

Algorithmique et Approche Fonctionnelle

Cours 7 : Itérateurs sur les Listes

29 Octobre 2007

schéma récursif en commun

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

- 1 si l est la **liste vide**, la valeur retournée par g ne dépend pas de g : 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 de $g(s)$**

schémas de définitions récursives

les fonctions suivantes, vues dans les cours précédents, ont toutes la même structure :

- la fonction `zeros` (cf. cours 4)
- la fonction `recherche` (cf. cours 4)
- la fonction `longueur` (cf. cours 4)
- la fonction `append` (cf. cours 4)
- la fonction `rev_append` (cf. cours 4)
- la fonction `map` (cf. cours 6)
- la fonction `trier` (cf. cours 5)
- la fonction `partage` (cf. cours 5)
- la fonction `couper` (cf. cours 5)
- la fonction `existe` (cf. cours 6)
- la fonction `filtre` (cf. cours 6)

laquelle ?

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 ($g(s)$ dans le transparent précédent)

le schéma récursif commun suggère d'appliquer l'opération f sur la liste l de la **droite vers la gauche**
 exemple, calcul de la somme d'une liste d'entiers, en supposant que la fonction f est `fun x acc -> x + acc`

```
# let rec somme l =
  match l with
  [] -> 0
  | x::s -> f x (somme s);;
val somme : int list -> int = <fun>
```

```
somme [1;2;3] => f 1 (somme [2;3])
              => f 1 (f 2 (somme [3]))
              => f 1 (f 2 (f 3 (somme [])))
              => f 1 (f 2 (f 3 0))
              => f 1 (f 2 3)
              => f 1 5
              => 6
```

exemple 1 : somme d'une liste d'entiers

```
# let somme l = fold_right (fun x acc -> x + acc) l 0;;
val somme : int list -> int = <fun>
```

on note f la fonction `(fun acc x -> x + acc)`

```
somme [1;2;3]
= fold_right f [1;2;3] 0
=> f 1 (fold_right f [2;3] 0)
=> f 1 (f 2 (fold_right f [3] 0))
=> f 1 (f 2 (f 3 (fold_right f [] 0)))
=> f 1 (f 2 (f 3 0))
=> f 1 (f 2 3)
=> f 1 5
=> 6
```

$$\text{fold_right } f \ [] \ \text{acc} = \text{acc}$$

$$\text{fold_right } f \ [e_1; e_2; \dots; e_n] \ \text{acc} = f \ e_1 \ (f \ e_2 \ (\dots (f \ e_n \ \text{acc}) \dots))$$

le code de l'itération **droite-gauche** est le suivant

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

fonction prédéfinie en Ocaml : `List.fold_right`

cette fonction n'est pas récursive terminale!

itération généraliste gauche-droite

pour les fonctions récursives terminales, ou pour celles dont l'ordre d'application de l'opération sur x et $g \ s$ n'est pas important, on peut utiliser un parcours **gauche-droite**

$$\text{fold_left } f \ \text{acc} \ [] = \text{acc}$$

$$\text{fold_left } f \ \text{acc} \ [e_1; e_2; \dots; e_n] = f \ (\dots (f \ (f \ \text{acc} \ e_1) \ e_2) \dots) \ e_n$$

le code de l'itération **gauche-droite** est le suivant

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

fonction prédéfinie en Ocaml : `List.fold_left`

cette fonction est récursive terminale!

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

```
# let somme l = fold_left (fun acc x -> x + acc) 0 l;;  
val somme : int list -> int = <fun>
```

on note f la fonction (fun acc x -> x + acc)

```
    somme [1;2;3]  
= fold_left f 0 [1;2;3]  
⇒ fold_left f (f 0 1) [2;3]  
= fold_left f 1 [2;3]  
⇒ fold_left f (f 1 2) [3]  
= fold_left f 3 [3]  
⇒ fold_left f (f 3 3) []  
= fold_left f 6 []  
⇒ 6
```

exemple 2 : calcul de la longueur d'une liste

version sans itérateur

```
# let rec longueur l =  
  match l with  
  [] -> 0  
  | x::s -> 1 + (longueur s);;  
val longueur : 'a list -> int = <fun>
```

version avec itérateur

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

```
# 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)

```
    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) [4]  
= fold_left plus1 3 [4]  
⇒ fold_left plus1 (plus1 3 4) []  
= fold_left plus1 4 []  
= 4
```

exemple 3 : concaténation de deux listes

version sans itérateur

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

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]
```

```
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 longueur = fold_left (fun acc x -> 1+acc) 0
val longueur : 'a list -> int = <fun>

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

let rev_append l1 l2 = fold_left (fun acc x -> x::acc) l2 l1
val rev_append : 'a list -> 'a list -> 'a list = <fun>

let map f l = fold_right (fun x acc -> (f x)::acc) l []
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

exemple 4 : liste des sous-listes d'une liste

(1/2)

la fonction sous_listes retourne la liste des sous-listes d'une liste l

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

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

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

```
let trier rel_ordre l = fold_right (inserer rel_ordre) l []
val trier : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

let partage p =
  fold_left
  (fun (g,d) x -> if x<=p then (x::g,d) else (g,x::d)) ([],[])
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>

let couper =
  fold_left (fun (l1,l2) x -> (x::l2,l1)) ([],[])
val couper : 'a list -> 'a list * 'a list = <fun>

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

let filtre p =
  fold_left (fun acc x -> if p x then x::acc else acc) []
val filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

exemple 4 : liste des sous-listes d'une liste

(2/2)

version avec itérateurs

```
#let sous_listes l =
  fold_left (fun p x -> (map (fun l->x::l) p)@p) [[]] l;;
val sous_listes : 'a list -> 'a list list = <fun>

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

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
```

la version récursive ne s'arrête pas

```
# let existe p = List.fold_left (fun acc x -> p x || acc) false
```

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