```
(** EX 1 *)
type interval = { inf : int; sup : int}

let make_interval a b =
    if a < b
    then { inf = a; sup = b}
    else { inf = b; sup = a}

let min4 a b c d = min a(min b(min c d))

let max4 a b c d = max a(max b(max c d))

let add_i {inf = i1; sup = s1} {inf = i2; sup = s2} =
    {inf = min4 (i1+i2) (i1+s1) (i2+s1) (i2+s2);
     sup = max4 (i1+i2) (i1+s1) (i2+s1) (i2+s2)}

let apply_op op {inf = i1; sup = s1}{inf = i2; sup = s2} =
    {inf = min4 (op i1 i2) (op i1 s1) (op i2 s1) (op i2 s2);
     sup = max4 (op i1 i2) (op i1 s1) (op i2 s1) (op i2 s2)}

let add = apply_op (+)
let sub = apply_op (-)
let mul = apply_op ( * )

let apply op i1 i2 =
  let a = op i1.inf i2.inf in
  let b = op i1.inf i2.sup in
  let c = op i1.sup i2.inf in
  let d = op i1.sup i2.sup in
  (* on utilise min4 et max4 pour calculer les bornes inf et sup *)
  make_interval (min4 a b c d) (max4 a b c d)
(* On peut maintenant creer add sub et mul en utilisant op et en lui passant
 * la fonction qui va bien *)
let add i1 i2 = apply ( + ) i1 i2
let sub i1 i2 = apply ( - ) i1 i2
let mul i1 i2 = apply ( * ) i1 i2

(** Ex2 *)
type t = B | N | R

let permute = List.map (function B -> N | N -> R | R -> B)

let rec permute_bis l =
    match l with
    | [] -> []
    | B::l -> N::permute_bis l
    | N::l -> R::permute_bis l
    | R::l -> B::permute_bis l

let compte_B l =
  List.fold_left (fun a b -> if b=B then a+1 else a) 0 l

let rec compte_B_bis l =
    match l with
    | [] -> 0
    | B::l -> 1 + compte_B_bis l
    | _::l -> compte_B_bis l

let is_B = function
    | B -> true
    | _ -> false

let compte_B_bis2 l =
    List.length (List.filter is_B l)

let plus_grande_sequence_de_B l =
  let f a b =
    let (actu,best) = a in
    if b=B then (actu+1,(max (actu+1) best)) else (0,best)
  in
  let (x,y) = List.fold_left f(0,0) l in y

let plus_grande_sequence_de_B_bis l =
    let rec seq guess count l =
        match l with
        | [] -> max guess count
        | B::l -> seq guess (count+1) l
        | _::l -> seq (max guess count) 0 l
    in
seq 0 0 l
(* variante : f x |> g equivaut a g (f x) *)
let plus_grande_sequence_de_B l =
  List.fold_left
    (fun (mx,cp) x ->
       match x with
      | B -> mx, cp+1
      | _ -> max mx cp, 0)
    (0,0) l
  |> (fun (x,y) -> max x y)

let remplace l =
    let rec last_ele l =
        match l with
        | [] -> failwith "Empty List"
        | [x] -> x
        | x::rest -> last_ele rest
    in
    List.map (fun a -> match a with | B -> last_ele l; | N -> N; |R -> R;) l

let rec last_val l =
        match l with
        | [] -> failwith "Empty List"
        | [x] -> x
        | x::rest -> last_val rest

let remplace_bis l =
    let last = last_val l in
    let rec remp = function
        | B::l -> last::remp l
        | x::l -> last::remp l
        | [] -> []
    in
remp l
(** Ex 3 *)
type monome = { coeff : int; degre : int};;
type polynome = monome list;;

let p = [ {coeff = 1; degre = 5};
          {coeff = (-2); degre = 4};
          {coeff = 3; degre = 1};
          {coeff = 1; degre = 0}; ]

let afficher_m m =
  let rec translate m =
    match m with
    | {coeff = x; degre = 1} -> Printf.printf "%iX" x
    | {coeff = x; degre = 0} -> Printf.printf "%i" x
    | {coeff = x; degre = y} -> Printf.printf "%iX^%i" x y in
translate m

let afficher_p = function
  | [] -> Printf.printf "0" (* cas du polynôme nul *)
  | m::l -> afficher_m m; (* on met à part le premier coefficient comme le suggère
l'énoncé *)
            let rec aff = function
              | [] -> ()
              | m::l ->
                 if m.coeff >= 0 then Printf.printf "+"; (* cette ligne permet
d'afficher + *)
                 afficher_m m; aff l in
            aff l

let deriver p =
    let rec trans_m m =
        match m with
        | {coeff = x; degre = 1} -> Printf.printf "%d" x
        | {coeff = x; degre = y} -> Printf.printf "%dX%d" (y-1) ((y-1)*x)
    in

    let rec aux l =
        match l with
        | [] -> []
        | x::rest -> trans_m x; aux rest
    in
aux p

let rec somme p1 p2 =
  match p1, p2 with
  (* si l'un est vide, on renvoie l'autre *)
  | [], _ -> p2
  | _, [] -> p1
  | m1::l1, m2::l2 ->
    (* si un monome est de degre strictement superieur, on l'ajoute en tete
     * de la liste et recolle au resultat de la somme du reste de sa liste avec
     * l'autre liste *)
    if m1.degre > m2.degre then m1 :: (somme l1 p2)
    else if m1.degre < m2.degre then m2 :: (somme p1 l2)
    (* sinon les 2 monomes sont du meme degre : on les additionne et on recolle
     * avec la somme des deux restes de liste *)
    else
      let m = { coeff = m1.coeff + m2.coeff; degre = m1.degre } in
      (* attention si les deux monomes s'annulent on n'ajoute pas le resultat *)
      if m.coeff = 0 then somme l1 l2
      else m :: (somme l1 l2)
(* calcul de puissance : cf l'algorithme du cours 2 d'IPF *)
let rec pow x n =
  if n < 0 then 0 (* ¯\_(ツ)_/¯ *)
  else if n = 0 then 1
  else
    let y = pow x (n/2) in
    if n mod 2 = 0 then y * y else y * y * x

let rec evaluer p x =
  let rec aux acc = function
    | [] -> acc
    (* on evalue le monome en tete et on ajoute le resultat a l'accumulateur *)
    | m :: p -> aux (acc + m.coeff * pow x m.degre) p
  in
  aux 0 p

(* variante plus courte *)
let rec evaluer p x =
  List.fold_left (fun acc m -> acc + m.coeff * pow x m.degre) 0 p
```