Programmation fonctionnelle avancée

Notes de cours

Cours 2

16 Septembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Les fonctions anonymes

Les fonctions anonymes

```
# fun x -> x * x;;
- : int -> int = <fun>
# (fun x -> x * x) 4;;
- : int = 16
# fun x y -> x * y;;
- : int -> int -> int = <fun>
```

- le mot-clé `fun` permet de créer des valeurs fonctionnelles
- les fonctions anonymes s'appliquent comme les fonctions nommées
- la déclation `let f x y = x * y` est donc équivalente à `let f = fun x y -> x * y`

Ordre Supérieur

Les fonctions sont des valeurs à part entière

Les fonctions sont des types de données comme les autres

Les fonctions sont des valeurs à part entière

Les fonctions sont des types de données comme les autres

Une fonction peut être :

- stockée dans une structure de donnée (n-uplets, enregistrements, listes etc.)
- passée en argument à une autre fonction
- retournée comme résultat d'une fonction

Les fonctions sont des valeurs à part entière

Les fonctions sont des types de données comme les autres

Une fonction peut être :

- stockée dans une structure de donnée (n-uplets, enregistrements, listes etc.)
- passée en argument à une autre fonction
- retournée comme résultat d'une fonction

Les fonctions prenant des fonctions en arguments ou rendant des fonctions en résultat sont dites **d'ordre supérieur**

Structures de données contenant des fonctions

Un n-uplet avec des composantes fonctionnelles :

```
# ( (fun x-> x + 1), 4 , (fun x -> x :: ['a']));;
- : (int -> int) * int * (char -> char list) = (<fun>,4,<fun>)
```

Structures de données contenant des fonctions

Un n-uplet avec des composantes fonctionnelles :

```
# ( (fun x-> x + 1), 4 , (fun x -> x :: ['a']));;
- : (int -> int) * int * (char -> char list) = (<fun>,4,<fun>)
```

Un enregistrement avec une étiquette fonctionnelle :

```
# type t = { f : int -> int ; x : int };;
type t = { f : int -> int; x : int; }
```

Structures de données contenant des fonctions

Un n-uplet avec des composantes fonctionnelles :

```
# ( (fun x-> x + 1), 4 , (fun x -> x :: ['a']));;
- : (int -> int) * int * (char -> char list) = (<fun>,4,<fun>)
```

Un enregistrement avec une étiquette fonctionnelle :

```
# type t = { f : int -> int ; x : int };;
type t = { f : int -> int; x : int; }

# { f = (fun x -> x + 1) ; x=10 };;
- : t = {f = <fun>; x = 10}
```

Structures de données contenant des fonctions

Un n-uplet avec des composantes fonctionnelles :

```
# ( (fun x-> x + 1), 4 , (fun x -> x :: ['a']));;
- : (int -> int) * int * (char -> char list) = (<fun>,4,<fun>)
```

Un enregistrement avec une étiquette fonctionnelle :

```
# type t = { f : int -> int ; x : int };;
type t = { f : int -> int; x : int; }

# { f = (fun x -> x + 1) ; x=10 };;
- : t = {f = <fun>; x = 10}
```

Une liste contenant des fonctions :

```
# [(fun x-> x+1); (fun x-> x * 2); (fun x-> 4)];;
- : (int -> int) list = [<fun>; <fun>; <fun>]
```

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ se traduisent immédiatement si l'on peut utiliser des arguments fonctionnels

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ se traduisent immédiatement si l'on peut utiliser des arguments fonctionnels

```
# let rec somme (f, n) =
   if n<=0 then 0
   else (f n) + somme (f, n - 1);;
val somme : (int -> int) * int -> int = <fun>
```

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ se traduisent immédiatement si l'on peut utiliser des arguments fonctionnels

```
# let rec somme (f, n) =
    if n<=0 then 0
    else (f n) + somme (f, n - 1);;
val somme : (int -> int) * int -> int = <fun>
```

```
# somme ((fun x-> x * x), 10);;
- : int = 385
```

Exemple : la méthode dichotomique (1/2)

Si f est une fonction continue et monotone, on peut trouver un zéro de f sur un intervalle $[a, b]$ par la méthode dichotomique quand $f(a)$ et $f(b)$ sont de signes opposés :

- si ϵ est la précision souhaitée et que $|b - a| < \epsilon$ alors on renvoie a
- sinon, couper l'intervalle $[a, b]$ en deux et recommencer sur l'intervalle contenant 0

La méthode dichotomique (2/2)

```
# let rec dichotomie (f,a,b,epsilon) =
   if abs_float(b -. a) < epsilon then a
   else
     let c = (a+.b) /. 2.0 in
     let na, nb = if (f a)*.(f c)>0.0 then (c,b) else (a,c) in
     dichotomie (f, na, nb, epsilon)

val dichotomie :
  (float -> float) * float * float * float -> float = <fun>
```

La méthode dichotomique (2/2)

```
# let rec dichotomie (f,a,b,epsilon) =
   if abs_float(b -. a) < epsilon then a
   else
     let c = (a+.b) /. 2.0 in
     let na, nb = if (f a)*.(f c)>0.0 then (c,b) else (a,c) in
     dichotomie (f, na, nb, epsilon)

val dichotomie :
  (float -> float) * float * float * float -> float = <fun>
```

Par exemple, on peut utiliser cette fonction pour trouver un encadrement de π en le calculant comme zéro de la fonction $\cos(x/2)$

```
# dichotomie ((fun x -> cos (x/.2.0)), 3.1, 3.2, 1e-10);;
- : float = 3.14159265356138384
```

Fonctions en résultat

Les fonctions à plusieurs arguments sont en fait des fonctions d'ordre supérieur qui rendent des fonctions en résultat

```
# let plus x y = x+y;;
val plus : int -> int -> int
```

Fonctions en résultat

Les fonctions à plusieurs arguments sont en fait des fonctions d'ordre supérieur qui rendent des fonctions en résultat

```
# let plus x y = x+y;;
val plus : int -> int -> int
```

Il faut lire le type de cette fonction de la manière suivante

```
int -> (int -> int)
```

Fonctions en résultat

Les fonctions à plusieurs arguments sont en fait des fonctions d'ordre supérieur qui rendent des fonctions en résultat

```
# let plus x y = x+y;;
val plus : int -> int -> int
```

Il faut lire le type de cette fonction de la manière suivante

```
int -> (int -> int)
```

De manière équivalente, on peut écrire la fonction `plus` de la façon suivante afin de souligner son résultat fonctionnel

```
# let plus x = (fun y -> x+y);;
val plus : int -> int -> int
```

Application partielle

Les fonctions d'ordre supérieur rendant des fonctions en résultats peuvent être appliquées partiellement

```
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>
```

Application partielle

Les fonctions d'ordre supérieur rendant des fonctions en résultats peuvent être appliquées partiellement

```
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>

# plus2 10;;
- : int = 12
```

Fonctions en arguments et en résultat (1/2)

On peut calculer de façon approximative la dérivée f' d'une fonction f avec un petit intervalle `dx` de la manière suivante :

```
# let derive (f,dx) = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : (float -> float) * float -> float -> float
```

Fonctions en arguments et en résultat (1/2)

On peut calculer de façon approximative la dérivée f' d'une fonction f avec un petit intervalle `dx` de la manière suivante :

```
# let derive (f,dx) = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : (float -> float) * float -> float -> float
```

```
# derive ( (fun x->x*.x),1e-10) 1.;;
- : float = 2.000000165480742
```

Fonctions en arguments et en résultat (2/2)

On peut réécrire la fonction `derive` de la manière suivante

```
# let derive dx f = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : float -> (float -> float) -> float -> float
```

On fixe le paramètre `dx` par application partielle

Fonctions en arguments et en résultat (2/2)

On peut réécrire la fonction `derive` de la manière suivante

```
# let derive dx f = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : float -> (float -> float) -> float -> float
```

On fixe le paramètre `dx` par application partielle

```
# let derivation = derive 1e-10;;
val derivation : (float -> float) -> float -> float
```

Fonctions en arguments et en résultat (2/2)

On peut réécrire la fonction `derive` de la manière suivante

```
# let derive dx f = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : float -> (float -> float) -> float -> float
```

On fixe le paramètre `dx` par application partielle

```
# let derivation = derive 1e-10;;
val derivation : (float -> float) -> float -> float
```

On peut alors définir par exemple la dérivée de la fonction sinus

```
# let sin' = derivation sin;;
val sin' : float -> float
```

Fonctions en arguments et en résultat (2/2)

On peut réécrire la fonction `derive` de la manière suivante

```
# let derive dx f = fun x -> (f(x +. dx) -. f(x))/. dx;;
val derive : float -> (float -> float) -> float -> float
```

On fixe le paramètre `dx` par application partielle

```
# let derivation = derive 1e-10;;
val derivation : (float -> float) -> float -> float
```

On peut alors définir par exemple la dérivée de la fonction sinus

```
# let sin' = derivation sin;;
val sin' : float -> float

# sin' 1.;;
- : float = 0.540302247387103307
# cos 1.;;
- : float = 0.540302305868139765
```

Ordre supérieur et polymorphisme

Le mélange **ordre supérieur+polymorphisme** permet d'écrire du code plus général et donc plus réutilisable

```
# let double x = 2 * x;;
# let carre x = x * x;;
```

Ordre supérieur et polymorphisme

Le mélange **ordre supérieur+polymorphisme** permet d'écrire du code plus général et donc plus réutilisable

```
# let double x = 2 * x;;
# let carre x = x * x;;
```

On utilise ces fonctions pour définir une fonction qui quadruple un entier x et une autre qui calcule x^4

```
# let quadruple  x = double (double x);;
# let puissance4 x = carre (carre x);;
```

Ordre supérieur et polymorphisme

Le mélange **ordre supérieur+polymorphisme** permet d'écrire du code plus général et donc plus réutilisable

```
# let double x = 2 * x;;
# let carre x = x * x;;
```

On utilise ces fonctions pour définir une fonction qui quadruple un entier x et une autre qui calcule x^4

```
# let quadruple  x = double (double x);;
# let puissance4 x = carre (carre x);;
```

Pour simplifier, on définit :

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

```
# let quadruple x = applique_deux_fois double x;;
# let puissance4 x = applique_deux_fois carre x;;
```

Exemples : fonctions génériques sur les listes (1/3)

La fonction permettant de tester l'existence d'un élément dans une liste vérifiant une propriété quelconque `p`

```
#let rec existe p l =
   match l with
       [] -> false
   | x::s -> p x || (existe p s);;
val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Exemples : fonctions génériques sur les listes (1/3)

La fonction permettant de tester l'existence d'un élément dans une liste vérifiant une propriété quelconque `p`

```
#let rec existe p l =
   match l with
      [] -> false
   | x::s -> p x || (existe p s);;
val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Le test d'appartenance à une liste s'écrit facilement en utilisant `existe` de la manière suivante

```
# let appartient x = existe (fun y->x=y);;
val appartient : 'a -> 'a list -> bool = <fun>
```

Exemples : fonctions génériques sur les listes (1/3)

La fonction permettant de tester l'existence d'un élément dans une liste vérifiant une propriété quelconque `p`

```
#let rec existe p l =
   match l with
       [] -> false
   | x::s -> p x || (existe p s);;
val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Le test d'appartenance à une liste s'écrit facilement en utilisant `existe` de la manière suivante

```
# let appartient x = existe (fun y->x=y);;
val appartient : 'a -> 'a list -> bool = <fun>

# appartient 'a' ['o';'c';'a';'m';'l'];;
- : bool = true
```

Exemples : fonctions génériques sur les listes (2/3)

La fonction `filtre` filtre tous les éléments d'une liste vérifiant une certaine propriété `p`

```
#let rec filtre p l =
   match l with
       [] -> []
     | x::s -> if p x then x::(filtre p s) else filtre p s;;
val filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Exemples : fonctions génériques sur les listes (2/3)

La fonction `filtre` filtre tous les éléments d'une liste vérifiant une certaine propriété `p`

```
#let rec filtre p l =
   match l with
       [] -> []
     | x::s -> if p x then x::(filtre p s) else filtre p s;;
val filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
# filtre (fun x->x mod 2=0) [1;2;3;4];;
- : int list = [2; 4]
```

Exemples : fonctions génériques sur les listes (3/3)

La fonction `map` transforme une liste `[e1;..;en]` en une liste `[f e1;..; f en]` pour une fonction `f` quelconque

```
#let rec map f l =
    match l with
    [] -> []
   | x::s -> (f x)::(map f s);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La fonction `map` transforme une liste `[e1;..;en]` en une liste `[f e1;..; f en]` pour une fonction `f` quelconque

```
#let rec map f l =
    match l with
    [] -> []
   | x::s -> (f x)::(map f s);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map float_of_int [1;2;3;4];;
- : float list = [1.0; 2.0; 3.0; 4.0]
```

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 ?

Schéma récursif en commun

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

1. si `l` est la liste vide, la valeur retournée par `f` ne dépend pas de `l` : c'est le cas de base de la récursion ;
2. sinon, `l` est de la forme `x::s` et la valeur retournée est calculée en effectuant une opération à partir de `x` et `f s`

Itération d'ordre supérieur

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

- L'argument `acc` représente la valeur retournée pour le cas de base de la récursion
- La fonction `f` est appliquée à chaque élément de la liste ainsi qu'au résultat de l'appel récursif

Exemple : la fonction `somme`

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)
```

Étape 1 : extraire l'opération récursive

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

Étape 2 : abstraire l'opération récursive

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

let somme l = somme_fold (fun a b -> a + b) l
```

Étape 4 : abstraire l'accumulateur

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

ou plus simplement

```
let somme l = somme_fold (+) l 0
```

Déroulons tout ça

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

Déroulons tout ça

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

Déroulons tout ça

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

Déroulons tout ça

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

Déroulons tout ça

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

Déroulons tout ça

`somme [1;2;3]`

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

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

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

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

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

Déroulons tout ça

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

Déroulons tout ça

$\texttt{somme [1;2;3]}$

$\Rightarrow \texttt{somme\_fold (+) [1;2;3] 0}$

$\Rightarrow \texttt{(+)}\ 1\ \texttt{(somme\_fold (+) [2;3] 0)}$

$\Rightarrow \texttt{(+)}\ 1\ \big(\texttt{(+)}\ 2\ \texttt{(somme\_fold (+) [3] 0)}\big)$

$\Rightarrow \texttt{(+)}\ 1\ \big(\texttt{(+)}\ 2\ \big(\texttt{(+)}\ 3\ \texttt{(somme\_fold (+) [] 0)}\big)\big)$

$\Rightarrow \big(+\big)\ 1\ \big(\texttt{(+)}\ 2\ \big(\texttt{(+)}\ 3\ 0\big)\big)$

$\Rightarrow \texttt{(+)}\ 1\ \big(\texttt{(+)}\ 2\ 3\big)$

$\Rightarrow \texttt{(+)}\ 1\ 5$

Déroulons tout ça

somme [1;2;3]

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

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

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

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

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

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

$\Rightarrow$ (+) 1 5

$\Rightarrow$ 6

L'itérateur `fold_right`

La fonction `somme_fold` n'est pas spécifique au calcul de la somme d'une liste d'entiers. Son schéma récursif est capturé par la fonction suivante :

$$\begin{array}{rcl}\texttt{fold\_right f } [\,] \texttt{ acc} & = & \texttt{acc} \\ \texttt{fold\_right f } [\,e_1; e_2; \ldots; e_n\,] \texttt{ acc} & = & \texttt{f } e_1 \ (f\ e_2\ (\cdots (f\ e_n\ acc) \cdots))\end{array}$$

Cette fonction s'appelle `List.fold_right` dans la bibliothèque standard de OCaml :

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

L'itérateur `fold_right`

La fonction `somme_fold` n'est pas spécifique au calcul de la somme d'une liste d'entiers. Son schéma récursif est capturé par la fonction suivante :

$$
\begin{array}{rcl}
\texttt{fold\_right f } [\,] \texttt{ acc} & = & \texttt{acc} \\
\texttt{fold\_right f } [\,e_1; e_2; \ldots; e_n\,] \texttt{ acc} & = & \texttt{f } e_1\ (f\ e_2\ (\cdots (f\ e_n\ acc)\cdots))
\end{array}
$$

Cette fonction s'appelle `List.fold_right` dans la bibliothèque standard de OCaml :

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

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

L'itérateur `fold_right`

La fonction `somme_fold` n'est pas spécifique au calcul de la somme d'une liste d'entiers. Son schéma récursif est capturé par la fonction suivante :

$$\begin{array}{rcl} \texttt{fold\_right}\ \texttt{f}\ [\,]\ \texttt{acc} & = & \texttt{acc} \\ \texttt{fold\_right}\ \texttt{f}\ [\,e_1; e_2; \ldots; e_n\,]\ \texttt{acc} & = & \texttt{f}\ e_1\ (f\ e_2\ (\cdots (f\ e_n\ acc) \cdots)) \end{array}$$

Cette fonction s'appelle `List.fold_right` dans la bibliothèque standard de OCaml :

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

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

Attention : cette fonction n'est pas récursive terminale !

L'itérateur récursif terminal `fold_left`

Pour les fonctions dont l'ordre d'application de l'opération sur `x` et `g s` n'est pas important, on peut utiliser le parcours suivant :

$$\begin{array}{rcl} \texttt{fold\_left f acc} \, [\,] & = & \texttt{acc} \\ \texttt{fold\_left f acc} \, [\, e_1; e_2; \ldots; e_n \,] & = & \texttt{f} \, (\cdots \, (f \, (f \; acc \; e_1) \; e_2) \cdots) \; e_n \end{array}$$

Cette fonction s'appelle `List.fold_left` dans la bibliothèque standard de OCaml :

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

L'itérateur récursif terminal `fold_left`

Pour les fonctions dont l'ordre d'application de l'opération sur `x` et `g s` n'est pas important, on peut utiliser le parcours suivant :

$$\begin{aligned} \texttt{fold\_left f acc } [\,] &= \texttt{acc} \\ \texttt{fold\_left f acc } [\,e_1; e_2; \ldots; e_n\,] &= \texttt{f } (\cdots\ (f\ (f\ acc\ e_1)\ e_2) \cdots)\ e_n \end{aligned}$$

Cette fonction s'appelle `List.fold_left` dans la bibliothèque standard de OCaml :

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

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

L'itérateur récursif terminal `fold_left`

Pour les fonctions dont l'ordre d'application de l'opération sur `x` et `g s` n'est pas important, on peut utiliser le parcours suivant :

$$\begin{array}{rcl} \texttt{fold\_left f acc } [\,] & = & \texttt{acc} \\ \texttt{fold\_left f acc } [e_1; e_2; \ldots; e_n] & = & \texttt{f } (\cdots (f\ (f\ acc\ e_1)\ e_2) \cdots)\ e_n \end{array}$$

Cette fonction s'appelle `List.fold_left` dans la bibliothèque standard de OCaml :

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

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

Cette fonction est récursive terminale !

Exemple 1 : somme d’une liste d’entiers (suite)

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

```
let somme l = fold_left (+) 0 l
```

Exemple 1 : somme d'une liste d'entiers (suite)

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

```
let somme l = fold_left (+) 0 l

val somme : int list -> int
```

Exemple 1 : somme d'une liste d'entiers (suite)

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

```
let somme l = fold_left (+) 0 l

val somme : int list -> int
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
⇒    fold_left (+) ((+) 0 1) [2;3]
```

Exemple 1 : somme d'une liste d'entiers (suite)

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

```
let somme l = fold_left (+) 0 l
```

```
val somme : int list -> int
```

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

Exemple 1 : somme d’une liste d’entiers (suite)

On peut écrire la fonction `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]
⇒    fold_left (+) ((+) 0 1) [2;3]
=    fold_left (+) 1 [2;3]
⇒    fold_left (+) ((+) 1 2) [3]
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
⇒  fold_left (+) ((+) 0 1) [2;3]
=  fold_left (+) 1 [2;3]
⇒  fold_left (+) ((+) 1 2) [3]
=  fold_left (+) 3 [3]
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
⇒  fold_left (+) ((+) 0 1) [2;3]
=  fold_left (+) 1 [2;3]
⇒  fold_left (+) ((+) 1 2) [3]
=  fold_left (+) 3 [3]
⇒  fold_left (+) ((+) 3 3) []
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
⇒  fold_left (+) ((+) 0 1) [2;3]
=  fold_left (+) 1 [2;3]
⇒  fold_left (+) ((+) 1 2) [3]
=  fold_left (+) 3 [3]
⇒  fold_left (+) ((+) 3 3) []
=  fold_left (+) 6 []
```

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `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]
⇒  fold_left (+) ((+) 0 1) [2;3]
=  fold_left (+) 1 [2;3]
⇒  fold_left (+) ((+) 1 2) [3]
=  fold_left (+) 3 [3]
⇒  fold_left (+) ((+) 3 3) []
=  fold_left (+) 6 []
⇒  6
```

Exemple 2 : Longueur d'une liste

Rappel : version sans itérateur

```
let rec longueur l =
  match l with
   |  [] -> 0
   | x::s -> 1 + (longueur s)

val longueur : 'a list -> int
```

Exemple 2 : Longueur d'une liste

Rappel : version sans itérateur

```
let rec longueur l =
  match l with
   |  [] -> 0
   | x::s -> 1 + (longueur s)

val longueur : 'a list -> int
```

Version avec itérateur :

```
let longueur l = fold_left (fun acc x -> 1 + acc) 0 l

longueur : 'a list -> int
```

```
# longueur [3;2;1;4];;

- : int = 4
```

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

$$\begin{aligned} & \texttt{longueur [3;2;1;4]} \\ = \quad & \texttt{fold\_left}\ \texttt{plus1}\ 0\ [3;2;1;4] \end{aligned}$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

$$
\begin{array}{ll}
 & \texttt{longueur [3;2;1;4]} \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 0\ [3;2;1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 0\ 3)\ [2;1;4]
\end{array}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

$$
\begin{aligned}
& \texttt{longueur [3;2;1;4]} \\
= \quad & \texttt{fold\_left plus1 0 } [3; 2; 1; 4] \\
\Rightarrow \quad & \texttt{fold\_left plus1 } (\texttt{plus1 0 3}) \ [2; 1; 4] \\
= \quad & \texttt{fold\_left plus1 1 } [2; 1; 4]
\end{aligned}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

$$
\begin{array}{rl}
& \texttt{longueur [3;2;1;4]} \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 0\ [3;2;1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 0\ 3)\ [2;1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 1\ [2;1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 1\ 2)\ [1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 2\ [1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 2\ 1)\ [1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 3\ [4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 3\ 4)\ [] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 4\ []
\end{array}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [3;2;1;4]`

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

$$
\begin{array}{rl}
 & \texttt{longueur [3;2;1;4]} \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 0\ [3;2;1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 0\ 3)\ [2;1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 1\ [2;1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 1\ 2)\ [1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 2\ [1;4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 2\ 1)\ [1;4] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 3\ [4] \\
\Rightarrow & \texttt{fold\_left}\ \texttt{plus1}\ (\texttt{plus1}\ 3\ 4)\ [] \\
= & \texttt{fold\_left}\ \texttt{plus1}\ 4\ [] \\
= & 4
\end{array}
$$

Exemple 3 : concaténation de deux listes

Rappel : version sans itérateur

```
let rec append l1 l2 =
  match l1 with
   | [] -> l2
   | x::s -> x::(append s l2)
append : 'a list -> 'a list -> 'a list
```

Exemple 3 : concaténation de deux listes

Rappel : version sans itérateur

```
let rec append l1 l2 =
  match l1 with
   | [] -> l2
   | x::s -> x::(append s l2)
append : 'a list -> 'a list -> 'a list
```

Version avec itérateur

```
# let append l1 l2 =
    fold_right (fun x acc -> x::acc) l1 l2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Exemple 3 : concaténation de deux listes

Rappel : version sans itérateur

```
let rec append l1 l2 =
  match l1 with
   | [] -> l2
   | x::s -> x::(append s l2)
append : 'a list -> 'a list -> 'a list
```

Version avec itérateur

```
# let append l1 l2 =
    fold_right (fun x acc -> x::acc) l1 l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

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

Exemple 3 : concaténation de deux listes

Rappel : version sans itérateur

```
let rec append l1 l2 =
  match l1 with
   | [] -> l2
   | x::s -> x::(append s l2)
append : 'a list -> 'a list -> 'a list
```

Version avec itérateur

```
# let append l1 l2 =
    fold_right (fun x acc -> x::acc) l1 l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

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

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

Autres fonctions

```
let zeros = fold_left (fun acc x -> x=0 && acc) true

val zeros : int list -> bool = <fun>
```

Autres fonctions

```
let zeros = fold_left (fun acc x -> x=0 && acc) true

val zeros : int list -> bool = <fun>
```

```
let recherche n =
    fold_left (fun acc x -> x=n || acc) false

val recherche : 'a -> 'a list -> bool = <fun>
```

Autres fonctions

```
let zeros = fold_left (fun acc x -> x=0 && acc) true

val zeros : int list -> bool = <fun>



let recherche n =
    fold_left (fun acc x -> x=n || acc) false

val recherche : 'a -> 'a list -> bool = <fun>



let existe p = fold_left (fun acc x -> p x || acc) false

val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Exemple 4 : liste des sous-listes d'une liste (1/2)

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)
```

Exemple 4 : liste des sous-listes d'une liste (1/2)

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
```

Exemple 4 : liste des sous-listes d'une liste (2/2)

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
```

Exemple 4 : liste des sous-listes d'une liste (2/2)

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];;
```

Exemple 4 : liste des sous-listes d'une liste (2/2)

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]; []]
```

Exemple 5 : `sous_liste` avec itérateurs

On commence par définir l'itérateur `map` de type

```
('a -> 'b) -> 'a list -> 'b list
```

tel que `map f [`e_1`; ...; `e_n`] = [f `e_1`; ...; f `e_n`]`

```
let rec map f l =
   match l with
  | [] -> []
  | x::s -> let v = f x in v :: (map f s)
```

Le fonction `sous_liste` peut alors s'écrire (avec `@` l'opérateur de concaténation en OCaml) :

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

Itérateurs vs. fonctions récursives (1/2)

Les versions sans itérateurs des fonctions `zeros`, `recherche` et `existe` sont plus efficaces que leurs versions avec itérateurs respectives

```
let rec existe p l =
  match l with
   | [] -> false
   | x::s -> p x || (existe p s)
```

On note `p` la fonction `fun x -> x=0`

```
existe p [3;0;2;1;4]
```

Itérateurs vs. fonctions récursives (1/2)

Les versions sans itérateurs des fonctions `zeros`, `recherche` et `existe` sont plus efficaces que leurs versions avec itérateurs respectives

```
let rec existe p l =
  match l with
   | [] -> false
   | x::s -> p x || (existe p s)
```

On note `p` la fonction `fun x -> x=0`

```
     existe p [3;0;2;1;4]
⇒  p 3 || existe p [0;2;1;4]
```

Itérateurs vs. fonctions récursives (1/2)

Les versions sans itérateurs des fonctions `zeros`, `recherche` et `existe` sont plus efficaces que leurs versions avec itérateurs respectives

```
let rec existe p l =
  match l with
   | [] -> false
   | x::s -> p x || (existe p s)
```

On note `p` la fonction `fun x -> x=0`

```
       existe p [3;0;2;1;4]
  ⇒   p 3 || existe p [0;2;1;4]
  =    existe p [0;2;1;4]
```

Les versions sans itérateurs des fonctions `zeros`, `recherche` et `existe` sont plus efficaces que leurs versions avec itérateurs respectives

```
let rec existe p l =
  match l with
   | [] -> false
   | x::s -> p x || (existe p s)
```

On note `p` la fonction `fun x -> x=0`

```
     existe p [3;0;2;1;4]
⇒   p 3 || existe p [0;2;1;4]
=   existe p [0;2;1;4]
⇒   p 0 || existe p [2;1;4]
=   true
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
          existe p [3;0;2;1;4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

$$\begin{array}{rl} & \texttt{existe p [3;0;2;1;4]} \\ = & \texttt{fold\_left f false [0;2;1;4]} \end{array}$$

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

$$
\begin{array}{ll}
 & \texttt{existe p [3;0;2;1;4]} \\
= & \texttt{fold\_left f false [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 3 || false) [0;2;1;4]}
\end{array}
$$

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
    existe p [3;0;2;1;4]
=   fold_left f false [0;2;1;4]
⇒   fold_left f (p 3 || false) [0;2;1;4]
=   fold_left f false [0;2;1;4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
     existe p [3;0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 3 || false) [0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 0 || false) [2;1;4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

$$
\begin{array}{rl}
 & \texttt{existe p [3;0;2;1;4]} \\
= & \texttt{fold\_left f false [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 3 || false) [0;2;1;4]} \\
= & \texttt{fold\_left f false [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 0 || false) [2;1;4]} \\
= & \texttt{fold\_left f true [0;2;1;4]}
\end{array}
$$

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
     existe p [3;0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 3 || false) [0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 0 || false) [2;1;4]
=    fold_left f true [0;2;1;4]
⇒    fold_left f (p 2 || true) [1;4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
    existe p [3;0;2;1;4]
=   fold_left f false [0;2;1;4]
⇒   fold_left f (p 3 || false) [0;2;1;4]
=   fold_left f false [0;2;1;4]
⇒   fold_left f (p 0 || false) [2;1;4]
=   fold_left f true [0;2;1;4]
⇒   fold_left f (p 2 || true) [1;4]
=   fold_left f true [1;4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
     existe p [3;0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 3 || false) [0;2;1;4]
=    fold_left f false [0;2;1;4]
⇒    fold_left f (p 0 || false) [2;1;4]
=    fold_left f true [0;2;1;4]
⇒    fold_left f (p 2 || true) [1;4]
=    fold_left f true [1;4]
⇒    fold_left f (p 1 || true) [4]
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

$$
\begin{array}{ll}
 & \texttt{existe p [3;0;2;1;4]} \\
= & \texttt{fold\_left f false [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 3 || false) [0;2;1;4]} \\
= & \texttt{fold\_left f false [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 0 || false) [2;1;4]} \\
= & \texttt{fold\_left f true [0;2;1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 2 || true) [1;4]} \\
= & \texttt{fold\_left f true [1;4]} \\
\Rightarrow & \texttt{fold\_left f (p 1 || true) [4]} \\
= & \texttt{fold\_left f true [4]}
\end{array}
$$

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
     existe p [3;0;2;1;4]
=  fold_left f false [0;2;1;4]
⇒  fold_left f (p 3 || false) [0;2;1;4]
=  fold_left f false [0;2;1;4]
⇒  fold_left f (p 0 || false) [2;1;4]
=  fold_left f true [0;2;1;4]
⇒  fold_left f (p 2 || true) [1;4]
=  fold_left f true [1;4]
⇒  fold_left f (p 1 || true) [4]
=  fold_left f true [4]
⇒  fold_left f (p 4 || true) []
```

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let f =  fun acc x -> p x || acc
let existe p = fold_left f false
```

```
   existe p [3;0;2;1;4]
=  fold_left f false [0;2;1;4]
⇒  fold_left f (p 3 || false) [0;2;1;4]
=  fold_left f false [0;2;1;4]
⇒  fold_left f (p 0 || false) [2;1;4]
=  fold_left f true [0;2;1;4]
⇒  fold_left f (p 2 || true) [1;4]
=  fold_left f true [1;4]
⇒  fold_left f (p 1 || true) [4]
=  fold_left f true [4]
⇒  fold_left f (p 4 || true) []
=  fold_left f true []
=  true
```