CSE 130, Spring 2014
Instructor: Ranjit Jhala

Name/ID ANSWER KEY

Final 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.

Question	Points	Score
1	25	
2	25	
3	35	
4	45	
5	20	
Total:	150	

1. [25 points] For each of the following OCaml or Scala programs, write down the **value** of `ans`.

(a) [5 points]

```
let rec range i j =
  if i > j
  then []
  else i :: (range (i+1) j)

let ans = range 1 5
```

ans = [1;2;3;4;5]

(b) [5 points]

```
let gelato g =
   let x = 10 in
   g x

let x   = 100

let f y = x + y

let ans = gelato f
```

ans = 110

(c) [5 points]

```
val ans = for ( i <- 1 to 5
              ; j <- i to 5
              ; k <- j to 5
              ; if (i*i + j*j == k*k))
          yield (i, j, k)
```

ans = Vector((3,4,5))

(d) [6 points]

```
class A () {
  def foo(x:Int) = 1 + this.bar(x)
  def bar(x:Int) = 1 + x
}

class B () extends A {
  override def bar(x:Int) = 100 + x
}

val x   = (new A) foo 10
val y   = (new B) foo 10

val ans = (x, y)
```

ans = (12, 111)

(e) [4 points]

```
var m1  = Map("tako" -> "nom nom", "uni" -> "blergh")

var m2  = m1

m1     += ("uni" -> "delicioso")

val ans = m2("uni")
```

ans = ”blergh”

2. [25 points] Consider the following Scala class and type definitions.

```
class A
class B extends A
type Point2A = {val x:A; val y:A}
type Point2B = {val x:B; val y:B}
type Point3A = {val x:A; val y:A; val z:A}
type Point3B = {val x:B; val y:B; val z:B}
```

Which of the below snippets of code **typechecks**? Circle the case that you believe holds.

(a) [5 points]

```
def ans = { def foo(p:Point2A) = error("ignore me")
            val p3 : Point3A    = error("ignore me")
            foo(p3) }
```

Does Not Typecheck **Typechecks** [YES: width-subtyping]

(b) [5 points]

```
def ans = { def foo(p:Point3A) = error("ignore me")
            val p2 : Point2A    = error("ignore me")
            foo(p2) }
```

Does Not Typecheck **Typechecks** [NO]

(c) [5 points]

```
def ans = { def foo(p: Point2A)  = error("ignore me")
            val p2 : Point2B     = error("ignore me")
            foo(p2) }
```

Does Not Typecheck **Typechecks** [YES: depth-subtyping]

(d) [5 points]

```
def ans = { def foo(f:(Point2A) => Int)  = error("ignore me")
            def f2(p:Point2B): Int       = error("ignore me")
            foo(f2) }
```

Does Not Typecheck **Typechecks** [NO: CO-VARIANT inputs]

(e) [5 points]

```
def ans = { def foo(f:(Point3B) => Int)  = error("ignore me")
            def f2(p:Point2A): Int       = error("ignore me")
            foo(f2) }
```

Does Not Typecheck **Typechecks** [YES: CONTRA-VARIANT inputs]

3. [35 points] A **binary-search-ordered dictionary** is a data structure that maps **keys** to **values**. We will represent dictionaries using a polymorphic Ocaml datatype:

```
type ('k, 'v) dict
  = Empty
  | Node of 'k * 'v * ('k, 'v) dict * ('k, 'v) dict
```

That is, a dictionary is represented as a tree, which is either empty, or a node with:

1. a binding from a `'k` key to an `'v` value,
2. a left sub-dictionary, and
3. a right sub-dictionary.

For example, consider the dictionary

fruit	price
apple	2.25
banana	1.50
cherry	2.75
grape	2.65
kiwi	3.99
orange	0.75
peach	2.25

that represents the prices (per pound) of various fruits. This dictionary is represented by the tree (on the left) which in turn is represented by the Ocaml value (of type `(string, float) dict`) bound to `fruitd` on the right.

```
                grape:
           _____2.65_____
          |              |
       banana:         orange:
   ___1.50___        ___0.75___
  |          |      |          |
apple:    cherry: kiwi:      peach:
2.25      2.75    3.99       2.25
```

```
let fruitd =
  Node ("grape", 2.65,
       Node ("banana", 1.50,
            Node ("apple", 2.25, Empty, Empty),
            Node ("cherry", 2.75, Empty, Empty)),
       Node ("orange", 0.75,
            Node ("kiwi", 3.99, Empty, Empty),
            Node ("peach", 2.25, Empty, Empty)))
```

Notice the tree is **Binary-Search-Ordered**, meaning for each node with a key `k`,

- keys in the **left** subtree are **less than** `k`, and
- keys in the **right** subtree are **greater than** `k`.

(a) [5 points] Recall the `type 'a option = None | Some of 'a`. Write a function

```
val find: 'k -> ('k, 'v) dict -> 'v option
```

such that `find k d` evaluates to `Some v` if `v` is the value associated with the key `k` in the dictionary `d`, and `None` otherwise. When you are done, you should get the following behavior:

```
# find "cherry" fruitd
- : float option = Some 2.75

# find "pomegranate" fruitd
- : float option = None
```

Fill in the blanks below to implement `find` as described.

```
let rec find k d =
  match d with
    | Empty ->

      ______________________None______________________
    | Node (k', v', l, r) when k = k' ->

      ______________________Some v'___________________

    | Node (k', v', l, r) when k < k' ->

      ______________________find k l__________________

    | Node (k', v', l, r) (* k' < k *) ->

      ______________________find k r__________________
```

(b) [8 points] Next, write a function

```
val deleteMax : ('k, 'v) dict -> ('k * 'v * ('k, 'v) dict)
```

such that `deleteMax d` returns a tuple of the **largest** key in `d`, the value corresponding to the key, and the dictionary **without** the corresponding key-value pair. When you are done you should get the following behavior:

```
# let d0 = Node ("banana", 1.50,
            Node ("apple", 2.25, Empty, Empty),
            Node ("cherry", 2.75, Empty, Empty)) ;;
...

# deleteMax d0 ;;

- : (string, float, (string, float) dict) =
    =  ("cherry", 2.75, Node ("banana", 1.50,
                              Node ("apple", 2.25, Empty, Empty),
                              Empty))
```

Fill in the blanks below to implement `deleteMax` as described. (It will only be called with non-`Empty` trees.)

```
let rec deleteMax d =
  match d with
  | Node (k', v', l, Empty) ->
        (k', v', l)
  | Node (k', v', l, r)  ->
        let (k", v", r') = deleteMax r in
        (k", v", Node (k', v', l, r'))
```

(c) [8 points] Using `deleteMax`, write a function

```
val delete : 'k -> ('k, 'v) dict -> ('k, 'v) dict
```

such that `delete k d` returns the dictionary with all the key-value pairs of `d` **except** `k`. If `k` was not present in `d` then the output should be the same as `d`. When you are done, you should get the following behavior:

```
#  delete "grape" fruitd ;;

- : (string, float) dict
  = Node ("cherry", 2.75,
       Node ("banana", 1.50,
            Node ("apple", 2.25, Empty, Empty),
            Empty),
       Node ("orange", 0.75,
            Node ("kiwi", 3.99, Empty, Empty),
            Node ("peach", 2.25, Empty, Empty)))
```

Fill in the blanks below, **using** `deleteMax`, to implement `delete`:

```
let rec delete k d =
  match d with
  | Empty ->
     Empty
  | Node (k', v', Empty, r) when k = k' ->
        r
  | Node (k', v', l, r) when k = k' ->
        let (k", v", l') = deleteMax l in
        Node (k", v", l', r)
  | Node (k', v', l, r) when k < k' ->
        Node (k', v', delete k l, r)
  | Node (k', v', l, r) (* when k' < k *) ->
        Node (k', v', l, delete k r)
```

(d) [7 points] The following function implements a `fold` over the dictionaries.

```
let rec fold f b t = match t with
  | Empty              -> b
  | Node (k, v, l, r) -> let b0 = fold f b r  in
                         let b1 = f k v b0    in
                         let b2 = fold f b1 l in
                         b2
```

What is the type of `fold`?

`val fold :` ('k -¿ 'v -¿ 'a -¿ 'a) -¿ 'a -¿ ('k, 'v) tree -¿ 'a

(e) [7 points] Fill in the blanks below to obtain a function

```
val keysWithValue : 'v -> ('k, 'v) dict -> 'k list *)
```

such that `keysWithValue v d` that returns the list of keys in `d` with value `v`. When done, you should get:

```
# keysWithValue 2.25 fruitd;;
- : string list = ["apple"; "peach"]
```

```
let keysWithValue v d =
  let f k' v' acc = ______ if v = v' then k' :: acc else acc ______ in
  let b           = ______ [] ______ in
  fold f b d
```

4. [45 points] Lets implement Scala-style `for`-loops in Ocaml, using the following functions:

```
let skip     = []

let yield x = [x]

let rec foreach xs f = match xs with
  | []     -> []
  | x::xs -> f x @ foreach xs f
```

(a) [3 points] What is the type of `skip`?

`val skip :` 'a list

(b) [4 points] What is the type of `yield`?

`val yield:` 'a -¿ 'a list

(c) [8 points] What is the type of `foreach`?

`val foreach:` 'a list -¿ ('a -¿ 'b list) -¿ 'b list

(d) [4 points] What is the value of `ans`?

```
let ans = foreach [1;2;3] (fun x ->
            yield (x * x)
          )
```

`ans =` [1; 4; 9]

(e) [6 points] What is the value of `ans`?

```
let ans = foreach [1; 2] (fun i ->
            foreach ["a"; "b"] (fun c ->
              yield (i, c)
            )
          )
```

`ans =` [(1, "a"); (1, "b"); (2, "a"); (2, "b")]

(f) [5 points] Recall the Scala code from the first question:

```
val ans = for ( i <- 1 to 5
              ; j <- i to 5
              ; k <- j to 5
              ; if (i*i + j*j == k*k))
          yield (i, j, k)
```

Translate it to the *equivalent* Ocaml, by filling the blanks below **using only** the functions `yield`, `skip`. (The function `range` is from Question 1):

```
let ans = foreach (range 1 5) (fun i ->
            foreach (range i 5) (fun j ->
              foreach (range j 5) (fun k ->
                if   (i*i + j*j = k*k)
                then yield (i,j,k)
                else skip
              )
            )
          )
```

(g) [5 points] Rewrite the usual `map` function for lists using only `foreach`, `skip` and `yield`:

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

let map f xs =
  foreach xs (fun x -¿
  yield (f x)
  )
```

(h) [5 points] Rewrite the usual `filter` function for lists using only `foreach`, `skip` and `yield`:

```
(* val filter : ('a -> bool) -> 'a list -> 'a list *)

let filter f xs =
  foreach xs (fun x -¿
  if (f x)
  then yield x
  else skip
  )
```

(i) [5 points] The function `flatten` of type:

```
val flatten : 'a list list -> 'a list
```

has the following behaviour:

```
# flatten [[1;2;3]; [4;5]; [6]] ;;
- : int list = [1;2;3;4;5;6]
```

Write `flatten` using only `foreach`, `skip` and `yield`:

```
let flatten xss =
foreach xss (fun xs -¿
foreach xs (fun x -¿
yield x
)
)
```

5. [20 points] Lets write a function to generate all **permutations** of a list.

   (a) [5 points] Write a function `insertAt` with the following behavior:

```
scala> insertAt(0, "cat", List("mouse", "giraffe", "hippo"))
res0: List[String] = List(cat, mouse, giraffe, hippo)

scala> insertAt(1, "cat", List("mouse", "giraffe", "hippo"))
res1: List[String] = List(mouse, cat, giraffe, hippo)

scala> insertAt(2, "cat", List("mouse", "giraffe", "hippo"))
res2: List[String] = List(mouse, giraffe, cat, hippo)

scala> insertAt(3, "cat", List("mouse", "giraffe", "hippo"))
res3: List[String] = List(mouse, giraffe, hippo, cat)
```

   Fill in the blanks to get a definition of `insertAt`

```
def insertAt[A](pos:Int, x:A, ys:List[A]): List[A] =
  (pos, ys) match {
    case (0, _)       => x :: ys
    case (n, y::ys_)  => y :: insertAt(n-1, x, ys_)
    case (_, Nil)     => x :: Nil
  }
```

   (b) [5 points] Next, write a function `spliceInto` with the following behavior:

```
scala> spliceInto("cat", List("mouse", "giraffe", "hippo"))
res4: List[List[String]] = List(List(cat, mouse, giraffe, hippo),
                                List(mouse, cat, giraffe, hippo),
                                List(mouse, giraffe, cat, hippo),
                                List(mouse, giraffe, hippo, cat))
```

   Fill in the blanks to get a definition of `spliceInto`

```
def spliceInto[A](x:A, ys:List[A]) : List[List[A]] =
  for (i <- ( 0 to ys.length ).toList)
    yield insertAt(i, x, ys)
```

(c) [10 points] Finally, use `spliceInto` to write a function `permutations` with the following behavior:

```
scala> permutations(List(0,1,2))
res5: List[List[Int]] = List(List(0, 1, 2),
                              List(1, 0, 2),
                              List(1, 2, 0),
                              List(0, 2, 1),
                              List(2, 0, 1),
                              List(2, 1, 0))
```

Fill in the blanks below to obtain an implementation of `permutations`

```
def permutations[A](xs:List[A]): List[List[A]] =
  xs match {
    case Nil     =>  List(Nil)
    case x::xs_  =>  for ( ys ¡- permutations(xs_)
                     ; zs ¡- spliceInto(x,ys) )
                     yield zs
  }
```