. Do not separate pages.

You may refer to any printed materials, but *no computational devices* (such as laptops, calculators, phones, iPads, friends, enemies, pets, lovers).

Read questions carefully. Show all work you can in the space provided.

Where limits are given, write no more than the amount specified.
The rest will be ignored.

Avoid seeing anyone else's work or allowing yours to be seen.

Do not communicate with anyone but an exam proctor.

If you have a question, raise your hand.

When time is up, stop writing.

The points for each part are rough indication of the time that part should take.

Run LaTeX again to produce the table

1. [**??** points]

   (a) [5 points] What type does Ocaml infer for `(<.>)`

   ```
   let (<.>) = fun f g x -> f (g x)
   ```

   _____

   (b) [5 points] What does the following expression evaluate to? (The `<.>`) is as defined for the previous part.)

   ```
   let foo = fun x -> x + 10 in
   let bar = fun x -> x * 10 in
   let goo = foo <.> bar     in
   goo 0
   ```

   _____

   (c) [10 points] Write a tail-recursive function whose behavior is identical to the function below:

   ```
   let rec giftList l = match l with
     | []    -> "that's what I want for Christmas!"
     | g::l' -> g ^ " and " ^ giftList l'
   ```

   _____

   _____

   _____

   _____

   _____

2. [**??** points]

In this question, we have given you three concrete recursive functions. Your goal is to write *generalized* (polymorphic, higher-order) versions of the functions that encapsulate the recursion pattern and make it easily reusable.

(a) [4 points]
```
let rec getEven xs = match xs with
  | []     -> None
  | x::xs' -> if (x mod 2 = 0)
              then Some x
              else getEven xs'
```

What is the type of `getEven` ? (Recall: `type 'a option = None | Some of 'a`)

_____

(b) [6 points] Write a function:
`val find_first: ('a -> bool) -> 'a list -> 'a option`
such that `getEven` is *equivalent to* `findFirst (fun x -> x mod 2 = 0)`

```
let rec find_first f xs = match xs with

  | []     -> _____

  | x::xs' -> _____

              _____

              _____
```

(c) [3 points] Consider the tree type
```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

What is the type of the following function `tree_to_string`

```
let rec tree_to_string t = match t with
  | Leaf          -> ""
  | Node(x, l, r) -> let ls = tree_to_string l in
                     let rs = tree_to_string r in
                     ls ^ ", " ^ x ^ ", " ^ rs
```

_____

(d) [7 points] Write a function

```
val post_fold: ('a -> 'b -> 'b -> 'b) -> 'b -> 'a tree -> 'b
```

such that `tree_to_string` is *equivalent to*

```
post_fold (fun x ls rs -> ls ^ ", " ^ x ^ ", " ^ rs) ""
```

```
let rec post_fold f b t = match t with

| Leaf         -> _____________________________________________

| Node (x,l,r) -> _____________________________________________

                  _____________________________________________

                  _____________________________________________
```

(e) [10 points] Write a function

```
val in_fold: ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a
```

such that `tree_to_string` is *equivalent to* `in_fold (fun str x -> str ^ ", " ^ x ^ ", ") ""`

```
let rec in_fold f b t = match t with

| Leaf         -> _____________________________________________

| Node (x,l,r) -> _____________________________________________

                  _____________________________________________

                  _____________________________________________
```

3. [**??** points]

Consider the following datatype

```
type ('a, 'b) either = Left of 'a | Right of 'b
```

The `either` type is a generalization of `option` which instead of just representing the *junk* value as `None`, allows you to attach some information of type `'a`.

(a) [2 points] Write down an expression that has the type

`(int, string) either`

and also has the type

`(int, bool) either`

(Say what?! How can an expression have two types? Remember `[]` can have type `int list` and also have type `string list`.)

_____

(b) [2 points] Write down an expression that has the type

`(int, string) either`

but does not have the type

`(int, bool) either`

_____

(c) [6 points] Write a function

`val assoc: 'k -> ('k * 'v) list -> ('k, 'v) either`

The function should have the following behavior:

```
# assoc "z" [("x", 1); ("y", 2); ("z", 3); ("z", 4)] ;;
- : (string, int) either = Right 3

# assoc "z" [("x", 1); ("y", 2)] ;;
- : (string, int) either = Left "z"

let rec assoc key kvs = match kvs with

| []          -> _____

| (k,v)::kvs' -> _____

                _____

                _____
```

(d) [7 points] Write a function

```
val map: ('b -> 'c) -> ('a, 'b) either -> ('a, 'c) either
```

The function should have the following behavior:

```
# map (fun i -> i + 1) (Left "x")
- : (string, int) either = Left "x"
# map (fun i -> i + 1) (Right 12)
- : ('a, int) either = Right 13
# map string_of_int (Right 12)
- : ('a, string) either = Right "12"
```

```
let map f e =
```

(e) [8 points] Write a function

```
val map2: ('b ->'c ->'d) -> ('a,'b) either -> ('a,'c) either -> ('a,'d) either
```

The function should have the following behavior:

```
# map2 (+) (Left "x") (Left "y") ;;
- : (string, int) either = Left "x"
# map2 (+) (Left "x") (Right 12) ;;
- : (string, int) either = Left "x"
# map2 (+) (Right 12) (Left "x") ;;
- : (string, int) either = Left "x"
# map2 (+) (Right 12) (Right 7)  ;;
- : ('a, int) either = Right 19
```

```
let map2 f e1 e2 =
```

4. [**??** points]

Consider the following (subset) of Nano-ML.

```
type binop = Plus | Div
type expr = Const of int | Var of string | Bin of expr * binop *  expr
```

As you doubtless know, the evaluation of certain expressions can result in errors such as unbound variables and divide-by-zeros. Instead of using exceptions as (in the programming assignment) we will track errors via the type:

```
type error = UnboundVariable of string | DivideByZero
```

(a) [5 points] Use the function `assoc` above to write a function
`val lookup: string -> (string * int) list -> (error, int) either`
The function should have the following behavior:

```
# lookup "z" [("x", 1); ("y", 2); ("z", 3); ("z", 4)];;
- : (error, int) either = Right 3
# lookup "z" [("x", 1); ("y", 2)] ;;
- : (error, int) either = Left (UnboundVariable "z")
```

```
let lookup x env = _____________________________________________

                   _____________________________________________

                   _____________________________________________

                   _____________________________________________
```

(b) [5 points] Write a function
`val safeDiv: int -> int -> (error, int) either`
Such that `safeDiv n m` returns the integer result `n/m` when it is defined (i.e. when `m` is not zero) and returns the divide-by-zero error otherwise. Concretely, your function should have the following behavior:

```
# safeDiv 40 2;;
- : (error, int) either = Right 20
# safeDiv 40 0;;
- : (error, int) either = Left DivideByZero
```

```
let safeDiv n m = _____________________________________________

                  _____________________________________________

                  _____________________________________________
```

(c) [10 points] Use the function `lookup` to write a function

```
val eval: (string * int) list -> expr -> (error, int) either
```

The function should have the following behavior:

```
# let e0 = Bin (Var "x", Plus, Var "y");;

# let e1 = Bin (e0, Div, Var "y");;
# eval [("x", 1); ("y", 2)] e1;;
- : (error, int) either = Right 1

# let e2 = Bin (e1, Div, Var "z");;
# eval [("x", 1); ("y", 2)] e2;;
- : (error, int) either = Left (UnboundVariable "z")

# let e3 = Bin (e1, Div, Const 0);;
# eval [("x", 1); ("y", 2)] e3;;
- : (error, int) either = Left DivideByZero
```

```
let rec eval env e = match e with

  | Const i           -> _____________________________________________

  | Var v             -> _____________________________________________

  | Bin(e1, Plus, e2) -> _____________________________________________

  | Bin(e1, Div, e2)  -> _____________________________________________
```