Cours de programmation fonctionnelle et logique
Cours 6 : listes et fonctionnelles

Table des matières

1 Fonctionnelles simples sur les listes **1**
1.1 Application successive d'une action à tous les éléments d'une liste 1
1.2 Application en parallèle d'une action à tous les éléments d'une liste 1

2 Fonctionnelles complexes sur les listes **2**
2.1 Notion d'abstraction de schéma de programme 2
2.2 Autre schéma récursif : l'accumulation 3
2.3 Accumuler avec les éléments d'une liste 3
2.4 Accumuler encore... 4

1 Fonctionnelles simples sur les listes

1.1 Application successive d'une action à tous les éléments d'une liste

Soit une fonction `f` et une liste `l`. La fonctionnelle `List.iter` applique `f` tour à tour à tous les éléments de `l`. Autrement dit, évaluer `List.iter f [e1;e2;e3;...;en]` est équivalent à exécuter la séquence`begin f e1; f e2; f e3; ...; f en; () end`.

```
#List.iter print_int [1;2;3];;
 123- : unit = ()
```

Nous pouvons écrire nous-mêmes une fonction équivalente.

```
#List.iter;;
 - : ('a -> unit) -> 'a list -> unit = <fun>

#let rec iter f = function
     []  -> ()
  | t::r -> f t; iter f r;;
    val iter : ('a -> 'b) -> 'a list -> unit = <fun>

#let rec iter f = function
     []  -> ()
  |  [a] -> f a
  | t::r -> f t; iter f r;;
      val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

1.2 Application en parallèle d'une action à tous les éléments d'une liste

La fonctionnelle `List.map` retourne la liste obtenue par application d'une fonction `f` à chaque élément d'une liste `l`. Autrement dit, évaluer `List.map f [e1;e2;e3;...;en]` équivaut à évaluer `[f e1;f e2;f e3;...;f en]`. `List.map` distribue la fonction `f` sur les éléments d'une liste.

```
#List.map (function x -> x+1) [1;2;3];;
 - : int list = [2; 3; 4]
```

Nous pouvons écrire nous-mêmes une fonction équivalente.

```
#List.map;;
 - : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
#let rec map f = function
      []  -> []
   | t::r -> f t :: map f r;;
     val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

2 Fonctionnelles complexes sur les listes

2.1 Notion d'abstraction de schéma de programme

Somme des éléments d'une liste

```
#let rec somme = function
      []  -> 0
   | t::r -> t + (somme r);;
     val somme : int list -> int = <fun>
```

Produit des éléments d'une liste

```
#let rec produit = function
      []  -> 1
   | t::r -> t * (produit r);;
     val produit : int list -> int = <fun>
```

Concaténation de toutes les chaînes d'une liste

```
#let rec implose = function
      [] -> ""
   | t::r -> t^(implose r);;
     val implose : string list -> string = <fun>
```

Concaténation des éléments d'une liste de listes

```
#let rec concatene_listes = function
     []   -> []
   | t::r -> t @ concatene_listes r;;
     val concatene_listes : 'a list list -> 'a list = <fun>
```

On voit se dessiner un schéma de programme

```
let rec f = function
    []  -> élément_neutre
 | t::r -> x opération_infixe (f l);;
```

Pour obtenir une fonctionnelle générale qui implémente ce schéma de programme, il nous suffit « d'abstraire » l'opération infixe et l'élément neutre.

Ici, toutes les opérations sont infixes : il faut les remplacer par des opérations préfixes équivalentes. Par exemple on remplace « + » par `add`

```
#let add = function x -> function y -> x + y;;
 val add : int -> int -> int = <fun>
```

```
#let rec itérateur f b = function
      []  -> b
   | t::r -> f t (itérateur f b r);;
     val itérateur : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
#let somme_i = itérateur add 0;;
 val somme_i : int list -> int = <fun>
```

Longueur d'une liste

```
#let rec taille = function
     [] -> 0
  | t::r -> 1 + (taille r);;
     val taille : 'a list -> int = <fun>
```

```
#let longueur = itérateur (function x -> function y -> 1+y) 0;;
 val longueur : '_a list -> int = <fun>
```

Concaténation de deux listes (éléments pris dans l'ordre inverse)

```
#let rec concat l2 = function
     []  -> l2
  | t::r -> t :: concat r l2;;
     val concat : 'a list -> 'a list -> 'a list = <fun>
```

```
#let concat_i l2 = itérateur (function x -> function y-> x :: y) l2;;
 val concat_i : 'a list -> 'a list -> 'a list = <fun>
```

2.2 Autre schéma récursif : l'accumulation

Somme avec accumulateur

```
#let rec somme_accu accu = function
     []  -> accu
  | t::r -> somme_accu (accu + t) r;;
     val somme_accu : int -> int list -> int = <fun>
```

```
#let somme l = somme_accu 0 l;;
 val somme : int list -> int = <fun>
```

Fonctionnelle générale

```
#let rec accumulateur_sur_listes f accu = function
     []  -> accu
  | t::r -> accumulateur_sur_listes f (f t accu) r;;
     val accumulateur_sur_listes : ('a -> 'b -> 'b) -> 'b -> 'a list ->
'b = <fun>
```

```
#let somme_accu_i = accumulateur_sur_listes (function x-> function y-> x+y) 0;;
 val somme_accu_i : int list -> int = <fun>
```

2.3 Accumuler avec les éléments d'une liste

La fonctionnelle `List.fold_left f b l` effectue de multiples compositions de la fonction à deux arguments `t`, en utilisant les éléments de la liste `l` comme seconds arguments de `f`. La valeur de base `b` est utilisée pour le premier argument du premier appel. Le résultat de chaque appel est passé en premier argument de l'appel suivant.

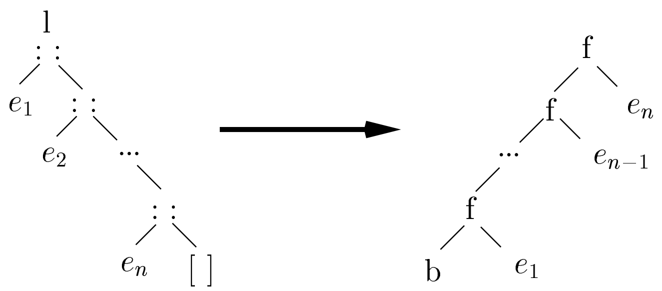
Le schéma d'application de cette fonction est donc le suivant :

`List.fold_left f b [e1;e2; ... ; en] = f(...(f(f b e1) e2)...)en`

Comme précédemment, nous pouvons écrire notre propre version de cette fonctionnelle.

```
#List.fold_left;;
 - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
#let rec fold_left f b = function
     []  -> b
  | t::r -> fold_left f (f b t) r;;
     val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```



Si f est l'addition on obtient : `List.fold_left f b [e1 ; e2 ; .... en] = b + e1 + e2 ... + en`.

```
#let somme =  List.fold_left (function x -> function y-> x + y) 0;;
 val somme : int list -> int = <fun>
```

Concaténation des éléments d'une liste de chaînes de caractères.

```
#let concat_left = List.fold_left (function x -> function y-> x ^ y) "";;
 val concat_left : string list -> string = <fun>
```

```
#concat_left ["a";"b";"c"];;
 - : string = "abc"
```

2.4 Accumuler encore...

`List.fold_right f l b` effectue de multiples compositions de la fonction `f` deux arguments, en utilisant les éléments de `l` comme premier argument. L'élément de base `b` est utilisé comme second argument lors du premier appel, puis le résultat de chaque appel est utilisé comme second argument de l'appel suivant.

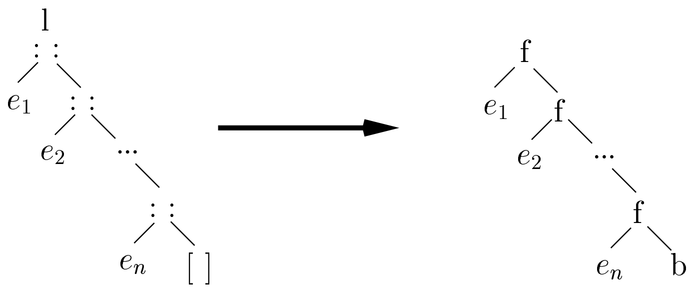
Le schéma d'application de cette fonction est donc le suivant :

`List.fold_right f [e1 ;e2 ; ... ; en] b = f e1 (f e2 (...(f en b) ...))`

Comme précédemment, nous pouvons écrire notre propre version de cette fonctionnelle.

```
#List.fold_right;;
 - : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

```
#let rec fold_right f l b = match l with
     []  -> b
  | t::r -> f t (fold_right f r b);;
    val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```



Concaténation des éléments d'une liste de chaînes de caractères.

```
#let concat_right l = List.fold_right (function x -> function y-> x ^ y) l "";;
 val concat_right : string list -> string = <fun>
```

```
#concat_right ["a";"b";"c"];;
 - : string = "abc"
```

Copie d'une liste (remplaçant `: :` par `: :` et `[]` par `[]`)

```
#let copie_liste l = List.fold_right (function x-> function y -> x::y) l [];;
 val copie_liste : 'a list -> 'a list = <fun>
```

Fonction de concaténation de deux listes

```
#let concatene l1 l2 = List.fold_right (function x-> function y -> x::y) l1 l2;;
 val concatene : 'a list -> 'a list -> 'a list = <fun>
```

ou

```
#let concatene = List.fold_right (function x-> function y -> x::y);;
 val concatene : '_a list -> '_a list -> '_a list = <fun>
```