1. **Continuation-passing style**

Convert each of the following OCaml functions into continuation-passing style, i.e. so that all calls are tail calls. Do not use any imperative features.

(a)

```ocaml
let rec factorial (n:int) : int =
  if n < 2 then 1 else n * factorial(n-1)
```

(b)

```ocaml
let rec fibonacci (n:int) : int =
  if n < 2 then 1 else fibonacci(n-1) + fibonacci(n-2)
```

Answer: (a)

```ocaml
let factorial (n:int) : int =
  let rec f (n:int) (k:int->'a) : 'a =
    if n < 2 then
      k 1
    else
      f (n-1) (fun a -> k (n * a)) in
  f n (fun a -> a)
```

(b)

```ocaml
let fibonacci (n:int) : int =
  let rec f (n:int) (k:int->'a) : 'a =
    if n < 2 then
      k 1
    else
      f (n-1) (fun a1 -> f (n-2) (fun a2 -> k (a1+a2))) in
  f n (fun a -> a)
```
2. **Algebraic Specifications**

A *bag* is a data type that is just like a set, except that it can contain multiple copies of a given element. Consider the following partial specification of an abstract data type of Bags:

ADT bag
Signatures:
  - empty : bag
  - singleton : elem -> bag
  - union : bag -> bag -> bag
  - delete : bag -> elem -> bag
  - count : bag -> elem -> int

Axioms:
  - count (delete b e') e =
    - if e = e' then max(0, (count b e) - 1) else (count b e)
  - ...

(a) Identify the constructors and the observers of this ADT, using the criteria given in lecture.
(b) Give enough additional axioms to define the ADT’s behavior fully.
(c) Give an OCaml implementation of the ADT, in which the `bag` type is represented as a history of the constructors used to build the bag.

**Answer:**
(a) **constructors:** empty, singleton, union, delete
(b) **observers:** count

(b)

\[
\begin{align*}
  \text{count empty } e &= 0 \\
  \text{count (singleton } e') e &= \text{if } e = e' \text{ then } 1 \text{ else } 0 \\
  \text{count (union } b1 \ b2) e &= \text{count } b1 e + \text{count } b2 e
\end{align*}
\]
module Bag :
sig
  type 'a bag
  val empty : 'a bag
  val singleton : 'a -> 'a bag
  val union : 'a bag -> 'a bag -> 'a bag
  val delete : 'a bag -> 'a -> 'a bag
  val count : 'a bag -> 'a -> int
end =
struct
  type 'a bag =
    Empty
  | Singleton of 'a
  | Union of 'a bag * 'a bag
  | Delete of 'a bag * 'a

  let empty = Empty
  let singleton e = Singleton e
  let union b1 b2 = Union (b1,b2)
  let delete b e = Delete (b,e)
  let rec count b e =
    match b with
    | Empty -> 0
    | Singleton e' -> if e = e' then 1 else 0
    | Union (b1,b2) -> count b1 e + count b2 e
    | Delete (b,e') -> if e = e' then max 0 (count b e) - 1 else count b e
end
3. **Dynamic Function Binding**

class Programmer():
    def who(self):
        return "Unknown programmer"
    def code(self):
        return "code"
    def hacks(self):
        return "hacks " + self.code()

Class PythonProgrammer(Programmer):
    def who(self):
        return "Python programmer"
    def code(self):
        return "classes"

class OCamlProgrammer(Programmer):
    def who(self):
        return "OCaml programmer"
    def hacks(self):
        return "fixes type errors"

team = (PythonProgrammer(), OCamlProgrammer())
for p in team:
    print (p.who(), p.hacks())

(a) Assuming normal Python semantics, where all member functions are treated as dynamically dispatched, what is the output of this program?

(b) Suppose we translated this example into C++, which uses type-based static binding by default, so that none of the member functions in this example were treated as dynamically dispatched. Assuming \texttt{team} were declared to be a suitable container object with elements of type \texttt{Programmer}, what would the output of this program be then?

**Answer: (a)**

Python programmer hacks classes
OCaml programmer fixes type errors

(b)

Unknown programmer hacks code
Unknown programmer hacks code
4. **Polymorphic Type Inference** Consider the following OCaml function definitions:

```ocaml
let f (x, y) = (y, x + 1)
let g z = [z; z]
let h p = g(f(f p))
```

What types will OCaml infer for `f`, `g`, and `h`? Hint: To infer the types you can extract and solve constraints, but this is not necessary; it is probably easier to figure them out informally.

**Answer:** OCaml actually infers a type for each function separately, and makes each as general as possible; hence the types for `f` and `g` don't depend on `h`, and we get:

- `f : int * 'a -> 'a * int`
- `g : 'a -> 'a list`
- `h : int * int -> (int * int) list`

If you follow the constraint extraction and solving approach presented in lecture for monomorphic types, and treat all three definitions as a single unit, then it is reasonable to assume that the types of `f` and `g` are affected by the way they are used in `h`, and we get:

- `f : int * int -> int * int`
- `g : (int * int) -> (int * int) list`
- `h : int * int -> (int * int) list`