

Programmation Fonctionnelle

Cours 3

Listes, filtrage par motif

Delia Kesner

Le type des listes

- ▶ Type prédéfini de OCaml
- ▶ Il n'y a pas un seul type liste, mais il y a des listes d'entiers, de flottants, de booléens, etc.
- ▶ On dit que `list` est un type *polymorphe* : pour tout type t , il y a un type $t\ \text{list}$.
- ▶ On peut former des listes d'éléments de n'importe quel type (fonctions ou listes incluses), mais les listes sont **homogènes** : tous les éléments de la même liste doivent être du même type.

Valeurs d'un même type I

```
[1; 2; 3];;  
# - : int list = [1; 2; 3]  
["ab"; "cd"];;  
# - : string list = ["ab"; "cd"]  
[4.0; 7e2];;  
# - : float list = [4.; 700.]  
[[1;2;3];[4;5]];;  
# - : int list list = [[1; 2; 3]; [4; 5]]  
[sin; cos];;  
# - : (float -> float) list = [<fun>; <fun>]  
[(fun x -> x>10); (fun x -> x<=4)];;  
# - : (int -> bool) list = [<fun>; <fun>]  
[(fun x -> x>10), (fun x -> x<=4)];;  
# - : ((int -> bool) * (int -> bool)) list = [(<fun>, <fun>)]
```

Mauvaises listes I

```
[42; 2.0; "toto"];;
```

```
[42; 2.0; "toto"];;
```

```
^^^^
```

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

```
[int_of_float; float_of_int];;
```

```
[int_of_float; float_of_int];;
```

```
^^^^^^^^^^^^^^^^
```

Error: This expression has **type** int -> float
but an expression was expected **of type** float -> int
Type int is **not** compatible **with type** float

Mauvaises listes II

```
[[1;2;3];4;5];;
```

```
[[1;2;3];4;5];;
```

^

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

Liste vide

- ▶ Quel est le type de la liste vide [] ?
 - ▶ 'a list (type polymorphe)
 - ▶ 'a est dite une variable de type
 - ▶ Toutes les variables de type commencent avec un symbole '
 - ▶ Les variables de types 'a, 'b, 'c se pronocent *alpha*, *beta*, *gamma*

Liste vide I

```
[];; (* liste vide *)  
# - : 'a list = []
```

```
[] < [1; 2];;  
# - : bool = true
```

```
[] < ["hello"; "goodbye"];;  
# - : bool = true
```

Construire une liste I

(* Construire une liste *)

[2];;

- : int list = [2]

2::[];;

- : int list = [2]

1::[2;3];;

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

1::2::[];;

- : int list = [1; 2]

(* Associe a droite *)

4::5::[6;7];;

- : int list = [4; 5; 6; 7]

4::(5::[6;7]);;

- : int list = [4; 5; 6; 7]

Construire une liste II

(* Mauvaises constructions *)

```
4::5::6;;
```

```
4::5::6;;  
      ^
```

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

```
(4::5)::[6;7];;
```

```
(4::5)::[6;7];;  
      ^
```

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

Construire une liste III

```
(4::5::[])::[6;7];;
```

```
(4::5::[])::[6;7];;  
      ^
```

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

```
(4::5::[])::[[6];[7]];;
```

```
# - : int list list = [[4; 5]; [6]; [7]]
```

Déconstruire une liste I

(* Destructeurs de listes *)

```
List.hd [1;2;3];;
```

```
# - : int = 1
```

```
List.tl [1;2;3];;
```

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

```
List.hd [];
```

```
# Exception: Failure "hd".
```

```
List.tl [];
```

```
# Exception: Failure "tl".
```

Comparer deux listes I

```
[1; 2] = [1; 2];;
```

```
# - : bool = true
```

```
[1; 2] = 1::[2];;
```

```
# - : bool = true
```

```
[1; 2] = [1; 1; 2];;
```

```
# - : bool = false
```

Comparer deux listes par l'ordre Lexicographique I

```
[] < [];;
```

```
# - : bool = false
```

```
[] < [2;3];;
```

```
# - : bool = true
```

```
[4; 2] < [4; 2; 3];;
```

```
# - : bool = true
```

```
[5; 3] < [5; 2; 3];;
```

```
# - : bool = false
```

```
[7; 2; 3] < [7; 3];;
```

```
# - : bool = true
```

Comparer deux listes par l'ordre Lexicographique II

```
[1; 4; 3] < [1; 3];;
```

```
# - : bool = false
```

```
[3;1000] < [5;3];;
```

```
# - : bool = true
```

```
[[4; 2; 3];[44; 52; 1]] < [[4; 3];[3; 80]];;
```

```
# - : bool = true
```

Fonctions de base sur les listes

- ▶ Les fonctions agissant sur les listes polymorphes ont des types polymorphes.
- ▶ L'opérateur `::` a le type :
`'a -> 'a list -> 'a list`
- ▶ La fonction `List.hd` donne le premier élément d'une liste :
`'a list -> 'a`
- ▶ La fonction `List.tl` donne la queue d'une liste :
`'a list -> 'a list`
- ▶ `List.append` concatène deux listes (en notation préfixe) **équivalent à** `@` (en **notation infix**)
`'a list -> 'a list -> 'a list = <fun>`
- ▶ Plus de fonctions dans le module `List` de la bibliothèque standard.

Autres fonctions utiles

- ▶ `List.length` donne la longueur d'une liste
- ▶ `List.nth` donne le n-ème élément d'une liste
- ▶ `List.rev` inverse une liste
- ▶ `List.flatten` aplatît une liste de listes
- ▶ `List.map` applique une fonction a tous les éléments d'une liste
- ▶ `List.find` renvoie le premier élément d'une liste vérifiant une condition donnée en entrée
- ▶ `List.filter` renvoie la sous-liste des éléments d'une liste vérifiant une condition donnée en entrée
- ▶ Les itérateurs `List.fold_right` et `List.fold_left`

Quelques fonctions sur les listes I

```
List.length [];;  
# - : int = 0  
List.length [1;4;8;9];;  
# - : int = 4  
List.append [1;2;3] [1;2;3];;  
# - : int list = [1; 2; 3; 1; 2; 3]  
List.append [[1;2;3];[4;3]] [[8;7]];;  
# - : int list list = [[1; 2; 3]; [4; 3]; [8; 7]]  
List.nth [1;4;8;9] 0;;  
# - : int = 1  
List.nth [1;4;8;9] 4;;  
# Exception: Failure "nth".  
List.nth [1;4;8;9] 0 + List.nth [1;4;8;9] 1;;  
# - : int = 5
```

Quelques fonctions sur les listes II

```
List.rev [1;4;8;9];;  
# - : int list = [9; 8; 4; 1]  
List.rev (List.rev [1;4;8;9]);;  
# - : int list = [1; 4; 8; 9]  
List.flatten [[1;2;3];[4;3];[8;7]];;  
# - : int list = [1; 2; 3; 4; 3; 8; 7]  
List.flatten [[1;1];[2;2];[3;3]];[[8;8];[7;7]];;  
# - : int list list =  
[[1; 1]; [2; 2]; [3; 3]; [8; 8]; [7; 7]]  
List.flatten (List.flatten [[1;1];[2;2];[3;3]];[[8;8];[7;7]]);;  
# - : int list = [1; 1; 2; 2; 3; 3; 8; 8; 7; 7]
```

La fonction map

`List.map;;`

`- : ('a -> 'b) -> 'a list -> 'b list = <fun>`

- ▶ Prend une fonction `f: 'a -> 'b`
- ▶ et une liste `[e1; .. ;en]: 'a list`
- ▶ et renvoie la liste `[f e1; .. ;f en]: 'b list`

La fonction map I

```
List.map sin [2. ; 2.4 ; 4.5] ;;  
# - : float list =  
[0.909297426825681709; 0.675463180551151;  
-0.977530117665097]
```

```
List.map (function x -> x+1) [82 ; 24 ; 5 ; 18] ;;  
# - : int list = [83; 25; 6; 19]
```

```
List.map (function x -> x < 10) [82 ; 24 ; 5 ; 18] ;;  
# - : bool list = [false; false; true; false]
```

La fonction find

List.find;;

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

- ▶ Prend une condition $c : 'a \rightarrow \text{bool}$
- ▶ et une liste $[e_1; \dots; e_n] : 'a \text{ list}$
- ▶ Reenvoie le premier élément de la liste e_i vérifiant la condition c si cet élément existe, erreur sinon

La fonction find

```
List.find (function x -> x < 10) [82 ; 24 ; 5 ; 18] ;;  
# - : int = 5
```

```
List.find (function x -> x < 10) [82 ; 24 ; 89 ; 18] ;;  
# Exception: Not_found.
```

La fonction filter

```
List.filter;;
```

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

- ▶ Prend une condition $c: 'a \rightarrow \text{bool}$
- ▶ et une liste $[e1; \dots; en]: 'a \text{ list}$
- ▶ et renvoie la liste de tous les éléments $[ei1; \dots; eik]$ de la liste originale $[e1; \dots; en]$ vérifiant la condition c

La fonction filter I

```
List.filter (function x -> x < 10) [2 ; 24 ; 5 ; 18] ;;  
# - : int list = [2; 5]
```

```
List.filter (function x -> x < 10) [82 ; 24 ; 89 ; 18] ;;  
# - : int list = []
```


Les itérateurs - motivations

- ▶ Plusieurs fonctions sur les listes ont exactement la même structure.
- ▶ On veut éviter l'écriture répétitive de toutes ces variantes, pour obtenir du code plus court.
- ▶ Tout en préservant du code fonctionnel pure.
- ▶ Cela favorise la généralité et la réutilisabilité.

Les motivations sur un exemple concret I

```
let rec sum l =  
  if l = [] then 0 else List.hd l + sum (List.tl l);;  
# val sum : int list -> int = <fun>
```

```
let rec mult l =  
  if l = [] then 1 else List.hd l * mult (List.tl l);;  
# val mult : int list -> int = <fun>
```

L'itérateur de droite

`List.fold_right;;`

`- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>`

- ▶ Prend une fonction `f: 'a -> 'b -> 'b`
- ▶ une liste `[e1; .. ;en]: 'a list`
- ▶ un élément `x: 'b`
- ▶ Si la liste est non vide, renvoie un élément construit comme suit :
`f e1 (f e2 ... (f en x)): 'b`
- ▶ Si la liste est vide, renvoie `x`

La fonction d'itération à droite I

```
List.fold_right ( + ) [2;4;6;8] 0;;
```

```
# - : int = 20
```

```
List.fold_right ( + ) [] 0;;
```

```
# - : int = 0
```

```
List.fold_right ( * ) [2;4;6;8] 1;;
```

```
# - : int = 384
```

```
List.fold_right ( * ) [] 1;;
```

```
# - : int = 1
```

```
List.fold_right (fun x y -> if x>y then x else y) [2;4;6;8] 0;;
```

```
# - : int = 8
```

```
List.fold_right (fun x y -> if x>y then x else y) [] 0;;
```

```
# - : int = 0
```

L'itérateur de gauche

`List.fold_left;;`

- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

- ▶ Prend une fonction $f: 'a \rightarrow 'b \rightarrow 'a$
- ▶ un élément $x: 'a$
- ▶ une liste $[e1; \dots; en]: 'b \text{ list}$
- ▶ Si la liste est non vide, renvoie un élément construit comme suit :
 $f \dots (f (f x e1) e2) \dots): 'a$
- ▶ Si la liste est vide, renvoie x

La fonction d'itération à gauche I

```
List.fold_left ( + ) 0 [2;4;6;8];;
```

```
# - : int = 20
```

```
List.fold_left ( + ) 0 [];;
```

```
# - : int = 0
```

```
List.fold_left ( * ) 1 [2;4;6;8];;
```

```
# - : int = 384
```

```
List.fold_left ( * ) 1 [];;
```

```
# - : int = 1
```

```
List.fold_left (fun x y -> if x>y then x else y) 0 [2;4;6;8];;
```

```
# - : int = 8
```

```
List.fold_left (fun x y -> if x>y then x else y) 0 [];;
```

```
# - : int = 0
```

Attention

Bien que similaires, les opérateurs `List.fold_right` et `List.fold_left` ne se comportent pas toujours de la même manière.

Différent calcul pour les deux itérateurs I

```
List.fold_right (-) [1;2;3] 0;; (* 1 - (2 - (3 - 0)) *)
```

```
# - : int = 2
```

```
List.fold_left (-) 0 [1;2;3];; (* ((0 - 1) - 2) - 3 *)
```

```
# - : int = -6
```


Différent type pour les deux itérateurs I

```
List.fold_right;;
```

```
# - : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

```
List.fold_left;;
```

```
# - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Différent code pour les deux itérateurs I

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

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

Itération droite et gauche

Si $f = +$ et $l = a :: r = [e1; \dots; en]$:

- ▶ Itération droite : $\text{fold_right } (+) a :: r \ 0 = (+) a (\text{fold_right } (+) r \ 0)$
- ▶ ce qui donne $(e1 + (e2 + \dots + (en + 0)))$
- ▶ N'est pas récursive terminale
- ▶ Itération gauche $\text{fold_left } (+) 0 a :: r = \text{fold_left } f (f \ 0 \ a) \ r$
- ▶ ce qui donne $(\dots((0+e1) +e2) \dots +en)$
- ▶ Est récursive terminale
- ▶ Privilégier donc l'itération gauche

Filtrage par motif

- ▶ En anglais : **pattern matching**
- ▶ Utilité:
 - ▶ faire une distinction de cas
 - ▶ déconstruire une donnée et accéder aux composantes
- ▶ Principe générale, pas seulement pour les listes.
- ▶ Forme générale:

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

Distinguer des cas I

```
let dist l = match l with  
| []      -> "vide"  
| a::r    -> "pas_vide";;  
# val dist : 'a list -> string = <fun>
```

```
let dist l = match l with  
| []      -> "vide"  
| [a]     -> "un_element"  
| a::b::r -> "plus_d'un_element";;  
# val dist : 'a list -> string = <fun>
```

Distinguer des cas II

```
let dist l = match l with  
| []          -> "vide"  
| a::[]       -> "un_element"  
| a::b::[]    -> "deux_elements"  
| a::b::c::[] -> "trois_elements"  
| a::b::c::r  -> "plus_de_trois_elements";;
```

```
# val dist : 'a list -> string = <fun>
```

```
dist[2;3];;
```

```
# - : string = "deux_elements"
```

```
dist[2;3;6;8;3];;
```

```
# - : string = "plus_de_trois_elements"
```

Distinguer des cas III

```
(* inutile d' utiliser des noms*)  
let dist l = match l with  
| []          -> "vide"  
| _::[]       -> "un_element"  
| _::_::[]    -> "deux_elements"  
| _::_::_    -> "plus_de_deux_elements";;  
# val dist : 'a list -> string = <fun>
```

Définition de head et tail I

```
let tete l = match l with  
| [] -> failwith "Liste_vide"      (* exception *)  
| a::_ -> a;;  
# val tete : 'a list -> 'a = <fun>
```

```
let queue l = match l with  
| [] -> failwith "Liste_vide"      (* exception *)  
| _::r -> r;;  
# val queue : 'a list -> 'a list = <fun>
```


Exemple erroné I

```
let longueur l = match l with  
| []      -> 0  
| a::r    -> 1 + (longueur r);;  
| a::r    -> 1 + (longueur r);;  
                ~~~~~
```

Error: Unbound **value** longueur

Hint: If this is a recursive definition,
you should add the '**rec**' keyword on line 1

Les motifs

- ▶ Un *motif* est construit seulement d'identificateurs et de constructeurs.
- ▶ Si un motif s'applique, *tous* les identificateurs dans le motif sont liés. Leur portée est l'expression à droite du motif.
- ▶ Dans un motif on peut écrire `_` ou un identificateur préfixé par `_`, dans ce cas il n'y a pas de liaison.
- ▶ Les motifs doivent être *linéaires* (pas de répétition d'identificateur dans le même motif)

Filtrage par motif

- ▶ Les motifs sont essayés dans l'ordre donné.
- ▶ Le système OCaml vérifie qu'aucun cas n'a été oublié : l'ensemble des motifs doit être *exhaustif*.
- ▶ La non-exhaustivité donne lieu à un *warning*.
- ▶ Il est fortement conseillé de faire des distinctions de cas exhaustifs.

Imprimer une liste avec séparateurs I

```
let rec imprimer_liste l = match l with
| []          -> print_newline()
| e::[]       -> print_int e
| e::r        -> print_int e;
                print_string "@@";
                imprimer_liste (r);;
# val imprimer_liste : int list -> unit = <fun>

imprimer_liste [1];;
# 1- : unit = ()

imprimer_liste [1;2;3];;
# 1@@2@@3- : unit = ()
```

Longueur impaire I

```
let rec longueur_impair l =  
  match l with  
  | []      -> false  
  | _::[]   -> true  
  | _::_::r -> longueur_impair r;;  
# val longueur_impair : 'a list -> bool = <fun>
```

```
longueur_impair [1;2;3];;  
# - : bool = true  
longueur_impair [1;2;3;4];;  
# - : bool = false
```

Advertissement: motifs non exhaustifs I

```
let g l = match l with
  | [] -> print_string "vide"
  | [h] -> print_string "non-vide";;
```

Lines 1-3, characters 10-34:

```
1 | .....match l with
2 |   | [] -> print_string "vide"
3 |   | [h] -> print_string "non-vide"..
```

Warning 8 [partial-match]: this pattern-matching is **not** exhaustive.
Here is an example **of** a case that is **not** matched:

```
_::_::_
```

```
val g : 'a list -> unit = <fun>
```

```
g ['a'];;
```

```
# non-vide- : unit = ()
```

Advertissement: motifs non exhaustifs II

```
g ['a'; 'b'];;
```

```
# Exception: Match_failure ("//toplevel//", 1, 10).
```

```
let h l = match l with
```

```
  | [] -> print_string "vide"
```

```
  | [_] -> print_string "un_element"
```

```
  | _ -> print_string "plus_d'un_element";;
```

```
# val h : 'a list -> unit = <fun>
```

```
h ['a'];;
```

```
# un_element- : unit = ()
```

```
h ['a'; 'b'];;
```

```
# plus d'un_element- : unit = ()
```

Erreur: non linéarité I

```
let f l = match l with
| x::x::r -> 1
| _        -> 0;;
| x::x::r -> 1
    ^
```

Error: Variable x is bound several times **in** this matching

Erreur: mauvais motif !

```
let f l = match l with  
  | (1+2)::r -> 3  
  | _          -> 5;;  
  | (1+2)::r -> 3  
    ^
```

Error: Syntax error: '))' expected

Line 2, characters 4-5:

```
2 |   | (1+2)::r -> 3  
    ^
```

This '(' might be unmatched

Advertissement: motifs mal ordonnés I

```
let f l = match l with
| []    -> print_string "vide"
| h::r  -> print_int h
| h:[]  -> print_int 3;;      (* jamais atteint *);;
| h:[]  -> print_int 3;;      (* jamais atteint *);;
  ^
```

Error: Syntax error

Attention aux noms des variables I

```
let rec trouve x l = match l with  
  | [] -> false  
  | x::_ -> true  
  | b::r -> trouve x r;;  
  | b::r -> trouve x r;;  
  ^^^^
```

Warning 11 [redundant-case]: this **match** case is unused.

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

```
trouve 1 [1;2;3];;
```

```
# - : bool = true
```

```
trouve 42 [1;2;3];;
```

```
# - : bool = true
```

Chercher un élément dans une liste I

```
let rec chercher a l = match l with  
  | []      -> false  
  | b::r -> if a=b then true else chercher a r;;  
# val chercher : 'a -> 'a list -> bool = <fun>
```

```
chercher 1 [1;2;3];;  
# - : bool = true  
chercher 42 [1;2;3];;  
# - : bool = false
```

Concaténation de deux listes I

```
let rec append l1 l2 = match l1 with  
  | []      -> l2  
  | h1::r1 -> h1 :: (append r1 l2);;
```

```
# val append : 'a list -> 'a list -> 'a list = <fun>
```

```
append [1;2;3;4] [5;6;7;8];;
```

```
# - : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

```
append [1;2;3;4] [];;
```

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

Chercher une sous-liste commençant par un élément l

```
let rec trouve_sous_liste a l = match l with  
  | [] -> []  
  | p::r -> if p = a then l else trouve_sous_liste a r;;  
# val trouve_sous_liste : 'a -> 'a list -> 'a list = <fun>
```

```
trouve_sous_liste 3 [5;6;7;8;9];;  
# - : int list = []  
trouve_sous_liste 3 [5;6;3;7;8;9];;  
# - : int list = [3; 7; 8; 9]
```

Chercher une sous-liste commençant par un élément II

```
(* Version qui ne passe pas l'argument a chaque appel *)  
let trouve_sous_liste_bis a l =  
  let rec aux l = match l with  
    | []      -> []  
    | p::r    -> if p = a then l else aux r  
  in aux l;;  
# val trouve_sous_liste_bis : 'a -> 'a list -> 'a list =  
  <fun>
```

```
trouve_sous_liste_bis 3 [5;6;7;8;9];;  
# - : int list = []  
trouve_sous_liste_bis 3 [5;6;3;7;8;9];;  
# - : int list = [3; 7; 8; 9]
```

Plus sur le filtrage par motif

- ▶ S'applique à n'importe quel type (sauf fonctions et objets)
- ▶ Motifs avec des alternatives
- ▶ Motifs avec des conditions

La fonction de Fibonacci I

```
let rec fib n = match n with  
  | 0 -> 0  
  | 1 -> 1  
  | _ -> fib (n-1) + fib (n-2);;  
# val fib : int -> int = <fun>
```

```
fib 10;;  
# - : int = 55
```

Même fonction mais en version compacte I

```
let rec fib n = match n with  
  | 0 | 1 -> n  
  | _      -> fib (n-1) + fib (n-2);;  
# val fib : int -> int = <fun>
```

```
fib 10;;  
# - : int = 55
```

Motifs avec conditions I

```
(* la fonction trouve avec when *)  
let rec trouve a l = match l with  
| []           -> false  
| b::_ when b=a -> true  
| _::r         -> trouve a r ;;  
# val trouve : 'a -> 'a list -> bool = <fun>
```

```
trouve 1 [1;2;3];;  
# - : bool = true  
trouve 42 [1;2;3];;  
# - : bool = false
```