Exercice 1 : Suppression du n-ième élément d'une liste

Q. 1 Définir une fonction prenant en argument un entier n et une liste l et calculant la liste l privée de son n-ième élément.

Solution

```
let rec suppression (l: 'a list) (n: int): 'a list =
  match l, n with
  | x :: q, 0 -> q
  | x :: q, _ -> x :: (suppression q (n - 1))
  | []    , _ -> []
```

Exercice 2 : Mélange

Dans cet exercice, on s'autorise à utiliser la fonction `List.rev : 'a list -> 'a list` calculant le miroir d'une liste.

Q. 1 Définir une fonction `melange: 'a list -> 'a list` prenant en paramètre une liste $[x_0; x_1; x_2; \ldots; x_{n-2}; x_{n-1}]$ et retournant la liste $[x_0; x_{n-1}; x_1; x_{n-2}; x_2; \ldots]$

Solution

```
let rec melange (l: 'a list): 'a list =
  match l with
  | []     -> []
  | p :: q -> p :: (melange (List.rev q))
```

Q. 2 Si la réponse à la première question est de complexité quadratique en la taille de la liste initiale, même question mais avec une complexité linéaire.

Solution

```
let melange_bis (l: 'a list): 'a list =
  (* intercale les éléments des listes passées en paramètre, jusqu'à
     obtention d'une liste de taille n *)
  let rec intercale (l1: 'a list) (l2: 'a list) (n: int): 'a list =
    match l1, l2 with
    (* Ce premier cas ne devrait pas arriver *)
    | [], []                  -> []
    | x :: q, [] | [], x :: q -> x :: (intercale q [] (n - 1))
    | x1 :: q1, x2 :: q2      ->
      if n >= 2 then x1 :: x2 :: (intercale q1 q2 (n - 2))
      else if n = 1 then x1 :: []
      else []
  in intercale l (List.rev l) (List.length l)
```

Exercice 3 : Parmi

Q. 1 Donner une fonction `k_parmi_n : 'a list -> int -> 'a list list` prenant en paramètre une liste l et un entier k et calculant l'ensemble de toutes les sous-listes de k éléments de la

liste l (on devra garder les éléments dans l'ordre d'apparition dans `l`). Ainsi `k_parmi_n [1; 2; 3; 4] 2` devra produire `[[1; 2]; [1; 3]; [1; 4]; [2; 3]; [2; 4]; [3; 4]]`.

Solution

```
let rec k_parmi_n (l: 'a list) (k: int): 'a list list =
  match l, k with
  | _, 0      -> [ [] ]
  | [], _     -> []
  | x :: q, _ ->
    List.map (fun r -> x :: r) (k_parmi_n q (k - 1))
    @ (k_parmi_n q k)
```

Exercice 4 : Un sur deux

Q. 1 Donner une fonction prenant en argument une liste l et calculant la liste obtenue en prenant un élément sur deux de l.

Solution

```
let rec un_sur_deux (l: 'a list): 'a list =
  match l with
  | [] | _ :: [] -> []
```

Exercice 5 : Partition listes

Q. 1 Implémenter la fonction `partition: ('a -> bool) -> 'a list -> 'a list * 'a list` prenant en argument un prédicat et une liste et calculant la paire de la liste des éléments satisfaisant le prédicat et ceux ne le satisfaisant pas.

Solution

```
let rec partition (p: 'a -> bool) (l: 'a list) : 'a list * 'a list =
  match l with
  | []     -> [], []
  | x :: q ->
    let lv, lf = partition p q in
    if (p x) then (x :: lv, lf) else (lv, x :: lf)
```

Ou une version récursive terminale.

```
let partition (p: 'a -> bool) (l: 'a list) : 'a list * 'a list =
  (* accumule, à l'envers, dans lv, (resp. lf) les éléments de l qui
     vérifient (resp. ne vérifient pas) le prédicat p *)
  let rec aux_partition (l: 'a list) (lv: 'a list) (l2: 'a list)
    : 'a list * 'a list =
    match l with
    | []               -> List.rev lv, List.rev l2
    | x :: q when (p x) -> aux_partition q (x :: lv) l2
    | x :: q           -> aux_partition q lv (x :: l2)
  in aux_partition l [] []
```

On définit le type `type 'a option = None | Some of 'a`

Q. 2 Implémenter une fonction `filter_map : ('a -> 'b option) -> 'a list -> 'b list` telle que `filter_map f l` applique `f` à chaque élément de `l`, ôte de la liste les éléments pour lesquels l'appel produit `None` et calcule la liste des éléments se trouvant dans le `Some` sinon.

Solution

```
let rec filter_map (p: 'a -> 'b option) (l: 'a list): 'b list =
  match l with
  | []     -> []
  | x :: q ->
    ( match p x with | None   -> [ ] | Some y -> [y] ) @ (filter_map p q)
```

Exercice 6 : Énumération des listes de booléens

Q. 1 Donner une fonction `generate : int -> bool list list` prenant en argument un entier n et générant la liste de toutes les listes de booléens de taille n.

Solution

```
let rec generate (n: int): bool list list =
  (* Calcule la liste de toutes les listes de booléens de taille n +
     |prefixe| dont prefixe est un préfixe *)
  let rec aux_gen (prefixe: bool list) (n: int): bool list list =
    if n = 0 then [ prefixe ]
    else
        (aux_gen (true  :: prefixe) (n - 1))
      @ (aux_gen (false :: prefixe) (n - 1))
  in aux_gen [] n
```

Q. 2 Donner une fonction `enumerate : int -> (int -> bool list)` prenant en argument un entier n et calculant une fonction f réalisant une bijection de $[\![0, 2^n - 1]\!]$ dans l'ensemble des listes de booléens de longueur n.

Solution

```
let enumerate (n: int) =
  (* decompose i < 2^n en binaire sur n bits. *)
  let rec decompose_binaire (n: int) (i: int): bool list =
    if n = 0 then []
    else (i mod 2 = 0) :: decompose_binaire (n - 1) (i / 2)
  in decompose_binaire n
```

Exercice 7 : Suppression des occurrences consécutives

Q. 1 Écrire une fonction `compresse : 'a list -> 'a list` qui élimine les éléments consécutifs identiques d'une liste. Par exemple :

```
# compresse [1; 1; 2; 3; 3; 3; 4; 4];;
- : int list = [1; 2; 3; 4]
```

Solution

```
let rec compresse (l: 'a list): 'a list =
  match l with
  | []     -> []
  | x :: q ->
    match (compresse q) with
    | y :: q' when x = y -> y :: q'
    | l                  -> x :: l
```

Ou en version récursive terminale.

```
let compresse_tr (l: 'a list): 'a list =
  (* aux compresse la liste l, en accumulant le résultat dans res. *)
  let rec aux (l: 'a list) (res: 'a list): 'a list =
    match l, res with
    | x :: l', y :: r' when x = y -> aux l' (y :: r')
    | x :: l', _                  -> aux l' (x :: res)
    | [], _                       -> List.rev res
  in aux l []
```

Exercice 8 : Liste 2

Q. 1 Implanter une fonction `combine : 'a list -> 'b list -> ('a * 'b) list` prenant en argument deux listes de même longueur et les combinant : `combine [a1; ...; an] [b1; ...; bn] = [(a1, b1); ...; (an, bn)]`.

Solution

```
(* Calcule la liste des couples [(x, y) pour y in l2]. *)
let rec encouple (x: 'a) (l2: 'b list): ('a * 'b) list =
  match l2 with
  | []      -> []
  | y :: l2' -> (x, y) :: (encouple x l2')

let rec combine (l1: 'a list) (l2: 'b list): ('a * 'b) list =
  match l1 with
  | []      -> []
  | x :: l1' -> (encouple x l2) @ (combine l1' l2)
```

Ou avec un itérateur.

```
let rec combine (l1: 'a list) (l2: 'b list): ('a * 'b) list =
  match l1 with
  | []      -> []
  | x :: l1' -> (List.map (fun y -> (x, y)) l2) @ (combine l1' l2)
```

Ou en version récursive terminale.

```
let combine (l1: 'a list) (l2: 'b list): ('a * 'b) list =
  (* coeur récursif, simulant deux boucles imbriquées : boucle extérieur
```

```
    sur l2, boucle intérieur sur l1_l (qui est donc "remis" à l1 à chaque
    pas d'itération sur l2). lres accumule le résultat. *)
  let rec aux (l1_l: 'a list) (l2: 'b list) (lres: ('a * 'b) list): ('a * 'b) list =
    match l1_l, l2 with
    | [], y :: l2'       -> aux l1 l2' lres
    | x :: l1', y :: l2' -> aux l1' l2 ((x, y) :: lres)
    | _, []              -> List.rev lres
  in aux l1 l2 []
```

Exercice 9 : Permutation

Q. 1 Donner une fonction `permutations : 'a list -> 'a list list` prenant en argument une liste et retournant l'ensemble de toutes les permutations de cette liste. L'ordre des permutations dans la liste résultat n'a pas d'importance.

Solution

```
(* Calcule la liste de toutes les listes pouvant être obtenues en insérant
   x à toutes les positions possibles dans l. *)
let intercale (x: 'a) (l: 'a list): 'a list list =
  (* coeur récursif, debut est le debut de la liste l, et fin la fin
     associée à debut. *)
  let rec zip (debut: 'a list) (fin: 'a list): 'a list list =
    let n_intercalement = List.rev debut @ [x] @ fin in
    match fin with
    | y :: fin' -> n_intercalement :: (zip (y :: debut) fin')
    | []        -> [n_intercalement]
  in zip [] l

(* Calcule la liste de toutes les listes pouvant être obtenues en insérant
   x à toutes les positions possibles dans toutes les listes de l. *)
let rec intercalel (x: 'a) (l: 'a list list): 'a list list =
  match l with
  | []      -> []
  | p :: l' -> (intercale x p) @ (intercalel x l')

let rec permutations (l: 'a list): 'a list list =
  match l with
  | []      -> [ [] ]
  | x :: l' -> intercalel x (permutations l')
```

Avec un peu d'itérateurs.

```
let rec permutations (l: 'a list): 'a list list =
  match l with
  | []      -> [ [] ]
  | x :: l' ->
    List.fold_left (fun acc permut ->
        (intercale x permut) @ acc
      ) [] (permutations l')
```