Programmation Fonctionnelle – Itérateurs

Adrien Durier

26 septembre 2023

Itérateurs sur les listes

Schémas de définitions récursives

Les fonctions suivantes ont toutes la même structure :

- la fonction `zeros`
- la fonction `recherche`
- la fonction `longueur`
- la fonction `append`
- la fonction `existe`
- la fonction `map`
- etc.

Laquelle ?

Itérateurs : Schéma récursif commun

Ces fonctions ont toutes le schéma récursif suivant (on note `l` la liste en entrée et `f` la fonction définie récursivement) :

1. Cas de base : si `l` est la liste vide, la valeur retournée ne dépend pas de `l`.
2. Cas de récursif : sinon, `l` est de la forme `x::s` et la valeur retournée est calculée à partir d'une opération `op` sur `x` et `f s`.

Itérateurs : Itération d'ordre supérieur

On peut capturer ce schéma à l'aide d'une fonction d'ordre supérieur prenant en argument une fonction `op` (à deux arguments, `x` et `(f s)`), une liste `l` et un élément de départ `acc`

1. Cas de base : si `l` est la liste vide, on renvoie `acc`.
2. Cas de récursif : sinon, `l` est de la forme `x::s` et on renvoie `op x (f s)`.

Itérateurs : Exemple

On montre comment abstraire le schéma d'une définition récursive à partir de la fonction somme suivante :

```
let rec somme l =
  match l with
   | [] -> 0
   | x::s -> x + (somme s)
```

Itérateurs : Exemple

Étape 1 : extraire l'opération du cas récursif

```
let rec somme l =
  match l with
   [] -> 0
   x::s -> (fun a b -> a + b) x (somme s)
```

Itérateurs : Exemple

Étape 2 : abstraire l'opération récursive

```
let rec fold op l =
  match l with
  | [] -> 0
  | x::s -> op x (fold op s)
let somme l = fold (fun a b -> a + b) l
```

Notez que `(fold op)` correspond à `somme`.

Itérateurs : Exemple

Étape 3 : abstraire l'accumulateur

```
let rec fold_right op l acc =
  match l with
  | [] -> acc
  | x::s -> op x (fold_right op s acc)
let somme l = fold_right (fun a b -> a + b) l 0
let somme l = fold_right (+) l 0
```

Notez que `(fold_right op l acc)` correspond à `(somme l acc)`, la fonction qui additionne les valeurs de `l` à `acc`.

Itérateurs : Exemple

```
somme [1;2;3]
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (fold (+) [3] 0))
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (fold (+) [3] 0))
 ⇒ (+) 1 ((+) 2 ((+) 3 (fold (+) [] 0)))
```

Itérateurs : Exemple

```
somme [1;2;3]
```

$\Rightarrow$ `fold (+) [1;2;3] 0`

$\Rightarrow$ `(+) 1 (fold (+) [2;3] 0)`

$\Rightarrow$ `(+) 1 ((+) 2 (fold (+) [3] 0))`

$\Rightarrow$ `(+) 1 ((+) 2 ((+) 3 (fold (+) [] 0)))`

$\Rightarrow$ `(+) 1 ((+) 2 ((+) 3 0))`

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (fold (+) [3] 0))
 ⇒ (+) 1 ((+) 2 ((+) 3 (fold (+) [] 0)))
 ⇒ (+) 1 ((+) 2 ((+) 3 0))
 ⇒ (+) 1 ((+) 2 3)
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (fold (+) [3] 0))
 ⇒ (+) 1 ((+) 2 ((+) 3 (fold (+) [] 0)))
 ⇒ (+) 1 ((+) 2 ((+) 3 0))
 ⇒ (+) 1 ((+) 2 3)
 ⇒ (+) 1 5
```

Itérateurs : Exemple

```
somme [1;2;3]
 ⇒ fold (+) [1;2;3] 0
 ⇒ (+) 1 (fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (fold (+) [3] 0))
 ⇒ (+) 1 ((+) 2 ((+) 3 (fold (+) [] 0)))
 ⇒ (+) 1 ((+) 2 ((+) 3 0))
 ⇒ (+) 1 ((+) 2 3)
 ⇒ (+) 1 5
 ⇒ 6
```

Itérateurs : `fold_right`

La fonction `fold_right` n'a plus rien de spécifique à la somme !
C'est un schéma récursif général prédéfini sour le nom de `List.fold_right` dans `ocaml` :

```
let rec fold_right op l acc =
 match l with
   | [] -> acc
   | x::l -> op x (fold_right op l acc)
```

Itérateurs : `fold_right`

La fonction `fold_right` n'a plus rien de spécifique à la somme !
C'est un schéma récursif général prédéfini sour le nom de `List.fold_right` dans `ocaml` :

```
let rec fold_right op l acc =
 match l with
   | [] -> acc
   | x::l -> op x (fold_right op l acc)
```

Type : `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

$$\texttt{fold\_right op [] acc} = \texttt{acc}$$

$$\texttt{fold\_right op } [e_1; e_2; \ldots; e_n] \texttt{ acc} = \texttt{op } e_1 \ (\texttt{op } e_2 \ (\cdots (\texttt{op } e_n \ \textit{acc}) \cdots))$$

Attention : cette fonction n'est pas récursive terminale !

Itérateurs : Exemples

On peut écrire append avec `List.fold_right` :

```
let rec append l1 acc =
  match l1 with
   | [] -> acc
   | x::s -> x::(append s acc)
append : 'a list -> 'a list -> 'a list
```

Itérateurs : Exemples

On peut écrire append avec `List.fold_right` :

```
  let rec append l1 acc =
    match l1 with
     | [] -> acc
     | x::s -> x::(append s acc)
  append : 'a list -> 'a list -> 'a list
```

Devient :

```
let append l1 acc =
  fold_right (fun x fs -> x::fs) l1 acc
val append : 'a list -> 'a list -> 'a list = <fun>
```

Itérateurs : Exemples

On peut écrire append avec `List.fold_right` :

```
  let rec append l1 acc =
    match l1 with
     | [] -> acc
     | x::s -> x::(append s acc)
  append : 'a list -> 'a list -> 'a list
```

Devient :

```
let append l1 acc =
  fold_right (fun x fs -> x::fs) l1 acc
val append : 'a list -> 'a list -> 'a list = <fun>
```

```
# append [1;2;3] [4;5;6];;
```

Itérateurs : Exemples

On peut écrire append avec `List.fold_right` :

```
  let rec append l1 acc =
    match l1 with
     | [] -> acc
     | x::s -> x::(append s acc)
  append : 'a list -> 'a list -> 'a list
```

Devient :

```
let append l1 acc =
  fold_right (fun x fs -> x::fs) l1 acc
val append : 'a list -> 'a list -> 'a list = <fun>
```

```
# append [1;2;3] [4;5;6];;
```

```
- : int list = [1; 2; 3; 4; 5; 6]
```

Itérateurs : `fold_left`

Pour les opérations `op` pour que l'on peut réécrire pour qu'elles prennent `f` de `(op acc x)` et de `s`, on préfèrerait utiliser le parcours suivant :

$$\texttt{fold\_left op acc []} = \texttt{acc}$$

$$\texttt{fold\_left op acc } [e_1; e_2; \ldots; e_n] = \texttt{op } (\cdots (\texttt{op } (\texttt{op } \textit{acc } e_1)\ e_2) \cdots)\ e_n)$$

. . .car c'est ce que fait la fonction prédéfinie `List.fold_left` de manière récursive terminale :

```
let rec fold_left op acc l =
  match l with
   | [] -> acc
   | x::s -> fold_left op (op acc x) s
```

Itérateurs : `fold_left`

Pour les opérations `op` pour que l'on peut réécrire pour qu'elles prennent `f` de `(op acc x)` et de `s`, on préfèrerait utiliser le parcours suivant :

$$\texttt{fold\_left op acc []} = \texttt{acc}$$

$$\texttt{fold\_left op acc } [e_1; e_2; \ldots; e_n] = \texttt{op } (\cdots (\texttt{op } (\texttt{op } \textit{acc } e_1)\ e_2) \cdots)\ e_n)$$

. . .car c'est ce que fait la fonction prédéfinie `List.fold_left` de manière récursive terminale :

```
let rec fold_left op acc l =
  match l with
    | [] -> acc
    | x::s -> fold_left op (op acc x) s
```

Son type est `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

Itérateurs : Exemples

On peut écrire la `somme` avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

```
    somme [1;2;3]
=   fold_left (+) 0 [1;2;3]
```

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 1 2) [3]}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 1 2) [3]} \\
= \quad & \texttt{fold\_left (+) 3 [3]}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 1 2) [3]} \\
= \quad & \texttt{fold\_left (+) 3 [3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 3 3) []}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 1 2) [3]} \\
= \quad & \texttt{fold\_left (+) 3 [3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 3 3) []} \\
= \quad & \texttt{fold\_left (+) 6 []}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la somme avec `List.fold_left` :

```
# let somme l = fold_left (+) 0 l
- val somme : int list -> int
```

$$
\begin{aligned}
& \texttt{somme [1;2;3]} \\
= \quad & \texttt{fold\_left (+) 0 [1;2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 0 1) [2;3]} \\
= \quad & \texttt{fold\_left (+) 1 [2;3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 1 2) [3]} \\
= \quad & \texttt{fold\_left (+) 3 [3]} \\
\Rightarrow \quad & \texttt{fold\_left (+) ((+) 3 3) []} \\
= \quad & \texttt{fold\_left (+) 6 []} \\
\Rightarrow \quad & \texttt{6}
\end{aligned}
$$

Itérateurs : Exemples

On peut écrire la `longueur` avec `List.fold_left` :

```
# let rec longueur acc l =
   match l with
    |  [] -> acc
    | x::s -> (longueur (acc+1) s)
- val longueur : int -> 'a list -> int
```

Itérateurs : Exemples

On peut écrire la `longueur` avec `List.fold_left` :

```
# let rec longueur acc l =
   match l with
    |  [] -> acc
    | x::s -> (longueur (acc+1) s)
- val longueur : int -> 'a list -> int
```

Devient :

```
# let longueur l = fold_left (fun acc x -> acc + 1) 0 l
- longueur : 'a list -> int
```

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

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

```
    longueur [3;2;1;4]
=   fold_left plus1 0 [3;2;1;4]
```

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

```
    longueur [3;2;1;4]
=   fold_left plus1 0 [3;2;1;4]
⇒   fold_left plus1 (plus1 0 3) [2;1;4]
```

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

```
    longueur [3;2;1;4]
=   fold_left plus1 0 [3;2;1;4]
⇒   fold_left plus1 (plus1 0 3) [2;1;4]
=   fold_left plus1 1 [2;1;4]
```

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

```
    longueur [3;2;1;4]
=   fold_left plus1 0 [3;2;1;4]
⇒   fold_left plus1 (plus1 0 3) [2;1;4]
=   fold_left plus1 1 [2;1;4]
⇒   fold_left plus1 (plus1 1 2) [1;4]
=   fold_left plus1 2 [1;4]
⇒   fold_left plus1 (plus1 2 1) [1;4]
=   fold_left plus1 3 [4]
⇒   fold_left plus1 (plus1 3 4) []
=   fold_left plus1 4 []
```

Itérateurs : Exemples

Évaluation de `longueur [3;2;1;4]`

On note `plus1` la fonction `(fun acc x -> acc + 1)`

```
    longueur [3;2;1;4]
=   fold_left plus1 0 [3;2;1;4]
⇒   fold_left plus1 (plus1 0 3) [2;1;4]
=   fold_left plus1 1 [2;1;4]
⇒   fold_left plus1 (plus1 1 2) [1;4]
=   fold_left plus1 2 [1;4]
⇒   fold_left plus1 (plus1 2 1) [1;4]
=   fold_left plus1 3 [4]
⇒   fold_left plus1 (plus1 3 4) []
=   fold_left plus1 4 []
=   4
```

Itérateurs : Exemples

```
# let zeros = fold_left (fun acc x -> acc && x=0) true
- val zeros : int list -> bool = <fun>
```

Itérateurs : Exemples

```
# let zeros = fold_left (fun acc x -> acc && x=0) true
- val zeros : int list -> bool = <fun>
```

```
# let recherche n =
    fold_left (fun acc x -> acc || x=n) false
- val recherche : 'a -> 'a list -> bool = <fun>
```

Itérateurs : Exemples

```
# let zeros = fold_left (fun acc x -> acc && x=0) true
- val zeros : int list -> bool = <fun>
```

```
# let recherche n =
    fold_left (fun acc x -> acc || x=n) false
- val recherche : 'a -> 'a list -> bool = <fun>
```

```
# let existe p = fold_left (fun acc x -> acc || p x) false
- val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Itérateurs : Exemples

On souhaite écrire une fonction `sous_listes` pour calculer la liste des sous-listes d'une liste `l` On commence par écrire une fonction

`cons` qui ajoute un élément à toutes les listes d'une liste de listes :

```
# let rec cons_elt x l =
    match l with
     | [] -> []
     | r::s -> (x::r)::(cons_elt x s)
- cons_elt : 'a -> 'a list list -> 'a list list
```

Itérateurs : Exemples

La fonction `sous_liste` s'écrit alors naturellement de la manière suivante :

```
# let rec sous_listes l =
    match l with
     | [] -> [[]]
     | x::s -> let p = sous_listes s in (cons_elt x p)@p
- sous_listes : 'a list -> 'a list list
```

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

Itérateurs : Exemples

Le fonction `sous_liste` peut aussi s'écrire :

```
let sous_listes =
  fold_left (fun p x -> (map (fun l->l@[x]) p)@p) [[]]
```

Exercice

Fonction `insert`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

1. `insert: 'a -> 'a list -> 'a list list`
   Dans `insert x [v1; v2; ...; vk]`, `x` est inséré à chaque position. Par exemple, `insere 1 [2; 3; 4]` :

$$[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]]$$

- **Indication :** Utilisez `List.map`.

```
let rec insert x l =
```

Fonction `insert`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

1. `insert: 'a -> 'a list -> 'a list list`
   Dans `insert x [v1; v2; ...; vk]`, `x` est inséré à chaque position. Par exemple, `insere 1 [2; 3; 4]` :

$$[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]]$$

- **Indication :** Utilisez `List.map`.

```
let rec insert x l =
  match l with
  | [] -> [[x]]
  | y :: s ->
```

Fonction `insert`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

1. `insert: 'a -> 'a list -> 'a list list`
   Dans `insert x [v1; v2; ...; vk]`, `x` est inséré à chaque position. Par exemple, `insere 1 [2; 3; 4]` :

$$[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]]$$

- **Indication :** Utilisez `List.map`.

```
let rec insert x l =
  match l with
  | [] -> [[x]]
  | y :: s ->
      let r = insert x s in
      (x :: l) :: (List.map (fun v -> y::v) r)
```

Exercice 2 : Fonction `permutations`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

2. `permutations : 'a list -> 'a list list` renvoit la liste des permutations de son argument. Par exemple, `permutations [1;2;3]` vaut :

$$[[1;3;2];[3;1;2];[3;2;1];[1;2;3];[2;1;3];[2;3;1]]$$

- **Indication :** Utilisez `List.fold_left`.

```
let rec permutations l =
```

Exercice 2 : Fonction `permutations`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

2. `permutations : 'a list -> 'a list list` renvoit la liste des permutations de son argument. Par exemple, `permutations [1;2;3]` vaut :

$$[[1;3;2];[3;1;2];[3;2;1];[1;2;3];[2;1;3];[2;3;1]]$$

- **Indication :** Utilisez `List.fold_left`.

```
let rec permutations l =
  match l with
  | [] -> [ [] ]
  | x :: s ->
```

Exercice 2 : Fonction `permutations`

But de l'exercice : calculer l'ensemble des permutations d'une liste `[1;2;...;n]`.

2. `permutations : 'a list -> 'a list list` renvoit la liste des permutations de son argument. Par exemple, `permutations [1;2;3]` vaut :

$$[[1;3;2];[3;1;2];[3;2;1];[1;2;3];[2;1;3];[2;3;1]]$$

- **Indication :** Utilisez `List.fold_left`.

```
let rec permutations l =
  match l with
  | [] -> [ [] ]
  | x :: s ->
      let r = permutations s in
      List.fold_left (fun acc p -> (insert x p) @ acc) [] r
```