

Chapitre 4 - Fonctions d'ordre supérieur

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Fonctions d'ordre supérieur

Une fonction **d'ordre supérieur** est une fonction qui

- renvoie une fonction, ou
- prend en paramètre une fonction, ou
- fait les deux

Fonction qui renvoie une fonction

On a vu qu'il n'y a que des fonctions à un paramètre, donc :

```
let f (x : int) (y : int) = x + y
```

n'est en fait que :

```
let f (x : int) : int -> int =  
  fun y -> x + y
```

f est donc une fonction qui :

- prend un paramètre x entier
- retourne la fonction `fun y -> x + y`, où x est la valeur du paramètre

Toute fonction à plusieurs paramètres est donc techniquement une fonction d'ordre supérieur car elle retourne une fonction.

Fonction qui prend en paramètre une fonction

Une fonction peut prendre une autre fonction en paramètre :

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

apply est une fonction qui :

- prend en argument une fonction *f*
 - qui elle prend en argument une valeur de type 'a et renvoie une valeur de type 'b
- prend en argument une valeur de type 'a
- renvoie une valeur de type 'b

Utiliser les types pour guider la définition

Pour définir une fonction, on commence souvent par écrire le type qu'on souhaite, puis on peut compléter en regardant les possibilités :

let `apply (f : 'a -> 'b) (x : 'a) : 'b = ???`

- On a une seule façon de produire des 'b : avec f
- On a une seule façon d'avoir un 'a pour f : c'est x
- Donc, le corps de la fonction doit être f x

Fonction d'ordre supérieur : sommes

Les fonctions d'ordre supérieur viennent aider à généraliser certaines constructions.

Si on veut calculer la somme des nombres entre a et b :

$$\sum_{i=a}^b i$$

Fonction d'ordre supérieur : sommes

Les fonctions d'ordre supérieur viennent aider à généraliser certaines constructions.

Si on veut calculer la somme des nombres entre a et b :

$$\sum_{i=a}^b i$$

```
let rec sum_numbers (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    a +. (sum_numbers (a +. 1.) b)
```

Fonction d'ordre supérieur : sommes

On veut calculer une somme différente :

$$\sum_{i=a}^b i^3$$

Fonction d'ordre supérieur : sommes

On veut calculer une somme différente :

$$\sum_{i=a}^b i^3$$

```
let rec sum_cubes (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    (a *. a *. a) +. (sum_cubes (a +. 1.) b)
```

Fonction d'ordre supérieur : sommes

Encore une autre (qui tends vers $\frac{\pi}{8}$) :

$$\sum_{i=a}^b \frac{1}{4i(4i+2)}$$

Fonction d'ordre supérieur : sommes

Encore une autre (qui tends vers $\frac{\pi}{8}$) :

$$\sum_{i=a}^b \frac{1}{4i(4i+2)}$$

```
let rec sum_pi (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    (1. /. (a *. (a +. 2.))) +. (sum_pi (a +. 4.) b)
```

Fonction d'ordre supérieur : sommes

Toutes ces fonctions sont similaires :

- on va de a jusque b
- on ajoute quelque chose qui dépend de a à la somme
- on itère avec une valeur suivante de a

On peut donc généraliser cela :

```
let rec sum (term : float -> float)
            (next : float -> float)
            (a : float) (b : float) : float =
  if a > b then
    0.
  else
    (term a) +. (sum term next (next a) b)
```

Fonction d'ordre supérieur : sommes

Avant généralisation :

```
let rec sum_numbers (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    a +. (sum_numbers (a +. 1.) b)
```

Après :

```
let inc (x : float) : float = x +. 1.  
let identity (x : 'a) : 'a = x  
  
let sum_numbers' (a : float) (b : float) : float =  
  sum identity inc a b
```

Fonction d'ordre supérieur : sommes

Avant généralisation :

```
let rec sum_cubes (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    (a *. a *. a) +. (sum_cubes (a +. 1.) b)
```

Après :

```
let cube (x : float) : float = x *. x *. x  
  
let sum_cubes' (a : float) (b : float) : float =  
  sum cube inc a b
```

Fonction d'ordre supérieur : sommes

Avant généralisation :

```
let rec sum_pi (a : float) (b : float) : float =  
  if a > b then  
    0.  
  else  
    (1. /. (a *. (a +. 2.))) +. (sum_pi (a +. 4.) b)
```

Après :

```
let sum_pi' (a : float) (b : float) : float =  
  let pi_term x = 1. /. (x *. (x +. 2.)) in  
  let pi_next x = x +. 4. in  
  sum pi_term pi_next a b
```

Fonction d'ordre supérieur : combinateur de fonction

On a également des fonctions qui “changent” d'autres fonctions :

Par exemple, on peut inverser les arguments d'une fonction :

```
let flip (f : 'a -> 'b -> 'c) : 'b -> 'a -> 'c =  
  fun x y -> f y x
```

- on a une fonction `f` en argument
- on retourne une fonction qui applique `f` avec les arguments inversés

Fonction d'ordre supérieur : combinateur de fonction

```
let flip (f : 'a -> 'b -> 'c) : 'b -> 'a -> 'c =  
  fun x y -> f y x
```

Une définition équivalente :

```
let flip (f : 'a -> 'b) (x : 'a) (y : 'b) : 'c =  
  f y x
```

Fonction d'ordre supérieur : combinateur de fonction

Curryfication :

```
let curry (f : 'a * 'b -> 'c) : 'a -> 'b -> 'c =  
  fun x y -> f (x, y)
```

Décurryification :

```
let uncurry (f : 'a -> 'b -> 'c) : 'a * 'b -> 'c =  
  fun (x, y) -> f x y
```

Fonction d'ordre supérieur : combinateur de fonction

Définitions équivalentes :

```
let curry (f : 'a * 'b -> 'c) (x : 'a) (y : 'b) : 'c =  
    f (x, y)
```

```
let uncurry (f : 'a -> 'b -> 'c) ((x, y) : 'a * 'b) : 'c =  
    f x y
```

Fonction d'ordre supérieur : combinateur de fonction

Composition de fonction ($f \circ g$) :

```
let comp (f : 'b -> 'c) (g : 'a -> 'b) : 'a -> 'c =  
  fun x -> f (g x)
```

Par exemple :

```
# (comp (fun x -> x + 1) (fun x -> x * 3)) 5;;  
- : int = 16
```

Fermeture

Quand on construit une fonction qui dépend de son **environnement lexical**, on parle d'une **fermeture**

- environnement lexical : les liaisons disponibles dans la fonction

```
let addx (x : int) : int -> int =  
  fun y -> x + y
```

- l'environnement lexical de `fun y -> x + y` comprend la liaison `x`
- l'expression `x + y` utilise la liaison `x`
- on dit alors que la fonction **capture** `x`
- elle peut continuer à accéder à la valeur de `x`

Fermeture

```
# let addx x = fun y -> x + y;;  
val addx : int -> int -> int = <fun>  
# let f = addx 3;;  
val f : int -> int = <fun>  
# f 1;;  
- : int = 4
```

Dans d'autres langages

Ces concepts se transposent tout à fait à d'autres langages

Python

```
def flip(f):  
    return lambda x, y: f(y, x)
```

```
def adder(x):  
    return lambda y: x+y
```

Dans d'autres langages

JavaScript

```
function flip(f) {  
  function g(x, y) { return f(y, x) }  
  return g;  
}
```

```
function adder(x) {  
  function f(y) { return x + y; }  
  return f;  
}
```

Fonctions d'ordre supérieur sur les listes

Fonctions d'ordre supérieur sur les listes

L'utilisation des fonctions d'ordre supérieur avec les listes est très fréquente :

- `map`
- `filter`
- `fold`

Parcourir une liste

À partir d'une liste de chaîne de caractère, on veut savoir la longueur de chaque élément dans la liste :

```
# let l = ["hello"; "world"];;  
val l : string list = ["hello"; "world"]  
# length_in_list l;;  
- : int list = [5; 5]
```

Parcourir une liste

```
let rec length_in_list (l : string list) : int list =  
  match l with  
  | [] -> []  
  | head :: tail ->  
    String.length head :: (length_in_list tail)
```

Parcourir une liste

À partir d'une liste d'entier, on souhaite savoir s'ils sont pairs ou pas

```
# let l = [1; 2; 3; 4; 5];;  
val l : int list = [1; 2; 3; 4; 5]  
# pairs_in_list l;  
- : (int * bool) list =  
[(1, false); (2, true); (3, false); (4, true); (5, false)]
```

Parcourir une liste

```
let rec pairs_in_list (l : int list) : (int * bool) list =  
  match l with  
  | [] -> []  
  | head :: tail ->  
    (head, head mod 2 = 0) :: (pairs_in_list tail)
```

Parcourir une liste

Les deux constructions sont très similaires :

```
let rec length_in_list (l : string list) : int list =  
  match l with  
  | [] -> []  
  | head :: tail ->  
    String.length head :: (length_in_list tail)
```

```
let rec pairs_in_list (l : int list) : (int * bool) list =  
  match l with  
  | [] -> []  
  | head :: tail ->  
    (head, head mod 2 = 0) :: (pairs_in_list tail)
```

Parcourir une liste

Quels sont leurs points communs ?

```
let rec g (l : ??? list) : ??? list = match l with  
| [] -> []  
| head :: tail -> ??? :: (g tail)
```

Où ??? sont les seules différences

- on prend en argument une liste d'un certain type, appelons le 'a
- on renvoie une liste d'un autre type, appelons le 'b
- on applique une transformation à head, appelons la f
- f est de type 'a -> 'b

Parcourir une liste

```
let rec g (l : 'a list) : 'b list = match l with  
| [] -> []  
| head :: tail -> (f head) :: (g tail)
```

Mais f est différent pour nos deux cas.

On va le passer en argument également.

La fonction map

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =  
  match l with  
  | [] -> []  
  | head :: tail -> (f head) :: (map tail)
```

La fonction `List.map`

Cette fonction est déjà définie pour nous dans le module `List` : `List.map`

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

`map f [a1; ...; an]` applies function `f` to `a1`, ..., `an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

Parcourir une liste avec map

On peut redéfinir nos fonctions :

```
let length_in_list (l : 'a list) : int =  
  List.map String.length l
```

```
let pairs_in_list (l : int list) : int =  
  List.map (fun x -> (x, x mod 2 = 0)) l
```

Plus d'utilisations de map

```
# List.map abs [1; -4; 9; -9; 12];;  
- : int list = [1; 4; 9; 9; 12]  
  
# List.map Char.code ['a'; 'A'; 'b'; 'B'; 'c'; 'C'];;  
- : int list = [97; 65; 98; 66; 99; 67]  
  
# List.map ((+) 1) [0; 1; 2; 3; 4];;  
- : int list = [1; 2; 3; 4; 5]
```

map dans d'autres langages

Python

```
print(list(map(lambda s: len(s), ["hello", "world", "!"])))
```

Java (≥ 8)

```
someList.stream()  
    .map(s -> s.length())  
    .collect(Collectors.toList())
```

Filtrer une liste

On souhaite garder uniquement les nombres positifs d'une liste

```
# positives [-2; -1; 0; 1; 2];;  
- : int list = [0; 1; 2]
```

Filtrer une liste

```
let rec positives (l : int list) : int list =  
  match l with  
  | [] -> []  
  | head :: tail when head >= 0 -> head :: (positives tail)  
  | _ :: tail -> positives tail
```

Filtrer une liste

On souhaite garder les chaînes non vides

```
# non_empty ["foo"; ""; ""; "bar"; ""];;  
- : string list = ["foo"; "bar"]
```

Filtrer une liste

```
let rec non_empty (l : string list) : string list =  
  match l with  
  | [] -> []  
  | "" :: tail -> non_empty tail  
  | head :: tail -> head :: (non_empty tail)
```

Filtrer une liste

À nouveau, c'est généralisable, on a la forme suivante :

```
let rec f (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | head :: tail when p head -> head :: (f tail)  
  | head :: tail -> f tail
```

Pour un certain prédicat $p : 'a \rightarrow \text{bool}$

Filtrer une liste

C'est la fonction `filter`

```
let rec filter (p : 'a -> bool) (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | head :: tail when p head -> head :: (filter l)  
  | head :: tail -> filter l
```

`p` est un **prédicat**, c'est-à-dire une fonction qui renvoie un booléen.

Filtrer une liste : `filter`

Documentation de `List.filter`:

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

`filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. The order of the elements in the input list is preserved.

Filtrer une liste

À nouveau, on peut raccourcir nos fonctions :

```
let positives (l : int list) : int list =  
  List.filter (fun x -> x > 0) l
```

```
let non_empty (l : string list) : string list =  
  List.filter (fun x -> String.length x > 0) l
```

Plus d'utilisations de filter

```
# List.filter (fun x -> x >= 3) [4; 5; 0; 1; 2; 7];;  
- : int list = [4; 5; 7]  
  
# List.filter (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5];;  
- : int list = [2; 4]  
  
# List.filter  
  (fun s -> String.contains s 'o')  
  ["hello"; "world"; "!"];;  
- : string list = ["hello"; "world"]
```

filter dans d'autres langages

Python

```
print(list(filter(lambda x: x < 5, range(10))))
```

Java (≥ 8)

```
someList.stream()  
    .filter(x -> x < 5)  
    .collect(Collectors.toList())
```

filter_map

Exercice

Définir la fonction

```
val filter_map : ('a -> 'b option) -> 'a list -> 'b list
```

- elle applique une opération retournant une option, sur chaque élément de la liste
- elle ne préserve que les éléments contenus dans des Some

Exemple :

```
# List.filter_map (fun x -> if x < 0 then Some (-x) else None) [-1; 0; 2];;  
- : int list = [1]
```

Autres itérations sur des listes

On souhaite savoir si tous les entiers d'une liste sont pairs

```
# all_even [0; 2; 4];;
```

```
- : bool = true
```

```
# all_even [0; 2; 3];;
```

```
- : bool = false
```

Autres itérations sur des listes

```
let rec all_even (l : int list) : bool =  
  match l with  
  | [] -> true  
  | head :: tail when head mod 2 = 0 -> all_even tail  
  | _ -> false
```

Autres itérations sur des listes

On souhaite savoir s'il existe un élément qui est pair

```
let rec exists_even (l : int list) : bool =  
  match l with  
  | [] -> false  
  | head :: tail when head mod 2 = 0 -> true  
  | _ -> exists_even l
```

Autres itérations sur des listes

On peut généraliser avec une fonction d'ordre supérieur

```
let rec all (p : 'a -> bool) (l : 'a list) : bool =  
  match l with  
  | [] -> true  
  | head :: tail when p head -> all p tail  
  | _ -> false
```

```
let rec exists (p : 'a -> bool) (l : 'a list) : bool =  
  match l with  
  | [] -> false  
  | head :: tail when p head -> true  
  | _ -> exists l p
```

Autres itérations sur des listes

On souhaite calculer la somme des éléments d'une liste

```
let rec sum (l : int list) : int =  
  match l with  
  | [] -> 0  
  | head :: tail -> head + (sum tail)
```

Autres itérations sur des listes

On souhaite calculer le produit des éléments d'une liste

```
let rec product (l : int list) : int =  
  match l with  
  | [] -> 1  
  | head :: tail -> head * (product tail)
```

Généralisation de somme et produit

```
let rec g (f : int -> int -> int) (l : int list) (init : int) : int =  
  match l with  
  | [] -> init  
  | head :: tail -> f head (g f tail init)
```

- on a un élément “initial” `init`
- et une fonction `f : int -> int -> int`

```
let sum l = g (+) l 0
```

```
let product l = g ( * ) l 1
```

Généralisation de somme et produit

On peut généraliser au delà du type `int` :

```
let rec g (f : 'a -> 'a -> 'a) (l : 'a list) (init : 'a) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> f head (g f tail init)
```

Généralisation de somme et produit

On peut généraliser encore plus :

- si `init` : 'a
- `g` doit renvoyer 'a (cas [])
- `f` peut prendre un 'b comme premier argument
- `l` doit être alors une 'b list

```
let rec g (f : 'b -> 'a -> 'a) (l : 'b list) (init : 'a) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> f head (g f tail init)
```

Généralisation de somme et produit : repli

On appelle cette fonction `fold_right` : c'est un **repli à droite**

```
let rec fold_right (f : 'b -> 'a -> 'a) (l : 'b list) (init : 'a) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> f head (fold_right f tail init)
```

Repli à droite : `fold_right`

Documentation de `List.fold_right` :

```
val fold_right : ('a -> 'acc -> 'acc) -> 'a list -> 'acc -> 'acc
```

```
fold_right f [a1; ...; an] init is f a1 (f a2 (... (f an init) ...)).
```

Not tail-recursive.

Utilisation de replis

```
let length (l : 'a list) : int =  
    List.fold_right (fun _ acc -> acc + 1) l 0
```

À chaque élément vu, on augmente la taille (*l'accumulateur*) de 1

Repli à droite

```
fold_right f [a; b; c] init = f a (f b (f c init))
```

Pour `length` : `f` ignore son premier argument et incrémente son second de 1 :

```
f a (f b (f c init)) = 1 + (1 + (1 + 0))
```

Les éléments de la liste sont combinés par l'opération `f` selon un **sens d'associativité** fixé. Il s'agit d'une associativité de la **droite** vers la gauche.

Associativité

Rappel : une fonction est associative ssi

$$f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z$$

Pour une fonction associative (e.g., +), le sens d'associativité n'a pas d'importance.

Repli à droite et associativité

Question : quel est le résultat de l'évaluation suivante ?

```
# List.fold_right (-) [1; 2; 3] 0;;
```

Repli à droite et associativité

```
# List.fold_right (-) [1; 2; 3] 0;;  
- : int = 2
```

On a calculé :

$$1 - (2 - (3 - 0)) = 1 - (-1) = 2$$

Repli à droite et associativité

Si on veut le repli avec le sens d'associativité de la **gauche** vers la droite, on peut faire un repli à gauche :

```
List.fold_left f init [a; b; c] = f (f (f init a) b) c
```

Définition de fold_left

```
let rec fold_left (f : 'a -> 'b -> 'a) (init : 'a) (l : 'b list) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> fold_left f (f init head) tail
```

fold_left et fold_right

L'ordre des arguments reflète le sens de l'associativité

```
val fold_left (f : 'acc -> 'a -> 'acc) (init : 'acc) (l : 'a list) : 'acc
val fold_right (f : 'a -> 'acc -> 'acc) (l : 'a list) (init : 'acc) : 'acc
```

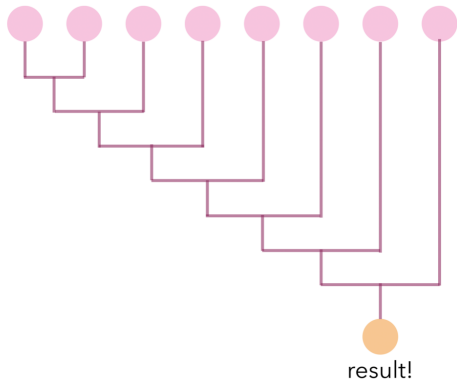
Utilisation de fold_left

```
# List.fold_left (-) 0 [1; 2; 3];;  
- : int = -6
```

Replis

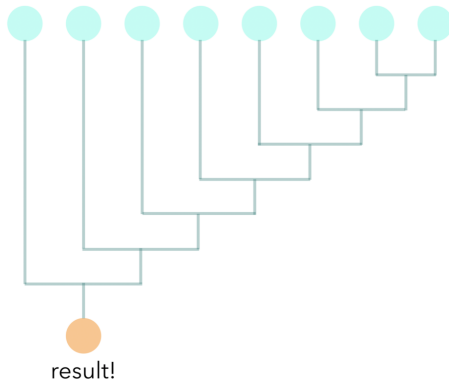
foldl

START → FINISH



foldr

FINISH ← START



Source

Exercice

Définir une fonction qui calcule le minimum des éléments d'une liste non vide, de trois façons :

- dans un langage impératif (Java, C, ...)
- en OCaml, sans repli
- en OCaml, avec un repli

Replis

Exercice

Définir une fonction qui calcule le nombre de fois qu'un certain élément apparaît dans une liste :

- sans repli
- avec un repli

Replis dans d'autres langages

Python

```
import functools
print(functools.reduce(lambda a, b: a+b, [0, 1, 2, 3, 4, 5]))
```

Java

```
Integer sum = list.stream().reduce(0, (a, b) -> a + b)
```

Attention : dans beaucoup de langages, reduce est un repli qui attend une liste non-vide, et utilise le premier élément comme accumulateur initial

Replis et efficacité

```
# let big_list = List.init 100000000 (fun i -> i);;  
# List.fold_right (+) big_list 0;;  
Stack overflow during evaluation (looping recursion?).  
# List.fold_left (+) 0 big_list;;  
- : int = 5000000050000000
```

Pourquoi ?

Replis et efficacité

```
let rec fold_right (f : 'b -> 'a -> 'a) (l : 'b list) (init : 'a) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> f head (fold_right f tail init)
```

```
let rec fold_left (f : 'a -> 'b -> 'a) (init : 'a) (l : 'b list) : 'a =  
  match l with  
  | [] -> init  
  | head :: tail -> fold_left f (f init head) tail
```

fold_right n'est pas récursive terminale ! (*tail-recursive*)

Replis : exercice

Exercice

Définir les fonctions suivantes avec des replis :

```
val exists : ('a -> bool) -> 'a list -> bool
```

```
val all : ('a -> bool) -> 'a list -> bool
```

Replis et universalité

L'opération de repli est **universelle** : on peut définir les autres opérations sur les liste avec

```
map f [a; b; c; ...] = [f a; f b; f c; ...]
                      = (f a) :: (f b) :: (f c) :: ... :: []
                      = (f a) :: ((f b) :: ((f c) :: (... :: [])))
```

```
# let l = [1; 2; 3; 4; 5];;
# List.fold_right (fun x xs -> (- x) :: xs) l [];;
- : int list = [-1; -2; -3; -4; -5]
```

Donc :

```
let map (f : 'a -> 'b) (l : 'a list) : 'b list =
  List.fold_right (fun x xs -> (f x) :: xs) l []
```

Replis et universalité

On peut faire de même pour filter :

```
# let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# List.fold_right
  (fun x xs -> if x < 5 then x :: xs else xs)
  1
  [];;
- : int list = [1; 2; 3; 4]
```

Donc :

```
let filter (p : 'a -> bool) (l : 'a list) : 'a list =
  List.fold_right (fun x xs -> if p x then x :: xs else xs) l []
```

Exercice de repli

Exercice

Définir les fonctions suivantes avec des replis :

```
val and_ : bool list -> bool
```

```
val reverse : 'a list -> 'a list
```

and_ renvoie vrai si tous les éléments de la liste sont vrais. reverse inverse l'ordre des éléments dans une liste.

map, filter et fold sur d'autres structures de données

On peut appliquer ces concepts à d'autres structures (au cas par cas).

Par exemple, sur un arbre, on peut définir :

```
val tree_map : ('a -> 'b) -> 'a tree -> 'b tree
```

```
val tree_fold : ('a -> 'b -> 'b) -> 'b -> 'a tree -> 'b
```

tree_map

```
type 'a tree =  
| Leaf  
| Node of 'a tree * 'a * 'a tree  
  
let rec tree_map (f : 'a -> 'b) (tree : 'a tree) : 'b tree =  
  match tree with  
  | Leaf -> Leaf  
  | Node (l, x, r) -> Node (tree_map f l, f x, tree_map f r)
```

tree_fold

```
type 'a tree =  
  | Leaf  
  | Node of 'a tree * 'a * 'a tree  
  
let rec tree_fold (f : 'b -> 'a -> 'b -> 'b) (init : 'b) (tree : 'a tree) : 'b  
  match tree with  
    | Leaf -> init  
    | Node (l, x, r) -> f (tree_fold f init l) x (tree_fold f init r)
```

Utilisation de tree_fold

```
let size (tree : 'a tree) : int =  
  tree_fold (fun l _ r -> l + r + 1) 0 t  
let depth (tree : 'a tree) : int =  
  tree_fold (fun _ l r -> 1 + max l r) 0 t  
let preorder (tree : 'a tree) : 'a list =  
  tree_fold (fun x l r -> [x] @ l @ r) [] t
```

Concepts clés

- **Fonction d'ordre supérieur** : fonction qui renvoie une fonction et/ou prend en paramètre une autre fonction
- **Fermeture** : une fonction et son environnement de définition
- **map** : `('a -> 'b) -> 'a list -> 'b list` : fonction qui permet de parcourir les éléments d'une liste et les remplacer un à un
- **filter** : `('a -> bool) -> 'a list -> 'a list` : fonction qui permet de sélectionner seulement certains éléments d'une liste
- **Replis** : généralisation universelles des itérations sur une liste, soit de gauche à droite, soit de droite à gauche
 - `fold_right f [a; b; c] init = f a (f b (f c init))`
 - `fold_left f init [a; b; c] = f (f (f init a) b) c`
- Les opérations telles que `fold` et `map` peuvent se généraliser au delà des listes

Exemple : Interpréteur BF