Do not start the exam until you are told to.

This is a closed-book exam closed-notes, no-calculator exam. You many only refer to two pages of your own notes.

Do not look at anyone else’s exam. Do not talk to anyone but an exam proctor during the exam.

Write your answers in the space provided.

Wherever it gives a line limit for your answer, write no more than the specified number of lines explanation / code. The rest will be ignored.

Work out your solution in blank space / scratch paper, and only put your answer in the answer blank given.

The points for each problem are a rough indicator of the difficulty of the problem.

Good luck!
1. [24 points] For each of the following Ocaml programs, write down the value of `ans`.

a. [4 points]
   ```ocaml
   let rec foo n f x =
     if n <= 0 then x else foo (n-1) f (f x) in
   foo 100 (fun y -> y + 1) 0
   ```

   `ans =` ____________

b. [4 points]
   ```ocaml
   let foo =
     let x = 1 in
     (fun y -> let x = x + y in x) in
   (foo 100, foo 1000)
   ```

   `ans =` ____________

c. [4 points]
   ```ocaml
   let rec foo xs ys =
     match xs, ys with
     | x::xs', y::ys' -> (x,y)::(foo xs' ys') in
     | _,_ -> [] in
   foo ([1;2;3],["a";"b"])
   ```

   `ans =` ____________

d. [4 points]
   ```ocaml
   type mix = Int of int | Bool of bool
   let foo x =
     match x with
     | 0 -> Bool true
     | -1 -> Bool false
     | _ -> Int x in
   foo 12
   ```

   `ans =` ____________

e. [8 points]
   ```ocaml
   let f g = fun x -> g (g x) in
   let h = f f (fun x -> x*10) in
   h 1
   ```

   `ans =` ____________
2. [ 25 points ] For each of the following Ocaml programs, write down the type of \texttt{ans}.

a. [ 5 points ]
\begin{verbatim}
let ans x =
  match x with
  | -2 -> Bool false
  | -1 -> Bool True
  | _  -> Int x
\end{verbatim}
\texttt{ans}:

b. [ 5 points ]
\begin{verbatim}
let ans f g x =
  if f x then x else g x
\end{verbatim}
\texttt{ans}:

c. [ 5 points ]
\begin{verbatim}
let rec ans n f x =
  if n <= 0 then x else ans (n-1) f (f x)
\end{verbatim}
\texttt{ans}:

d. [ 5 points ]
\begin{verbatim}
let ans b f g =
  (fun x -> (if b then f else g) x)
\end{verbatim}
\texttt{ans}:

e. [ 5 points ]
\begin{verbatim}
let rec ans x ys =
  match ys with
  | [] -> x
  | y::ys' -> ans (y x) ys'
\end{verbatim}
\texttt{ans}:
3. [24 points] For each Ocaml function below, write down a tail-recursive function that will produce the same output for each input. You can create any local helper functions, as long as they are all tail-recursive.

a. [8 points]
   let rec fac x =
   if x <= 1 then 1 else x * fac (x-1)

b. [8 points]
   let rec map f xs =
   match xs with
   | [] -> []
   | x::xs' -> (f x)::(map f xs')

   Hint: First, try to figure out what foldr does.

c. [8 points]
   let rec foldr f xs b =
   match xs with
   | [] -> b
   | x::xs' -> f x (foldr f xs' b)
4. [ 55 points ]

a. [ 3 points ] Consider the following Ocaml datatype representing Nano-ML types.

\[
\text{type ty = Tyint | Tybool | Tyfun of ty * ty}
\]

Thus, \text{Tyint} represents the Nano-ML type \text{int} and \text{Tyfun(Tyint,Tybool)} represents the Nano-ML type \text{int->bool}. Write down the Ocaml value of type \text{ty} corresponding to the ML type: \text{int -> int -> int}

b. [ 7 points ] A \textit{type environment} is like an environment, i.e. the “phone book” mapping names to values, but only maps variables to their \textit{types} (not values, as in an environment). Consider the following Ocaml datatype representing Nano-ML type environments (similar to the type \texttt{env} in PA4).

\[
\text{type tyenv = (string * ty) list}
\]

Write a function: \texttt{lookup : tyenv -> string -> ty option} such that:

- \texttt{lookup [(x1,t1);...;(xn,tn)] x} returns \texttt{Some ti} if \texttt{x} is equal to \texttt{x} and for all \texttt{j} less than \texttt{i}, \texttt{xj} is not equal to \texttt{x}, and returns \texttt{None} if none of the \texttt{xi} are equal to \texttt{x}. This is like looking up the value of \texttt{x} (as in PA4) but here we only care about the type. Thus,

- \texttt{lookup [("x",Tyint);("y",Tyint);("x",Tybool)] "x"} should return \texttt{Some Tyint} meaning the variable \texttt{x} has the type \texttt{Tyint} in the given type environment,

- \texttt{lookup [("x",Tyint);("y",Tyfun(Tyint,Tyint));("x",Tybool)] "y"} should return \texttt{Some (Tyfun (Tyint,Tyint))},

- \texttt{lookup [("x",Tyint);("y",Tyint);("x",Tybool)] "z"} should return \texttt{None} as the variable \texttt{z} is not bound in the type environment.

Write the function \texttt{lookup} by filling in the blanks below.

\[
\text{let rec lookup tenv x =}
\]

\[
\begin{align*}
\text{Some Tyint} & \quad \text{(for checking)} \\
\text{Tyfun (Tyint, Tyint)} & \quad \text{(for checking)} \\
\text{None} & \quad \text{(for checking)}
\end{align*}
\]
Next, consider the Ocaml datatypes representing typed Nano-ML expressions. These are just Nano-ML expressions, where additionally, each function’s argument is given a type.

```
type binop = Plus | Minus | Eq | Lt | And | Or

type expr =
  Const of int
| Var of string
| Bin of expr * binop * expr
| If of expr * expr * expr
| Let of string * expr * expr (* let X = E1 in E2 ---> Let (X,E1,E2) *)
| App of expr * expr (* E1 E2 ---> App(E1,E2) *)
| Fun of string * ty * expr (* fun X:T -> E ---> Fun(X,T,E) *)
```

Notice that the case for `Fun` in the definition of `expr` takes an argument which is the `type` of the formal parameter. Thus,

- `Fun("x",Tyint,Bin(Var "x",Plus,Const 10))` represents the function that takes an integer argument `x` and returns the argument plus 10,
- `Fun("x",Tyint, Fun("y",Tyint, If (Binop(Var "x", Lt, Var "y"), Var "y", Var "y"), Var "y", Var "x")` represents a curried function of type `int -> int -> int` which takes two arguments and returns the larger argument.

c. [5 points] Write down the Ocaml value of type `expr` corresponding to Nano-ML expression.

```
let x = 10 in
let y = x + 12 in
x + y
```

Finally, fill in the blanks below to obtain a function `check: typenv -> expr -> typ` such that `check env e` returns `Some t` if the type of `e` in the type environment `env` is `t`, and returns `None` if `e` is not well typed in the environment. For example:

- `check [("x",Tyint);("y",Tyfun(Tyint,Tyint));("z",Tybool)] (Var "y")` should return `Some (Tyfun(Tyint,Tyint))`,
- `check [("x",Tyint);("y",Tyfun(Tyint,Tyint));("x",Tybool)] (Binop (Var "x",Plus,Const 2))` should return `Some Tyint`.
- `check [("x",Tyint);("y",Tyfun(Tyint,Tyint));("x",Tybool)] (Binop (Var "x",Plus,Var "y")`) should return `None`, and,
- `check ["x",Tyint]) (App (Fun("x",Tyint,Bin(Var "x",Plus,Const 10)), Var "z"))` should return `Some Tyint`. 

6
d. [40 points] let rec check env e =
  match e with
  | Const i ->

  | Var x ->

  | Plus (e1,e2) | Minus (e1,e2) ->
    let t1 = check env e1 in
    let t2 = check env e2 in

  | Leq (e1,e2) | Eq (e1,e2) ->
    let t1 = check env e1 in
    let t2 = check env e2 in

  | And (e1,e2) | Or (e1,e2) ->
    let t1 = check env e1 in
    let t2 = check env e2 in

  | App (e1,e2) ->
    let t1 = check env e1 in
    let t2 = check env e2 in
    (match (t1,t2) with None, _ | _, None -> None
                         | _ , _ -> None ->)

  | Fun (x,t,e) ->
    (match (check ((x,t)::env) e) with None -> None
                         | _ -> None ->)

  | Let (x,e1,e2) ->
    (match check env e1 with None -> None
                         | _ -> None ->)

  | If (p,t,f) ->
    let tp = check env p in
    let tt = check env t in
    let tf = check env f in

  | _ -> None

5. [18 points] For each of the following Scala programs, write down the value of ans.

a. [6 points]
```scala
val x = Array("a", "b", "c")
val y = Array("1", "2", "3")

def f(a: Array[String], b: Array[String]) {
    val a = Array("100", "200")
    b(0) = "45"
}

val _ = f(x, y)
val ans = (x, y)
```

ans = ________________

b. [6 points]
```scala
val a = 10
val b = Array(100)

def f(x : Int) = {
    val b = Array(x)
    (y: Int) => {
        val rv = y - a - b(0)
        b(0) = y
        rv
    }
}

val f1 = f(1000)
val ans = (f1(10000), f1(10000))
```

ans = ________________

c. [6 points]
```scala
def q(n:Int)(g: Int=>Int) = {
    val count = Array(n)
    (x: Int) => {
        if (count(0) <= 0) 0 else {
            count(0) -= 1
            g(x)
        }
    }
}

val fac: Int => Int = q(7) { (k: Int) =>
    if (k <= 1) 1 else { k * (fac(k-1)) }
}

val ans = (fac(5), fac(5))
```

ans = ________________
6. [ 35 points ]
   a. [ 10 points ] Explain in at most two lines, one reason why Java disallows multiple inheritance.

   b. [ 10 points ] Explain in at most two lines, why the above problem does not arise with multiple interfaces.

We would like to write a Scala function `tick` that takes no arguments, such that:
1. the i-th call `tick()` returns i, and,
2. the behavior of `tick` is not changed by any other code in the program (except re-assigning the name `tick` to something else).

Consider the following implementation.

```
var ctr = 0
def tick() = {
  ctr += 1
  ctr
}
```

c. [ 5 points ] Explain in at most two lines, why the above does not meet the requirements for `tick`.

d. [ 10 points ] Write down a correct implementation of `tick` that meets the specification given in the previous question. **Hint:** You just have to bind the name `tick` to an appropriate function object.
7. [53 points] For this question, you will write Scala code that determines whether a given graph \((V, E)\) can be colored with \(k\) colors. A graph \((V, E)\) is a set of vertices \(V\) and a set of edges \(E\) that are pairs of vertices. Two vertices \(u, v\) are adjacent if there is an edge \((u, v)\) in \(E\). A \(k\)-coloring of a graph is an assignment of colors from \(1, \ldots, k\) to the vertices \(V\), such that every two adjacent vertices get different colors.

Assume that the \(n\) vertices are represented by the numbers \(0, \ldots, n - 1\), and the edges as a list of pairs of integers corresponding to the vertices. Thus, the following graph:

\[
\begin{array}{cccc}
0 & \quad & 1 \\
| & \quad & | \\
| & \quad & | \\
| & \quad & | \\
3 & \quad & 2 \\
\end{array}
\]

is represented by the list of edges: \(\text{List}((0,1),(1,2),(2,3),(3,0))\).

We will represent an assignment of \(k\) colors to the \(n\) vertices as a list: \([c_0, \ldots, c_{n-1}]\) where each \(0 \leq c_i \leq k - 1\). Note that if \(c\) is the list corresponding to the coloring, then \(c(i)\) is the color assigned to the vertex \(i\).

a. [8 points] First, write a function \(\text{valid}\) which takes as input a list of edges \(\text{es}\) and a coloring \(c\), and returns \(\text{True}\) if the coloring is valid and \(\text{False}\) otherwise. When you are done, you should get:

```scala
scala> val es = [(0,1),(1,2),(2,3),(3,0)]
scala> valid(es,List(0,1,0,1))
res: Boolean = true
scala> valid(es,List(0,0,1,1))
res: Boolean = false
```

The body of the function should be at most 4 lines long. Write it by filling in the blanks below:

```scala
def valid(es:List[(Int, Int)], c: List[Int]): Boolean = {
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
}
```
b. [ 20 points ] Next, you will write a function `colorings` that takes as input a number of vertices `n` and a number of colors `k` and either returns an iterator over all possible colorings of `n` vertices with `k` colors.

When you are done, you should get:

```scala
github> for (c <- colorings(3, 2)) println(c)
```

```
List(0, 0, 0)
List(0, 0, 1)
List(0, 1, 0)
List(0, 1, 1)
List(1, 0, 0)
List(1, 0, 1)
List(1, 1, 0)
List(1, 1, 1)
```

Write it by filling in the blanks below:

```scala
def colorings(n: Int, k: Int): List[List[Int]] = {
    if (n <= 0) _______________________________________________
    else { _______________________________________________
        _______________________________________________
        _______________________________________________
    }
}
```
Now, we have a procedure for determining if a given graph with \( n \) vertices, represented by the edges \( es \), can be colored. First, we find the number of vertices:

```scala
def vertices(es: List[(Int, Int)]) = {
  var n = 0
  for ((i,j) <- es) { n = n max i max j }
  n
}
```

and then we can iterate over all the colorings to see if a valid coloring \( \text{exists} \):

```scala
def colorable(es: List[(Int, Int)], k: Int) : Boolean = {
  val n = vertices(es)
  val cs = colorings(n, k)
  cs.exists(valid(es, _))
}
```

The problem with this approach is that we have to generate and store all the possible colorings in advance in the list output by \( \text{colorings} \).

Instead, we will write a class called \( \text{colorings} \) whose instances have a \( \text{next} \) method that allow us to \( \text{iterate} \) over the colorings without generating all of them.

```scala
case class colorings(n: Int, k: Int) {
  var curr: List[Int] = initColoring(n)

  def hasNext() : Boolean = ! (lastColoring(curr, k))

  def next(): List[Int] = { curr = nextColoring(curr, k); curr }

  def exists(f: List[Int] => Boolean): Boolean = {
    if (f(curr)) true
    else if (hasNext()) {next(); exists(f)}
    else false
  }
}
```

Write the appropriate implementations of functions \( \text{initColoring}, \text{lastColoring} \) and \( \text{nextColoring} \). When you are done, you should get the following behavior using the new class \( \text{colorings} \).

```scala
scala> val c = colorings(3, 2)
scala> c.curr
res: List[Int] = List(0, 0, 0)
scala> c.next()
res: List[Int] = List(0, 1, 0)
```

. . .

```scala
scala> c.next()
res: List[Int] = List(0, 1, 0)
```
res: List[Int] = List(1, 1, 1)

scala> c.hasNext()
res: Boolean = false

Moreover, the function colorable defined above will work correctly.

c. [25 points]

def initColoring(n: Int) =

  return ___________________________________________________________________


def lastColoring(c, k):

  return ___________________________________________________________________


def nextColoring(xs: List[Int], k: Int): List[Int] =

  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________