CSE 130, Spring 2014
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 a double-sided "cheat sheet", 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.

Question	Points	Score
1	35	
2	20	
3	25	
Total:	80	

1. [35 points] For each of the following OCaml programs, write down the **type** or **value** of the given variable, or circle "Error" if you think there is a type or runtime error.

(a) [5 points]

```
let ans =
  let x       = 10        in
  let f y z = x + y + z in
  let x       = 100       in
  let h       = f 5       in
  h x
```

Error **Value** ans = ______________________

(b) [6 points]

```
let rec chain fs = match fs with
  | []     -> fun x -> x
  | f::fs' -> fun x -> f (chain fs' x)
```

Error **Type** chain : ______________________

(c) [5 points]

```
let ans = chain [ (fun x -> x  * x)
                ; (fun x -> 16 * x)
                ; (fun x -> x + 1)
                ] 1
```

Error **Value** ans = ______________________

(d) [3 points]

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

let ans0 = Node (2, Node (1, Leaf, Leaf)
                  , Node (3, Leaf, Leaf))
```

Error **Type** ans0 : ______________________

(e) [5 points]

```
let rec flerb xs = match xs with
  | []     -> Leaf
  | x::xs' -> Node (x, Leaf, flerb xs')
```

Error **Type** flerb : ______________________

(f) [3 points]

```
let ans = flerb [0;1;2]
```

Error **Value** `ans` = ____________________

(g) [5 points]

```
let rec glub f t = match t with
  | Leaf         -> Leaf
  | Node (x,l,r) -> Node (f x, glub f l, glub f r)
```

Error **Type** `glub` : ____________________

(h) [3 points]

```
let ans = glub (fun x -> 2 * x) ans0
```

Error **Value** `ans` = ____________________

2. [20 points] Consider the two functions `sum` and `fac` shown below:

```
let rec sum n = match n with
  | 0 -> 0
  | n -> n + sum (n-1)

let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n-1)
```

(a) [5 points] Write a **tail recursive** version of `sum` by filling in the blanks below:

```
let sumTR n =

  let rec helper acc n = ________________________

    | ______________ -> ____________________________

    | ______________ -> ____________________________

  in

    in helper ______________________________________
```

(b) [5 points] Write a **tail recursive** version of `fac` by filling in the blanks below:

```
let facTR n =

  let rec helper acc n = ________________________

    | ______________ -> ____________________________

    | ______________ -> ____________________________

  in

    in helper ______________________________________
```

(c) [6 points] *Spot that pattern!* Now write a higher-order function

```
val foldn : ('a -> int -> 'a) -> 'a -> int -> 'a
```

that generalizes the tail-recursion in `sumTR` and `facTR`, by filling in the blanks below:

```
let foldn f b n =
  let rec helper acc n = ______________________
    | _____________ -> ___________________________
    | _____________ -> ___________________________
  in
    in helper ____________________________________
```

(d) [4 points] Your solution for `foldn` should be such that you can now implement `sum` and `fac` *without recursion* simply by passing in appropriate parameters to `foldn`. What are those parameters?

```
let sum = foldn ___________________________ _____________

let fac = foldn ___________________________ _____________
```

3. [25 points]

In NanoML, we used exceptions at various places, for example, when looking up a variable that did not exist in the environment. A better approach is to use the `'a option` type, defined thus:

```
type 'a option = None | Some of 'a
```

(a) [4 points] Now, instead of throwing an exception (who knows *where* or *how* or even *if* it will get caught!) if a function can possibly fail, we can have it return an `option` value. For example:

```
let safeDiv num den = match den with
  | 0  -> None
  | _  -> Some (num / den)
```

Since division is undefined (and throws a nasty failure), we instead write a `safeDiv` that gracefully returns a `None` if the result is undefined, and `Just i` when the denominator is non-zero and hence the division is safe. What is the **type** of `safeDiv`?

Error **Type** `safeDiv`: ______________________________

(b) [5 points] Fill in the blanks to write a version of `lookup` that returns an `option`, that is:

```
val lookup: 'a -> ('a * 'b) list -> 'b option
```

When you are done, you should get the following behavior:

```
# lookup "a" [("a", 1); ("b", 2), ("a", 10)];;
- : int option = Some 1

# lookup "z" [("a", 1); ("b", 2), ("a", 10)];;
- : int option = None
```

```
let rec lookup k kvs = ____________________

    | ________________ ->

        ________________________________________

    | ________________ ->

        ________________________________________
```

(c) [4 points] Fill in the blanks to write a function

```
val lift1 : ('a -> 'b) -> 'a option -> 'b option
```

such that when you are done, you get the following behavior

```
# lift1 string_of_int (Some 1);;
- : string option = Some "1"

# lift1 string_of_int None;;
- : string option = None
```

```
let lift1 f xo = ______________________

    | __________________ -> ______________________

    | __________________ -> ______________________
```

(d) [5 points] Fill in the blanks to write a function

```
val lift2 : ('a -> 'b -> 'c) -> 'a option -> 'b option -> 'c option
```

such that when you are done, you get the following behavior

```
# lift2 (+) (Some 1) (Some 10);;
- : int option = Some 11

# lift2 (+) (None) (Some 10);;
- : int option = None

# lift2 (+) (Some 1) (None);;
- : int option = None

# lift2 (+) (None) (None);;
- : int option = None
```

```
let lift2 f xo yo = ______________________

    | __________________ -> ______________________

    | __________________ -> ______________________
```

(e) [7 points] Consider the subset of NanoML given by the type:

```
type expr = Var  of string          (* variable                 *)
          | Con  of int             (* constant                 *)
          | Neg  of expr            (* negation of an expression *)
          | Plus of expr * expr     (* sum of two expressions    *)
```

Write an interpreter function

```
val eval : (string * int) list -> expr -> int option
```

such that when you are done you get the following behavior:

```
# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Var "y"));;
- int option = Some 3

# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Con 20));;
- int option = Some 21

# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Var "z"));;
- int option = None

# eval [("x", 1); ("y", 2); ("x", 100)] (Neg (Var "y"));;
- int option = Some (-2)

# eval [("x", 1); ("y", 2); ("x", 100)] (Neg (Var "z"));;
- int option = None
```

Note: Your implementation of `eval` must not use any `match-with` expressions other than the one given. Instead, use `lift1` and `lift2`. You may also use any other functions you have implemented during this exam.

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

    | Var x         -> ______________________________________

    | Con i         -> ______________________________________

    | Neg e'        -> ______________________________________

    | Plus (e1, e2) -> ______________________________________
```