Exo 1

1) f : int -> int -> int * int
   s : int

2) g : int -> float
   gl : liste de float

3) h : func -> liste de int * string -> liste de string

   r : liste de string

Exo 2

1)
```
let rec f_i_a func i j acc =
    if i > j then
        acc
    else if (func j) = true then
        f_i_a i (j-1) (j :: acc)
    else
        f_i_a i (j-1) acc
```

```
let filter_int func i j =
    f_i_a func i j []
```

2) type op = Del | Dup | Nop

```
let rec apply_op lv lo =
    match lv, lo with
    [], [] -> []
  | [], _ -> failwith "erreur"
  | _, [] -> failwith "erreur"
  | vi::llv, oi::llo ->
      let res apply_op llv llo in
      match oi with
        Dup -> vi :: vi :: res
      | Nop -> vi :: res
      | Del -> res
```

3) La fonction n'est pas récursive terminale car à la fin on ne propage pas le résultat, on appelle la fonction g avec le résultat de f.

4) f : (fun 'a -> 'a -> 'a) -> 'a list -> 'a

Exo 3

1)

```
let pr_coord c =
    match c with
    Tel (v) -> Printf.printf "tel : %s\n " v
  | Email (v) -> Printf.printf "email : %s\n" v
  | Social (v) -> Printf.printf "social : %s  %s\n"
      (fst v) (snd v)

let pr_contact c =
    Printf.printf "%s  %s\n" c.prenom c.nom
let () = if c.favorit = true then Printf.printf "Favorit\n" in
    List.iter (fun x -> (pr_coord x)) c.coords
```

3/
```
let pr_carnet car =
    List.iter (fun x -> (pr_contact x)) car
```

4)
```
let contact_masc_coords car =
    let f_f a x =
        if (List.len (a.coords)) < (List.len (x.coords)) then
            x
        else
            a
    in
    if List.len car = 0 then failwith "error"
    List.fold_left f_f (List.hd car) car
```

```
(4 |> sqrt / |> int_of_float
```