CSE 130, Spring 2015
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 *hand-written or printed cheat sheets*, 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	20	
2	15	
3	15	
Total:	50	

1. [20 points] For each of the following OCaml programs, write down the **Value** of the given variable, or circle **Error** if you think there is a type or run-time error.

   (a) [5 points]

```
let ans =
  let x  = 0               in
  let a1 = let x = 1       in
           fun y z -> [x;y;z]
  in
  let a2 = let x = 100     in
           a1 x
  in
  a2 x
```

   Error **Value** `ans` = _____

   The next two parts share the following type and function definition:

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

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

let t0 = Node ( "cat"
              , Node ("dog"  , Leaf, Leaf)
              , Node ("hippo", Leaf, Leaf))
```

   (b) [4 points]

```
let ans =
  let f = (fun _ vl vr -> 1 + vl + vr) in
  fold f 0 t
```

   Error **Value** `ans` = _____

   (c) [4 points]

```
let ans =
  let f = (fun x vl vr -> vl ^ x ^ vr) in
  fold f "" t0
```

   Error **Value** `ans` = _____

The next two parts share the following type and function definition:

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

let rec find f xs = match xs with
  | []       -> None
  | (x::xs') -> if f x then
                  Some x
                else
                  find f xs'

let xs0 = [2;4;8;16;32]
```

(d) [4 points]

```
let ans = let f x = x > 10 in
          find f xs0
```

Error **Value** `ans` = _____

(e) [3 points]

```
let ans = let f x = (x mod 2) = 1 in
          find f xs0
```

Error **Value** `ans` = _____

2. [15 points]

For this problem, you will write some functions that: use Ocaml's *lists* to implement a *Set* API. We will *represent sets of values of type* `'a` by using lists.

```
type 'a set = Set of 'a list
```

(a) [2 points] First, implement a function

```
val empty : 'a set
```

by filling in the definition below

```
let empty = = ____________________________
```

(b) [5 points] Write a function

```
val member : 'a -> 'a set -> bool
```

such that `member x s` returns `true` if `x` is in the set corresponding to `s` and `false` otherwise.

```
let member x s = match s with
  | _______________ -> ____________________________________________
  | _______________ -> ____________________________________________
```

(c) [3 points] Write a function

```
val add : 'a -> 'a set -> 'a set
```

such that `add x s` returns a set which has all the elements of `s` *and also* the element `x`.

```
let add x s = match s with
  | _______________ -> ___________________________________
```

We can use `add` to obtain a function

```
val union : 'a set -> 'a set -> 'a set
```

such that `union s1 s2` returns a new set which has all the elements of `s1` *and also* the elements of `s2`.

```
let union s1 s2 = match s2 with
  | Set x2s -> List.fold_left (fun s x -> add x s) s1 x2s
```

(d) [5 points] Finally, write a function

```
val del : 'a -> 'a set -> 'a set
```

such that `del x s` contains all the elements of `s` *except* the element `x`. **Hint:** Use `List.filter`.

```
let del x s = match s with

  | _____________ -> ____________________________________
```

When you are done, you should see the following behavior at the Ocaml prompt:

```
# let s0 = empty    ;;
# (mem 1 s0, mem 2 s0) ;;
- : bool * bool = (false, false)

# let s1 = add 1 s0 ;;
# (mem 1 s1, mem 2 s1) ;;
- : bool * bool = (true, false)

# let s2 = add 2 s1 ;;
# (mem 1 s2, mem 2 s2) ;;
- : bool * bool = (true, true)

# let s3 = union s1 s2 ;;
# (mem 1 s3, mem 2 s3) ;;
- : bool * bool = (true, true)

# let s4 = del 1 s3 ;;
# (mem 1 s4, mem 2 s4) ;;
- : bool * bool = (false, true)
```

3. [15 points]

Consider the following *small subset* of NanoML:

```
type binop = Plus

type expr  = Const of int
           | Var of string
           | Bin of expr   * binop * expr
           | Let of string * expr  * expr
           | App of expr   * expr
           | Fun of string * expr
```

Well-formed Expressions: The following expressions `e1`, `e2`, `e3` are *good* in that all the variables that are *used* are defined i.e. *bound* in the expression:

```
(* e1 === 1 + 2 *)

let e1 = Bin (Const 1, Plus, Const 2)

(* e2 === let x = 1 in
          let y = 2 in
          x + y        *)

let e2 = Let ("x", Const 1,
         Let ("y", Const 2,
         Bin (Var "x", Plus, Var "y")))

(* e3 === let x = 10 in
          (fun y -> x + y) x *)

let e3 = Let ("x", Const 10,
         App (Fun ("y", Plus (Var "x", Plus, Var "y"))
             ,Var "x"))
```

Ill-formed Expressions: However, the following expressions `e1'`, `e2'` and `e3'` are *bad* because they contain undefined (or "unbound" variables). That is, if you try to evaluate them in an *empty* environment (i.e. run `eval ([], e)`) you will get a `"variable not bound"` error:

```
(* e1' === 1 + x *)

let e1' = Bin (Const 1, Plus, Var "x")

(* e2' === let y = 2 in
           x + y        *)

let e2' = Let ("y", Const 2,
          Bin (Var "x", Plus, Var "y"))
```

```
(* e3' === (let z = 10 in
              (fun y -> y + z)) z  *)

let e3' = App (Let ("z", Const 10,
                 Fun ("y", Plus (Var "y", Plus, Var "z")))
               ,Var "z")
```

(a) [12 points] Use `empty`, `add`, `union` and `del` to write a function

```
val free : expr -> string set
```

such that `free e` returns the set of *free variables* in an expression `e`.

```
let rec free e = match e with

  | Var x           -> _____________________________________

  | Const n         -> _____________________________________

  | Bin (e1, op, e2) -> _____________________________________

  | App (e1, e2)    -> _____________________________________

  | Let (x, e1, e2)  -> _____________________________________

  | Fun (x, e1)     -> _____________________________________
```

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

```
# mem "x" (free e1) ;;
- : bool = false

# mem "x" (free e1') ;;
- : bool = true
```

(b) [3 points] Next, use `free` to complete the implementation of

```
let isWellFormed e = _____________________________________
```

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

```
# List.map isWellFormed [e1; e2; e3];;
- : bool list = [true; true; true]

# List.map isWellFormed [e1'; e2'; e3'];;
- : bool list = [false; false; false]
```