# Devoir surveillé n° 1
# Éléments de correction

## Exercice 1

1.
```
let somme x y =
  match x,y with
  | Flottant a, Flottant b -> Flottant (a +. b)
  | Flottant a, Entier b -> Flottant (a +. float_of_int b)
  | Entier a, Flottant b -> Flottant (float_of_int a +. b)
  | Entier a, Entier b -> Entier (a+b)
;;
```

2. (a)
```
let p = { nom = "Quatre Fromages" ;
          base = Tomate ;
          prix = 9.9};;
```

(b)
```
let rec prix_total pizzas =
  match pizzas with
  | [] -> 0.
  | t::q -> t.prix +. prix_total q
;;
```

## Exercice 2

1.
```
let suivant u =
  if u mod 2 = 0
  then u/2
  else 3*u + 1
;;
```

2.
```
let rec vol n =
  match n with
  | 1 -> [1]
  | _ -> n :: vol (suivant n)
;;
```

3.
```
let rec altitude_max n =
  match n with
  | 1 -> 1
  | _ -> let m = altitude_max (suivant n) in
         if n > m then n else m
;;
```

4. Dans ce qui suit, la fonction `vol_alt` prend en argument un entier `p` correspondant à l'altitude courante, un entier `lgr` correspondant à la durée du vol en altitude précédemment atteinte (0 si le vol n'est pas en altitude) et un entier `lgr_max` correspondant à la durée du vol en altitude de durée maximale déjà rencontrée.

```
let rec duree_vol_alt n =
  let rec vol_alt p lgr lgr_max =
    match p with
    | 1 -> lgr_max
    | _ -> if p > n
           then if lgr + 1 > lgr_max
                then vol_alt (suivant p) (lgr + 1) (lgr + 1)
                else vol_alt (suivant p) (lgr + 1) lgr_max
           else vol_alt (suivant p) 0 lgr_max
  in vol_alt n 0 0
;;
```

## Exercice 3

1.
```
let ensemble n =
  let rec construit p liste =
    match p with
    | 0 -> liste
    | _ -> construit (p-1) (p::liste)
  in construit n []
;;
```

2. (a)
```
let rec add l x =
  match l with
  | [] -> [x]
  | t::q -> if t > x
            then x::l
            else
              if t = x
              then l
              else t::(add q x)
;;
```

```
let rec subb l x =
  match l with
  | [] -> []
  | t::q ->
     if t > x
     then l
     else
       if t = x
       then q
       else t::(subb q x)
;;
```

(b)
```
let rec reunion l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | t1::q1, t2::q2 -> if t1 < t2
                      then t1::(reunion q1 l2)
                      else
                        if t2 < t1
                        then t2::(reunion l1 q2)
                        else t1::(reunion q1 q2)
;;

let rec intersection l1 l2 =
  match l1, l2 with
  | [], _ -> []
  | _, [] -> []
  | t1::q1, t2::q2 -> if t1 = t2
                      then t1::(intersection q1 q2)
                      else
                        if t1 < t2
                        then intersection q1 l2
                        else intersection l1 q2
;;
```

3. (a) La première fonction proposée se base sur l'indication donnée dans l'énoncé. On remarque néanmoins que cela oblige à ajouter $n$ en fin de liste pour toutes les parties à $p-1$ éléments de $E_{n-1}$.
La fonction `ajout_n` ajoute l'élément `n` à toutes les listes de la liste de listes passée en argument.

```
let rec parties n p =
  match p with
  | 0 -> [[]]
  | _ -> if p > n
         then []
         else
           let rec ajout_n l =
             match l with
             | [] -> []
             | t::q -> (add t n) ::(ajout_n q)
           in (parties (n-1) p) @ ajout_n (parties (n-1) (p-1))
;;
```

Une autre possibilité consiste à construire $E_n$ puis à considérer les éléments dans l'ordre croissant.
La fonction `List.map` a été utilisée ici pour raccourcir la solution ; il est conseillé de savoir la réécrire.

```
let parties n p =
  let rec pcombinaisons ens p =
    match p with
    | 0 -> [[]]
    | _ ->
       if p < 0
       then []
       else
         match ens with
         | [] -> []
         | t::q ->
            List.map (fun l -> add l t) (pcombinaisons q (p-1))
            @ pcombinaisons q p
  in pcombinaisons (ensemble n) p
;;
```

(b) On appelle `parties 3 2`. La fonction réalise deux appels récursifs :

- `parties 2 2`, qui elle-même appelle
  - `parties 1 2`, qui renvoie `[]`
  - `parties 1 1`, qui appelle `parties 0 1`, qui renvoie `[]`), et `parties 0 0`, qui renvoie `[[]]` ; 1 est ajouté à toutes les listes de `[[]]`, cet appel renvoie `[[1]]`.

  puis 2 est ajouté à toutes les listes de `[[1]]`, `parties 2 2` renvoie `[[1, 2]]`
- `parties 2 1`, qui elle-même appelle
  - `parties 1 1`, qui renvoie `[[1]]` (voir au dessus)
  - `parties 1 0`, qui renvoie `[[]]`.

  puis 2 est ajouté à toutes les listes de `[[]]`, `parties 2 1` renvoie `[[1],[2]]`

Enfin, `parties 3 2` revoie la concaténation du résultat de `parties 2 2` et de la liste obtenue en ajoutant 3 à toutes les listes de `parties 2 1`, on obtient bien `[[1, 2], [1, 3],[2, 3]]`