Feuille de révisions n°1 - Programmation en OCaml : manipulation de listes

1 Pied à l'étrier

R. 1-1 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction lèvera une exception `Invalid_argument` dans le cas où la liste est vide. On rappelle que l'exception `Invalid_argument` est prédéfinie en OCaml, elle doit être accompagnée d'une chaîne de caractères. Par exemple : l'évaluation de l'expression `raise (Invalid_argument("ceci est un message d'erreur"))` lève l'exception `Invalid_argument("ceci est un message d'erreur")`.

Solution

```
let rec last (l: 'a list): 'a =
  (* Retourne le dernier élément de l. Lève l'exception Invalid_argument
     dans le cas où la liste est vide. *)
  match l with
  | x :: [] -> x
  | _ :: l' -> last l'
  | _       -> raise (Invalid_argument "liste vide")
```

R. 1-2 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction doit retourner `None` si la liste est vide et `Some(x)` si la liste contient au moins un élément et que le dernier élément est `x`.

Solution

```
let rec last (l: 'a list): 'a option =
  (* Retourne le dernier élément de l. Retourne None si la liste est vide
     et Some(x) si la liste contient au moins un élément et que le dernier
     est x. *)
  match l with
  | x :: [] -> Some(x)
  | _ :: l' -> last l'
  | _       -> None
```

R. 1-3 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On n'utilisera pas de fonctionnelle d'itération.

Solution

```
let rev (l: 'a list): 'a list =
  (* Calcule le miroir de la liste l. *)
  let rec aux (l: 'a list) (res: 'a list) =
    match l with
```

```
    | []     -> res
    | p :: q -> aux q (p :: res)
  in aux l []
```

R. 1-4 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

Solution

```
let rev (l: 'a list): 'a list =
  (* Calcule le miroir de la liste l. *)
  List.fold_left (fun acc x -> x :: acc) [] l
```

R. 1-5 Écrire une fonction permettant de calculer la concaténation de deux listes. On n'utilisera pas de fonctionnelle d'itération.

Solution

```
let concat (lg: 'a list) (ld: 'a list): 'a list =
  (* Calcule la concaténation des listes lg et ld (dans ce sens). *)
  let rec verse (todo: 'a list) (lres : 'a list): 'a list =
    (* Calcule la concaténation du miroir de todo et de lres. *)
    match todo with
    | []   -> lres
    | a::q -> verse q (a :: lres)
  in verse (List.rev lg) ld
```

R. 1-6 Écrire une fonction permettant de calculer la concaténation de deux listes. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

Solution

```
let concat (lg: 'a list) (ld: 'a list): 'a list =
  (* Calcule la concaténation des listes lg et ld (dans ce sens). *)
  List.fold_left (fun accu elem -> elem::accu) ld (List.rev lg)
```

R. 1-7 Écrire une fonction prenant en argument une liste `l` et un élément `x` et calculant la liste `l` privée de toutes les occurrences de l'élément `x`.

Solution

```
let prive_de (l: 'a list) (x: int) : 'a list =
  (* Calcule une liste contenant les éléments de l qui ne sont pas x. *)
  let rec aux (ll: 'a list) (lres: 'a list) : 'a list =
    (* Calcule ll privée de x concaténée au miroir de lres. *)
    match ll with
    | []   -> List.rev lres
    | a::q -> if a <> x then aux q (a :: lres) else aux q lres
  in aux l []
```

R. 1-8 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction devra être récursive et ne pas faire appel à une fonction récursive auxiliaire.

Solution

```
let decoupe (l: 'a list) (i: int) : 'a list * 'a list =
  (* Calcule un couple de listes : la liste des éléments de l d'indice
     strictement inférieurs à i, la liste des éléments de l d'indice
     supérieurs à i. *)
  let rec aux (g: 'a list) (d: 'a list) (i: int) =
    if i = 0 then List.rev g, d
    else
      match d with
      | []     -> List.rev g, d
      | p :: q -> aux (p :: g) q (i-1) in
  aux [] l i
```

R. 1-9 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction devra être récursive et ne pas faire appel à des fonctions récursives auxiliaires.

Solution

```
let rec decoupe (l: 'a list) (i: int) : 'a list * 'a list =
  (* Calcule un couple de listes : la liste des éléments de l d'indice
     strictement inférieurs à i, la liste des éléments de l d'indice
     supérieurs à i. *)
  if i = 0 then [], l
  else match l with
    | []     -> [], []
    | p :: q ->
      let g, d = decoupe q (i-1) in
      (p :: g, d)
```

R. 1-10 Écrire une fonction prenant en argument une liste `l` et la découpant en deux listes, dans la première on rangera les éléments d'indices pairs dans `l`, dans la seconde on rangera les éléments d'indices impairs dans `l`. L'ordre des éléments dans les listes résultats devra être celui de la liste d'entrée.

Solution

```
let un_sur_deux (l: 'a list) : ('a list * 'a list) =
  (* Retourne un couple de listes : les éléments de l d'indice pair, les
     éléments de l d'indice impair. *)
  let rec aux (lat: 'a list) (lres1: 'a list) (lres2: 'a list)
    : 'a list * 'a list =
    (* lat est la liste restant à traitée, lres1 est la liste des éléments
       d'indice pairs déjà traités, lres2 est la liste des éléments
       d'indice impairs déjà traités. lres1 et lres2 sont à l'envers. *)
```

```
    match lat with
    | []          -> (lres1     , lres2)
    | a::[]       -> (a :: lres1, lres2)
    | a::b::suite -> aux suite (a :: lres1) (b :: lres2)
  in
  let (lr1, lr2) = aux l [] [] in
  (List.rev lr1, List.rev lr2)
```

R. 1-11 Écrire une fonction testant si une liste est un palindrome. Cette fonction devra être en $\mathcal{O}(n)$ où n est la longueur de la liste.

Solution

```
let est_palindrome (l: 'a list) : bool =
  (* Teste si la liste l est un palindrome. *)
  l = (rev l)
```

2 Listes de listes

R. 1-12 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On utilisera la fonction de concaténation de listes `@`, en prenant garde à ne pas obtenir une complexité quadratique.

Solution

```
let flatten_1 (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  let rec aux (ll: 'a list list) (lres : 'a list) : 'a list =
    (* ll est la liste des listes encore à traiter, lres est la liste (à
       l'envers) des éléments déjà aplaties. *)
    match ll with
    | []   -> lres
    | a::q -> aux q ((List.rev a) @ lres)
    (* on ajoute à gauche : @ de coût linéaire en son opérande gauche *)
```

R. 1-13 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On demande deux versions : l'une avec deux fonctions mutuellement récursives, chacune récursives terminales, l'autre avec une unique fonction auxiliaire récursive terminale.

Solution

Avec deux fonctions mutuellement récursives terminales :

```
let flatten_mtr (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
```

```
  let rec grand_pas (todo: 'a list list) (res: 'a list): 'a list =
    (* todo est la liste des éléments encore à traiter. res est la liste
       des éléments déjà aplatis. *)
    match todo with
    | []      -> List.rev res
    | x :: xs -> petit_pas x xs res
  and petit_pas (une_liste: 'a list) (todo: 'a list list) (res: 'a list)
    : 'a list =
    (* une_liste est une liste de la liste initiale l, que l'on transvase
       dans res. todo est la liste des listes encore à traiter. res est la
       liste des éléments déjà aplatis. *)
    match une_liste with
    | []      -> grand_pas todo res
    | y :: ys -> petit_pas ys todo (y :: res)
  in
  grand_pas l []
```

Avec une unique fonction auxiliaire récursive terminale :

```
let flatten_tr (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  let rec aux (todo: 'a list list) (res: 'a list): 'a list =
    (* todo est la liste des éléments encore à traiter. res est la liste
       des éléments déjà aplatis. *)
    match todo with
    | []              -> List.rev res
    | ([]) :: xs      -> aux xs res
    | (y :: ys) :: xs -> aux (ys :: xs) (y :: res)
  in aux l []
```

R. 1-14 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes, on se limitera à des itérations au moyen de la fonctionnelle `List.fold_left`.

Solution

```
let flatten (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  List.rev (
    List.fold_left (fun accu l ->
        List.fold_left (fun acc x ->
            x :: acc
          ) accu l
      ) [] l
  )
```