TP 1, corrigé

```
let rec longueur l = match l with
    | [] -> 0
    | t::q -> 1 + longueur q;;

let rec maximum l = match l with
    | [] -> failwith "liste vide"
    | [a] -> a
    | t::q -> max t (maximum q);;

let miroir l =
    let rec aux l acc = match l with
    | [] -> acc
    | t::q -> aux q (t::acc) in aux l [];;

let rec present l e = match l with
    | [] -> false
    | t::q -> t=e || present q e;;

let rec supprime l e = match l with
    | [] -> []
    | t::q when t=e -> supprime q e
    | t::q -> t::(supprime q e);;

let rec concat l1 l2 = match l1 with
    | [] -> l2
    | t::q -> t::(concat q l2);;

let rec applique f l = match l with
    | [] -> []
    | t :: q -> (f t)::(applique f q);;

let rec reuens s1 s2 = match s1 with
    | [] -> s2
    | t::q when List.mem t s2 -> reuens q s2
    | t::q -> t::(reuens q s2);;

let rec interens s1 s2 = match s1 with
    | [] -> []
    | t1::q1 when List.mem t1 s2 -> t1 :: (interens q1 s2)
    | t1::q1 -> interens q1 s2;;

let rec diffens s1 s2 = match s1 with
    | [] -> []
    | t1::q1 when List.mem t1 s2 -> diffens q1 s2
    | t1::q1 -> t1::(diffens q1 s2);;

let diffsym s1 s2 =
    let a = reuens s1 s2 and b = interens s1 s2 in diffens a b;;
```

```
let dernier prop l =
let rec aux l acc = match l with
|   [] -> if acc=[] then failwith "pas trouvé" else List.hd acc
| t::q when prop t -> aux q (t::acc)
| t::q -> aux q acc in aux l [];;

let aumoins2 l =
let rec aux l acc = match l with
    | [] -> acc
    | t::q when (List.mem t q) && not (List.mem t acc) -> aux ↩
       q (t::acc)
    | t::q -> aux q acc in aux l [];;

let rec prefixe l = match l with
    | [] -> [ ]
    | t::q -> [t]::(List.map (fun l -> t::l)  (prefixe q));;

type exp = Const of float | X | Somme of exp * exp | Produit ↩
    of exp * exp | Puiss of exp * int ;;
let test = Produit(Puiss (X, 2) , Somme( X, Const 1.5));;

let rec str_of_exp e = match e with
 | Const x -> string_of_float x
 | X -> "x"
 | Somme (e1,e2) -> "("^(str_of_exp e1)^"+"^(str_of_exp e2)^")"
 | Produit (e1,e2) -> "("^(str_of_exp e1)^"*"^(str_of_exp e2)^")"
 | Puiss (e,n) -> (str_of_exp e)^"^"^string_of_int n;;

let rec image e x = match e with
    | Const c -> c
    | X -> x
    | Somme (e1,e2) -> (image e1 x) +. (image e2 x)
    | Produit (e1,e2) -> (image e1 x) *. (image e2 x)
    | Puiss (e,n) -> (image e x)** (float_of_int n);;

let rec deriv e = match e with
    | Const _ -> Const 0.
    | X -> Const 1.
    | Somme (e1,e2) -> Somme (deriv e1, deriv e2)
    | Produit (e1,e2) -> Somme (Produit(deriv e1, e2), Produit ↩
       (e1, deriv e2))
    | Puiss (e,n) -> Produit (Const (float_of_int n), Produit ↩
       (deriv e, Puiss (e, n-1)));;

type zbarre = Moinsinfini | Plusinfini | Valeur of int ;;
let degre p = match p with
    | [] -> Moinsinfini
    | _ -> Valeur (List.length p -1);;
```

```
let rec exprap x n = match n with
    | 0 -> 1
    | n -> let rac = exprap x (n/2) in (if (n mod 2 =0) then 1 ↩
       else x) * rac * rac;;

let eval p x0 =
    let rec aux l puiss = match l with
        | [] -> 0
        | coeff :: reste -> coeff * (exprap x0 puiss) + aux ↩
           reste (puiss+1) in
    aux (List.rev p) 0;;

let horner p x0 =
    let rec aux p = match p with
        | [] -> 0
        | t :: q -> t + x0 *(aux q) in aux (List.rev p);;

let limite p x = match x,p with
    | _ , [] -> Valeur 0
    | _ , [a] -> Valeur a
    | Moinsinfini, t :: q when t < 0 -> if List.length p mod 2 ↩
       = 1 then Plusinfini else Moinsinfini
    | Moinsinfini, t::q -> if List.length p mod 2 = 1 then ↩
       Moinsinfini else Plusinfini
    | Plusinfini, t :: q -> if t<0 then Moinsinfini else ↩
       Plusinfini
    | Valeur a, _ -> Valeur (horner p a);;

let rec normalize l = match l with
    | [] -> []
    | 0 :: q -> normalize q
    | _ -> l;;

let addition p1 p2 =
    let rec aux p1 p2 = match p1, p2 with
        | _,[] -> p1
        | [], _ -> p2
        | x1 :: q1 , x2 :: q2 -> (x1 + x2) :: (aux q1 q2) in
    normalize (List.rev (aux (List.rev p1) (List.rev p2)));;

let scalaire p a = List.map (fun x -> x*a) p;;
let multX p = p@[0];;

let produit p1 p2 =
    let rec aux p2 = match p2 with
        | [] -> []
        | t2 :: q2 -> addition (scalaire p1 t2) (multX (aux ↩
           q2)) in
    aux (List.rev p2);;
```