OCaml - Listes

Aubin SIONVILLE

MPI Clemenceau - 2021-2023

1 Bases

Dernier élément d'une liste

Fonction récursive retournant le dernier élément d'une liste d'un type quelconque.
Cette fonction doit lever une exception si la liste est vide.

```
let rec last (l: 'a list): 'a =
match l with
| x :: [] -> x
| _ :: l' -> last l'
| _ -> raise (Invalid_argument "liste vide");;
```

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

```
let rec last_option (l: 'a list): 'a option =
match l with
| x :: [] -> Some(x)
| _ :: l' -> last_option l'
| _ -> None;;
```

Miroir d'une liste

Fonction retournant la liste miroir d'une liste d'un type quelconque.

```
let rev (l: 'a list): 'a list =
    let rec aux (l: 'a list) (res: 'a list): 'a list =
        match l with
        | [] -> res
        | x :: l' -> aux l' (x :: res)
    in aux l [];;
```

Fonction retournant la liste miroir d'une liste d'un type quelconque. En utilisant **List.fold_left**.

```
let rev_fold (l: 'a list) = List.fold_left (fun res x -> x :: res) [] l;;
```

Concaténation de deux listes

Fonction permettant de calculer la concaténation de deux listes.
Sans fonctionnelle d'itération.

```
let concat (lg: 'a list) (ld: 'a list): 'a list =
    let rec verse (todo: 'a list) (lres: 'a list): 'a list =
        match todo with
        | [] -> lres
        | x :: l' -> verse l' (x :: lres)
    in verse (List.rev lg) ld;;
```

Fonction permettant de calculer la concaténation de deux listes. En utilisant **List.fold_left**.

```
let concat_fold (lg: 'a list) (ld: 'a list): 'a list =
    List.fold_left (fun res x -> x :: res) ld (List.rev lg);;
```

Suppression de toutes les occurrences d'un élément

Fonction prenant en arguments une liste l et un élément x et calculant la liste l privée de toutes les occurrences de x.

```
let rec remove_all (l: 'a list) (x: 'a): 'a list =
    match l with
    | [] -> []
    | y :: l' ->
        if x = y then
            remove_all l' x
        else y :: remove_all l' x;;
```

Découpage de listes

Fonction prenant en arguments 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.
La fonction devra être récursive et ne pas faire appel à des fonctions récursives auxiliaires.

```
let decoupe (l: 'a list) (i: int): ('a list * 'a list) =
    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)
            | x :: l' -> aux (x :: g) l' (i - 1)
    in aux [] l i;;
```

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.

```
let un_sur_deux (l: 'a list): ('a list * 'a list) =
    let rec aux (lat: 'a list) (lres1: 'a list) (lres2: 'a list) =
        match lat with
        | [] -> (lres1, lres2)
        | a::[] -> (a::lres1, lres2)
        | a::b::l' -> aux l' (a::lres1) (b::lres2)
    in let (l1, l2) = aux l [] []
    in (List.rev l1, List.rev l2);;
```

Palindrome

Fonction testant si une liste est un palindrome.
La fonction devra être en $\mathcal{O}(n)$ où n est la longueur de la liste.

```
let palindrome (l: 'a list): bool =
    l = rev(l);;
```

Listes de listes

Mise à plat

Fonction prenant en argument une liste l de listes et retournant la liste des éléments des sous-listes de l, en utilisant @ sans être en $\mathcal{O}(n^2)$.

```
let flatten_a (l: 'a list list): 'a list =
    let rec aux (ll: 'a list list) (lres: 'a list): 'a list =
        match ll with
        | [] -> lres
        | x :: l' -> aux l' ((List.rev x) @ lres)
    in aux l [];;
```

Fonction prenant en argument une liste l de listes et retournant la liste des éléments des sous-listes de l, en utilisant deux fonctions récursives auxiliaires. La fonction doit être récursive terminale.

```
let flatten_rt (l: 'a list list): 'a list =
    let rec grand_pas (todo: 'a list list) (lres: 'a list): 'a list =
        match todo with
        | [] -> List.rev lres
        | x :: l' -> petit_pas x l' lres
    and petit_pas (une_list: 'a list) (todo: 'a list) (lres: 'a list): 'a list =
        match une_list with
        | [] -> grand_pas todo lres
        | x :: l' -> petit_pas l' todo (x :: lres)
    in grand_pas l [];;
```

Fonction prenant en argument une liste l de listes et retournant la liste des éléments des sous-listes de l, en utilisant **List.fold_left**.

```
let flatten_f (l: 'a list list): 'a list =
    List.fold_left ( fun acc l ->
        List.fold_left (fun acc x ->
            x :: acc
        ) acc l
    ) [] l
    |> List.rev;;
```

Listes et comparaisons

Mini et maxi

Fonction prenant en argument une liste l et retournant un couple contenant le minimum et le maximum de l. La fonction doit être récursive et ne pas faire appel à des fonctions récursives auxiliaires. On lève une exception si la liste est vide.

```
let min_max (l: 'a list): 'a * 'a =
    match l with
    | [] -> raise (Invalid_argument "Liste vide")
    | x :: [] -> (x, x)
    | x :: l' ->
        let mini, maxi = min_max l
        in (min x mini, max x maxi);;
```

Fonction prenant en argument une liste l et retournant un couple contenant le minimum et le maximum de l. La fonction doit être récursive terminale et peut utiliser une fonction récursive auxiliaire. On lève une exception si la liste est vide.

```
let min_max_rt (l: 'a list): 'a * 'a =
    let rec aux (l: 'a list) (mini: 'a) (maxi: 'a): 'a * 'a =
        match l with
        | [] -> (mini, maxi)
        | x :: l' -> aux l' (min x mini) (max x maxi)
    in
    match l with
    | [] -> raise (Invalid_argument "Liste vide")
    | x :: l' -> aux l' x x;;
```

Fonction prenant en argument une liste l et retournant un couple contenant le minimum et le maximum de l en utilisant **List.fold_left**. On lève une exception si la liste est vide.

```
let min_max_f (l: 'a list): 'a * 'a =
    match l with
    | [] -> raise (Invalid_argument "Liste vide")
    | x :: l' ->
        List.fold_left (fun (mini, maxi) x ->
            (min x mini, max x maxi)
        ) (x, x) l';;
```

Sous-séquences croissantes

Fonction prenant en argument une liste l et retournant une liste de sous-séquences de l croissantes maximales pour l'extension à gauche.
Exemple : $[1,2,4,8,5,3,4,5,1] -> [[1,2,4,8],[5],[3,4,5],[1]]$

```
let sous-sequences-croissantes (l: 'a list): ('a list list) =
    let rec aux (ll: 'a list) (ltemp: 'a list) (lres: 'a list list): 'a list =
        match ll, ltemp with
        | [], [] -> List.rev lres
        | [], _ -> List.rev (ltemp :: lres)
        | x :: l', [] -> aux l' (x :: []) lres
        | x :: l', y :: l'' ->
            if x >= y then aux l' (x :: ltemp) lres
            else aux l' (x :: []) ((List.rev ltemp) :: lres)
    in aux l [] [];;
```

Tris

Fonction prenant en arguments une liste de $'a$ élément, un entier k et une fonction sur les $'a$ éléments à valeur dans $[\![0, k]\!]$ et retournant la liste des éléments de l triés par ordre croissant de leur valeur par la fonction f. La complexité de l'algorithme doit être en $O(\max(n, k))$.

```
let rangement (l: 'a list) (k: int) (f: 'a -> int): 'a list =
    let aux = Array.make k [] in
    List.iter (fun x ->
        let i = (f x) in
        aux.(i) <- x :: aux.(i)
    ) l;
    let ll = Array.to_list aux in
    List.flatten ll;;
```

Utilisations classiques

Tri fusion

Fonction prenant en argument une liste l et la triant par un tri fusion.

```
let rec fusion (l1: 'a list) (l2: 'a list): 'a list =
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | x :: l1', y :: l2' ->
        if x <= y then x :: fusion l1' l2
        else y :: fusion l1 l2';;

let rec tri_fusion (l: 'a list): 'a list =
    match l with
    | [] -> []
    | [x] -> [x]
    | _ ->
        let l1, l2 = List.split_at (List.length l / 2) l in
        fusion (tri_fusion l1) (tri_fusion l2);;
```

Files

Implémentation du type File avec deux listes. On assurera une complexité en $\mathcal{O}(1)$ pour chaque opération.

```
type 'a file = 'a list * 'a list;;

let file_vide: 'a file = ([], []);;

let est_vide (f: 'a file): bool =
    match f with
    | [], [] -> true
    | _ -> false;;

let enfiler (f: 'a file) (x: 'a): 'a file =
    match f with
    | l1, l2 -> (x :: l1, l2);;

let defiler (f: 'a file): 'a file =
    match f with
    | [], [] -> raise (Invalid_argument "File vide")
    | l1, [] -> (List.rev l1, [])
    | l1, x :: l2 -> (l1, l2);;

let tete (f: 'a file): 'a =
    match f with
    | [], [] -> raise (Invalid_argument "File vide")
    | l1, [] -> List.hd (List.rev l1)
    | l1, x :: l2 -> x;;
```

Permutations

Fonction prenant en argument une liste l et retournant une liste de toutes les permutations de l.

```
let rec permutations (l: 'a list): 'a list list =
    match l with
    | [] -> [[]]
    | x :: l' -> let ll = permutations l'
        in List.fold_left (fun lres l ->
            let rec aux (l: 'a list) (lres: 'a list list): 'a list list =
                match l with
                | [] -> lres
                | x :: l' -> aux l' ((x :: l) :: lres)
            in aux l lres
        ) [] ll;;
```

Liste des booléens

Fonction prenant en argument un entier n et retournant la liste 2^n listes contenant n booléens.

```
let rec liste_bool (n: int): bool list list =
    if n = 0 then [[]]
    else
        let ll = liste_bool (n - 1) in
        List.fold_left (fun lres l ->
            (true :: l) :: (false :: l) :: lres
        ) [] ll;;
```