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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
1. [25 points] For each of the following OCaml or Scala programs, write down the value of \texttt{ans}.

(a) [5 points]
\begin{verbatim}
  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]
\end{verbatim}

(b) [5 points]
\begin{verbatim}
  let gelato g = 
    let x = 10 in 
    g x

  let x = 100

  let f y = x + y

  let ans = gelato f

  ans = 110
\end{verbatim}

(c) [5 points]
\begin{verbatim}
  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))
\end{verbatim}
(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.

```scala
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]
```scala
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]
```scala
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]
```scala
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]
```scala
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]
```scala
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:

```ocaml
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

<table>
<thead>
<tr>
<th>fruit</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>2.25</td>
</tr>
<tr>
<td>banana</td>
<td>1.50</td>
</tr>
<tr>
<td>cherry</td>
<td>2.75</td>
</tr>
<tr>
<td>grape</td>
<td>2.65</td>
</tr>
<tr>
<td>kiwi</td>
<td>3.99</td>
</tr>
<tr>
<td>orange</td>
<td>0.75</td>
</tr>
<tr>
<td>peach</td>
<td>2.25</td>
</tr>
</tbody>
</table>

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.

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:

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

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

Fill in the blanks below to implement find as described.

```ocaml
let rec find k d =  
match d with  
| Empty ->  
| Node (k', v', l, r) when k = k' ->  
| Node (k', v', l, r) when k < k' ->  
| Node (k', v', l, r) (* k' < k *) ->
```

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

```ocaml
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:

```ocaml
# 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.

```ocaml
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` 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] Let's implement Scala-style for-loops in Ocaml, using the following functions:

```ocaml
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`?

```ocaml
val skip : 'a list
```

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

```ocaml
val yield: 'a -> 'a list
```

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

```ocaml
val foreach: 'a list -> ('a -> 'b list) -> 'b list
```

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

```ocaml
let ans = foreach [1;2;3] (fun x ->
  yield (x * x)
)
ans = [1; 4; 9]
```

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

```ocaml
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:

```scala
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):

```ocaml
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`:

```ocaml
(* 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`:

```ocaml
(* 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:

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

has the following behaviour:

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

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

```plaintext
let flatten xss =
    foreach xss (fun xs -¿
        foreach xs (fun x -¿
            yield x
            )
        )
    )
```
5. [20 points] Let's write a function to generate all **permutations** of a list.

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

```scala
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`

```scala
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
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`

```scala
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
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`:

```scala
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
  }
```