• Do not start the exam until you are told to.

• This is a closed-book exam closed-notes, no-calculator exam. You may 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

2. 25 Points

3. 24 Points

4. 55 Points

5. 18 Points

6. 35 Points

7. 53 Points

TOTAL 234 Points
1. [24 points] For each of the following Ocaml programs, write down the value of `ans`.

a. [4 points]
```ocaml
let ans =
  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 ans =
  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 ans =
  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 ans =
  let foo x =
    match x with
    0 -> Bool true
    | -1 -> Bool false
    | _ -> Int x in
  foo 12

ans = ____________________
```

e. [8 points]
```ocaml
let ans =
  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 `ans`.

a. [5 points]
   ```ocaml
type mix = Int of int | Bool of bool
let ans x =
  match x with
  | -2 -> Bool false
  | -1 -> Bool True
  | _ -> Int x
```

ans:____________________

b. [5 points]
   ```ocaml
let ans f g x =
  if f x then x else g x
```

ans:____________________

c. [5 points]
   ```ocaml
let rec ans n f x =
  if n <= 0 then x else ans (n-1) f (f x)
```

ans:____________________

d. [5 points]
   ```ocaml
let ans b f g =
  (fun x -> (if b then f else g) x)
```

ans:____________________

e. [5 points]
   ```ocaml
let rec ans x ys =
  match ys with
  | [] -> x
  | y::ys' -> ans (y x) ys'
```

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')

c. [8 points]

   let rec foldr f xs b =
      match xs with
         [] -> b
         | x::xs' -> f x (foldr f xs' b)

   Hint: First, try to figure out what foldr does.
4. [55 points]

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

```ocaml
type ty = Tyint | Tybool | Tyfun of ty * ty
```

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

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

```ocaml
type tyenv = (string * ty) list
```

Write a function:

```ocaml
let lookup : tyenv -> string -> ty option
```

such that:

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

- `lookup [("x",Tyint);("y",Tyint);("x",Tybool)] "x"` should return `Some Tyint` meaning the variable `x` has the type `Tyint` in the given type environment,
- `lookup [("x",Tyint);("y",Tyfun(Tyint,Tyint));("x",Tybool)] "y"` should return `Some (Tyfun (Tyint,Tyint))`,
- `lookup [("x",Tyint);("y",Tyint);("x",Tybool)] "z"` should return `None` as the variable `z` is not bound in the type environment.

Write the function `lookup` by filling in the blanks below.

```ocaml
let rec lookup tenv x =
```

```ocaml

```
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.

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

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

a. [6 points]
   ```python
   x = ["a", "b", "c"]
y = [1, 2, 3]

def f(a, b):
a = ["p", "q"]
b[0] = "r"

f(x, y)
ans = (x, y)

ans = ________________
```

b. [6 points]
   ```python
   def f(x):
b = [x]
def g(y):
    rv = y - a - b[0]
b[0] = y
    return rv
return g

a = 10
b = [100]

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

ans = ________________
```

c. [6 points]
   ```python
   def q(n, g):
count = [n]
def g1(x):
    if count[0] > 0:
      count[0] -= 1
      return g(x)
    else:
      return 0
return g1

def fac(k):
  if k <= 1: return 1
  else: return k*(fac(k-1))

fac = q(7, fac)

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 Python 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.

```python
ctr = 0
def tick():
    global ctr
    ctr = ctr + 1
    return 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 Python 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:

![Graph Diagram](image)

is represented by the list of edges: \([(0,1),(1,2),(2,3),(3,0)]\). 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 `valid` which takes as input a list of edges `es` and a coloring `c`, and returns `True` if the coloring is valid and `False` otherwise. When you are done, you should get:

```python
>>> es = [(0,1),(1,2),(2,3),(3,0)]
>>> valid(es,[0,1,0,1])
True
>>> valid(es,[0,0,1,1])
False
```

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

```python
def valid(es,c):
    # your code here
```

```python
```
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 the list of all possible colorings of `n` vertices with `k` colors or yields all possible colorings. When you are done:

```python
>>> for c in colorings(3,2):
>>>     ... print c
should result in:
[0,0,0]
[0,0,1]
[0,1,0]
[0,1,1]
[1,0,0]
[1,0,1]
[1,1,0]
[1,1,1]
The body of the function should be at most 6 lines long (a real pythonist/a could do it in two lines). Write it by filling in the blanks below:

def colorings(n,k):

    """
    """
Now, we have a procedure for determining if a given graph with n vertices, represented by the edges \( \text{es} \), can be colored. First, we find the number of vertices.

```python
def vertices(es):
    vs = []
    for (i,j) in es: vs += [i, j]
    return max(vs)
```

and then we can iterate over all the colorings.

```python
def colorable(es,k):
    for c in colorings(vertices(es),k):
        if valid(es,c): return True
    return False
```

The problem with this approach is that we have to generate and store all the possible colorings in advance in the list output by `colorings`. Instead, we will write a class called `colorings` whose instances have a `next` method that allow us to iterate over the colorings without generating all of them.

```python
class colorings:
    def __init__(self,n,k):
        self.colors = k
        self.vertices = n
        self.current = 0
    def next(self):
        if self.current == 0:
            self.current = initColoring(self.vertices)
        elif lastColoring(self.current,self.colors):
            raise StopIteration
        else:
            self.current = nextColoring(self.current,self.colors)
        return self.current
    def __iter__(self):
        return self
```

Write the appropriate implementations of functions `initColoring`, `lastColoring` and `nextColoring`. When you are done, you should get the following behavior using the new class `colorings`. Do not use `yield` for these functions.

```bash
>>> c = colorings(3,2)
>>> c.next()
[0,0,0]
>>> c.next()
[0,0,1]
>>> c.next()
[0,1,0]
>>> c.next()
[1,0,0]
>>> c.next()
[1,0,1]
>>> c.next()
[1,1,0]
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
Moreover, the function `colorable` defined above will work correctly.

c. [25 points]

def initColoring(n):
    return ________________________________________________________________

def nextColoring(c,k):
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________
    ______________________________________________________________________

def lastColoring(c,k):
    return ________________________________________________________________
8. [35 points]
a. [15 points] Write a decorator `tracked` for classes (i.e., like the `counted` decorator from lecture), which
decorates a class by adding to it two new methods: `instId` and `getInst`, such that for each instance `x` of
the decorated class, `x.instId()` returns a unique integer identifier `i` where `x` is the `i`-th instance of the
decorated class, and `x.getInst(j)` returns the `j`-th instance of the decorated class.
Assume that the class being decorated does not have any methods `instId` and `getInst`, and don’t worry
about the original fields. When you are done, you should get the following behavior:

```python
>>> class C:
...     def __init__(self, v):
...         self.x = v

>>> C = tracked(C)
>>> c1 = C("Jack")
>>> c2 = C([0,1,2])
>>> c3 = C(2004)
>>> ids = [c1.instId(), c2.instId(), c3.instId()]
>>> [c1.getInst(id).x for id in ids]
["Jack", [0,1,2], 2004]
```
b. [10 points] Write a decorator `automap` that decorates functions of single arguments as follows.

```python
@automap
def square(x):
    return x*x
```

```python
>>> square(2)
4
>>> square([1,2,3,4,5])
[1, 4, 9, 16, 25]
```

If the decorated function is passed a single argument, it behaves like the original function. If passed a list of arguments, it returns the list of outputs of the application of the function to each input element.

**Hint:**

```python
>>> type([1,2,3]) == list
True
```

c. [10 points] Make the above decorator work for any sequence type, i.e. any object with a method `__iter__`.

**Hint:**

```python
>>> tuple([1,2,3])
(1, 2, 3)
```