

Programmation Fonctionnelle

Cours 03

Michele Pagani



Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes
pagani@pps.univ-paris-diderot.fr

28 septembre 2015

Listes

type list

- des valeurs d'un même type peuvent être regroupées en listes:

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

```
# ['a'; 'b'; 'c'];;  
- : char list = ['a'; 'b'; 'c']
```

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

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

- attention:** tous les éléments de la même liste doivent être du même type

```
# [1; "deux"; 3];;  
      ^^^
```

Error: This expression has **type** string but an expression was expected **of type** int

construire une list

Une liste est soit vide soit a une tête et une queue

liste vide [] a un type **polymorphe** (⇒ plus tard)

```
# [];;  
- : 'a list = []
```

- pour tout type a, il y a la liste vide []
- 'a est une **variable de type**.

constructeur :: ajoute une tête a une queue

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

```
# 1::2::3::[];; (*associe a droite*)  
- : int list = [1; 2; 3]
```

```
# 1::2::3;;  
      ^^^
```

Error: This expression has **type** int but an expression was expected **of type** int list

de-construire une list

- L'extraction des éléments d'une liste se fait par **filtrage par motif**, ou **pattern-matching** en anglais.

```
match x with
| motif1 -> expr1
| motif2 -> expr2
...
| motifN -> exprN
```

- un **motif** (ou **pattern** en anglais) est une expression fait par variables et constructeurs.

Exemple:

```
# let tete x = match x with
| [] -> failwith "liste_vide"
| a::q -> a
val tete : 'a list -> 'a = <fun>
```

À quoi sert le filtrage par motif ?

① à faire une distinction de cas

```
# let f x = match x with
| [] -> "vide"
| t::q -> "pas_vide";;
val f : 'a list -> string = <fun>
```

```
# let f x = match x with
| [] -> "vide"
| [a] -> "un_element"
| a::b::q -> "plus_d'un_element"
val f : 'a list -> string = <fun>
```

② à accéder aux éléments de la liste

```
# let rec sum x = match x with
| [] -> 0
| a::q -> a + sum q
val sum : int list -> int = <fun>
```

```
# sum [0; 10; 2];;
- : int = 12
```

⇒ les variables du motif sont liées dans l'expression qui suit !

Une "esquisse" d'évaluation

$$\begin{aligned}
\text{sum } [10;2] &\xrightarrow{x=[10;2]} \text{match } x \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q \\
&\xrightarrow{x=[10;2]} \text{match } [10;2] \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q \\
&\xrightarrow{\substack{x=[10;2] \\ a=10 \\ q=[2]}} a + \text{sum } q \\
&\xrightarrow{\substack{x=[10;2] \\ a=10 \\ q=[2]}} 10 + \text{sum } [2] \\
&\xrightarrow{x=[2]} 10 + (\text{match } x \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q) \\
&\xrightarrow{x=[2]} 10 + (\text{match } [2] \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q) \\
&\xrightarrow{\substack{x=[2] \\ a=2 \\ q=[]}} 10 + (a + \text{sum } q) \\
&\xrightarrow{\substack{x=[2] \\ a=2 \\ q=[]}} 10 + 2 + \text{sum } [] \\
&\xrightarrow{x=[]} 10 + 2 + (\text{match } x \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q) \\
&\xrightarrow{x=[]} 10 + 2 + (\text{match } [] \text{ with } [] \rightarrow 0 \mid a::q \rightarrow a + \text{sum } q) \\
&\xrightarrow{x=[]} 10 + 2 + 0 \\
&\xrightarrow{x=[]} 12
\end{aligned}$$

Un peu de "sucre syntaxique"

(* les écritures suivantes sont équivalentes *)

```
let f x = match x with
| [] -> "vide"
| a::t -> "pas_vide"
```

```
let f x = match x with
| [] -> "vide"
| a::t -> "pas_vide"
```

```
let f = function
| [] -> "vide"
| a::t -> "pas_vide"
```

- Le filtrage par motif est un outil très général: ils s'applique à n'importe quel type, pas seulement aux listes

⇒ à voir plus tard...

list (concatenation)

- @ : concaténation de deux listes (infix)

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

Error: This expression has **type** int but an expression was expected of **type** 'a list

- Exercice: en utilisant le filtrage par motif, définir la fonction
append : 'a list -> 'a list -> 'a list contenant deux listes

Le module List

```
Module List

module List: sig .. end
List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

val length : 'a list -> int
  Return the length (number of elements) of the given list.

val hd : 'a list -> 'a
  Return the first element of the given list. Raise Failure "hd" if the list is empty.

val tl : 'a list -> 'a list
  Return the given list without its first element. Raise Failure "tl" if the list is empty.

val nth : 'a list -> int -> 'a
  Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise Failure "nth" if the list is too short. Raise Invalid_argument "List.nth" if n is negative.

val rev : 'a list -> 'a list
  List reversal.

val append : 'a list -> 'a list -> 'a list
  Concatenate two lists. Same function as the infix operator @. Not tail-recursive (length of the first argument). The @ operator is not tail-recursive either.

val rev_append : 'a list -> 'a list -> 'a list
  List.rev_append l1 l2 reverses l1 and concatenates it to l2. This is equivalent to List.rev l1 @ l2, but rev_append is tail-recursive and more efficient.

val concat : 'a list list -> 'a list
  Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

val flatten : 'a list list -> 'a list
  Same as concat. Not tail-recursive (length of the argument + length of the longest sub-list).

Iterators

val iter : ('a -> unit) -> 'a list -> unit
  List.iter f [a1; ...; an] applies function f in turn to a1; ...; an. It is equivalent to begin f a1; f a2; ...; f an; () end.

val iteri : (int -> 'a -> unit) -> 'a list -> unit
  Same as List.iter, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.
  Since 4.00.0

val map : ('a -> 'b) -> 'a list -> 'b list
  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. Not tail-recursive.

val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
  Same as List.map, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument. Not tail-recursive.
  Since 4.00.0

val rev_map : ('a -> 'b) -> 'a list -> 'b list
  List.rev_map f l gives the same result as List.rev (List.map f l), but is tail-recursive and more efficient.

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

Comment appeler fonctions d'un module ?

- on met le nom du module en prefix au nom de la fonction

```
List.rev (List.sort compare [3;5;2])
```

ou bien, en utilisant des parenthèses:

```
List.(rev (sort compare [3;5;2]))
```

- on ouvre le module par l'instruction open

```
open List;;
```

```
rev (sort compare [3;5;2])
```

⇒ faites attention aux conflits des noms !

list (head et tail)

```
List.hd : 'a list -> 'a  
List.tl : 'a list -> 'a list
```

```
# List.hd [1;2;3];;  
- : int = 1
```

```
# List.tl [1;2;3];;  
- : int list = [2; 3]
```

```
# List.tl 1;;  
  ^^
```

Error: This expression has **type** int but an expression was expected of **type** 'a list

```
# List.hd [];;  
Exception: Failure "hd". (*exception, pas erreur*)
```

```
# List.tl [];;  
Exception: Failure "tl". (*exception, pas erreur*)
```

list (head et tail)

- Une implémentation possible de hd dans le module List:

```
let hd = function
  [] -> failwith "hd"
  | t::q -> t;;
val hd : 'a list -> 'a = <fun>
```

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

```
# hd [];;
Exception: Failure "hd".
```

- le mécanisme d'**exception** permet de traiter les cas limites
(\Rightarrow plus tard)

list (fonction map)

List.map : ('a -> 'b) -> 'a list -> 'b list

- ① elle prend une fonction:
 $f : 'a \rightarrow 'b$
- ② une liste :
 $[e1; \dots ; en] : 'a \text{ list}$
- ③ et renvoie la liste:
 $[f(e1); \dots ; f(en)] : 'b \text{ list}$

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]
```

```
# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

list (recherche d'éléments)

List.find : ('a -> bool) -> 'a list -> 'a

- ① elle prend une fonction, représentante une condition:
 $c : 'a \rightarrow \text{bool}$
- ② une liste :
 $[e1; \dots ; en] : 'a \text{ list}$
- ③ et renvoie le premier élément qui satisfait la condition (s'il existe):
 $ei : 'a$

```
# List.find (fun x -> x mod 2 = 0) [5;3;2;4;1];;
- : int = 2
```

```
# List.find (fun x -> x mod 2 = 0) [5;3;1];;
Exception: Not_found.
```

list (recherche d'éléments)

List.filter : ('a -> bool) -> 'a list -> 'a list

- ① elle prend une fonction, représentante une condition:
 $c : 'a \rightarrow \text{bool}$
- ② une liste :
 $[e1; \dots ; en] : 'a \text{ list}$
- ③ et renvoie la liste des éléments qui satisfont la condition:
 $[ei1; \dots ; eik] : 'a \text{ list}$

```
# List.filter (fun x -> x mod 2 = 0) [5;3;2;4;1];;
- : int list = [2; 4]
```

```
# List.filter (fun x -> x mod 2 = 0) [5;3;1];;
- : int list = []
```

list (itérateurs)

List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

- 1 elle prend une fonction:
f : 'a -> 'b -> 'b
- 2 une liste de 'a:
[e1; e2; ... ;en] : 'a list
- 3 un élément de 'b:
x : 'b
- 4 et renvoie un élément de 'b:
f e1 (f e2 ... (f en x)...) : 'b

```
# List.fold_right (fun x y -> x^y) ["Hello"; "␣"; "world"] "!";;  
- : string = "Hello␣world!"
```

```
# List.fold_right (fun x y -> x^y) [ ] "!";;  
- : string = "!"
```

list (itérateurs)

List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

- 1 elle prend une fonction:
f : 'a -> 'b -> 'a
- 2 un élément de 'a:
x : 'a
- 3 une liste de 'b:
[e1; e2; ... ;en] : 'b list
- 4 et renvoie un élément de 'a:
f (... (f (f x e1) e2) ...) en : 'a

```
# List.fold_right (fun x y -> x^y) ["Hello"; "␣"; "world"] "!";;  
- : string = "Hello␣world!"
```

```
# List.fold_left (fun x y -> x^y) "!" ["Hello"; "␣"; "world"];;  
- : string = "!!Hello␣world"
```

Exercices

- 1 Écrire la fonction `sum` qui renvoie la somme des entiers d'une liste donnée en entrée en utilisant un itérateur de listes (`fold_left` ou `fold_right`).
- 2 Écrire une fonction `count_vowel` qui compte le nombre de voyelles dans une liste de caractères donnée en entrée. Par exemple: `count_vowel ['h'; 'e'; 'l'; 'l'; 'o']` évalué à 2.
- 3 Écrire une fonction `make_set` qui prend en entrée une liste `l` et renvoie une liste sans répétitions contenant tous les éléments de `l`. Par exemple: `make_set [0;3;2;3]` évalué à `[0;3;2]`.