

Programmation Fonctionnelle



Emmanuel Chailloux

3 juillet 2023

Plan du cours 2:

- ▶ les listes
- ▶ définitions de types
 - ▶ abréviations
 - ▶ enregistrements
 - ▶ union discriminante
 - ▶ types paramétrés
 - ▶ types récurifs
- ▶ labels, bibliothèques labellisée
- ▶ exceptions

Listes homogènes

- ▶ liste vide : []
 - ▶ constructeur ::
 - ▶ type paramétré 'a list
 - ▶ accesseurs List.hd et List.tl
-

```
1 # [] ;;
2 - : 'a list
3 # 1::(2::(3::[])) ;;
4 - : int list
5 # [1; 2; 3;] ;;
6 - : int list
7 # [1; "hello"; 3] ;;
8 erreur de typage
9 # List.hd [1.1; 1.2; 1.3] ;;
10 - : float = 1.1
11 # List.hd [] ;;
12 Exception: Failure "hd".
```

Autres fonctions dans le module List : length, mem, append, map, nth.

length : comptage des éléments d'une liste (1)

fonctionne pour toutes les listes ('a list), 2 cas :

- ▶ la liste [] ne contient aucun élément ;
- ▶ la liste x::xs contient un élément de plus que la liste xs.

La fonction length satisfait donc les deux équations :

$$\begin{cases} (\text{length } []) = 0 \\ (\text{length } (x :: xs)) = 1 + (\text{length } xs) \end{cases}$$

```
1 # let rec length (l : 'a list) : int =
2   if l = [] then 0
3   else 1 + length (List.tl l);;
4 val length : 'a list -> int = <fun>
5
6 # length [2;4;7];;
7 - : int = 3
8 # length [];;
9 - : int = 0
10 # length ['A'; 'G'];;
11 - : int = 2
```

```
1 let rec length (l : 'a list) : int =
2   match l with
3   | [] -> 0
4   | x::xs -> 1 + (length xs) ;;
5 val length : 'a list -> int
6 # length [2;4;7];;
7 - : int = 3
8 # length [];;
9 - : int = 0
10 # length ['A'; 'G'];;
11 - : int = 2
```

mem : appartenance d'un élément à une liste (1)

fonctionne pour toutes les listes ('a list)

3 cas pour $\text{mem } z \ l$:

- ▶ si la liste l est vide, alors z n'appartient pas à la liste ;
- ▶ si la liste l est de la forme $x :: xs$ alors, il y a deux possibilités :
 - ▶ le premier élément de la liste l est égal à z et donc z appartient à $x :: xs$
 - ▶ z appartient à la liste xs

la fonction mem satisfait donc les trois équations suivantes :

$$\left\{ \begin{array}{ll} (\text{mem } z \ []) = \textit{false} & \\ (\text{mem } z (x :: xs)) = \textit{true} & \textit{si } x = z \\ (\text{mem } z (x :: xs)) = (\text{mem } z xs) & \textit{sinon} \end{array} \right.$$

mem : appartenance d'un élément à une liste (2)

Implantation : 2 versions

```
1 # let rec mem (z : 'a) (xs : 'a list) : bool = match xs with
2   | [] -> false
3   | x :: xs -> if (x = z) then true else (mem z xs) ;;
4 val mem : 'a -> 'a list -> bool = <fun>
```

Attention au masquage des noms de paramètres dans les motifs

```
1 # let rec mem (z : 'a) (xs : 'a list) : bool = match xs with
2   | [] -> false
3   | x :: xs -> (x = z) || (mem z xs) ;;
4 val mem : 'a -> 'a list -> bool = <fun>
```

```
1 # mem 3 [1 ; 2 ; 3 ; 4] ;;
2 - : bool = true
3 # mem 3 [1 ; 2 ; 4] ;;
4 - : bool = false
5 # mem true [] ;;
6 - : bool = false
7 # mem true [false] ;;
8 - : bool = false
9 # mem [true] [[false]; [true; false]] ;;
10 - : bool = false
```

append : concaténation de deux listes (1)

fonctionne pour deux listes de même type :

polymorphisme et partage (la seconde liste n'est pas copiée)

La concaténation de deux listes en crée une troisième en suivant les schémas d'équations suivants :

$$\begin{aligned}(\text{append } [] [y_1; \dots; y_m]) &= [y_1; \dots; y_m] \\(\text{append } [x_1; \dots; x_n] [y_1; \dots; y_m]) &= [x_1; \dots; x_n; y_1; \dots; y_m]\end{aligned}$$

On peut noter que :

$$(\text{append } (x_1 :: [x_2; \dots; x_n]) [y_1; \dots; y_m]) = x_1 :: [x_2; \dots; x_n; y_1; \dots; y_m]$$

d'où la définition de la concaténation apr les deux équations suivantes :

$$\begin{cases} (\text{append } [] ys) &= ys \\ (\text{append } (x :: zs) ys) &= x :: (\text{append } zs ys) \end{cases}$$

append : concaténation de deux listes (2)

Implantation :

```
1 # let rec append (xs : 'a list) (ys : 'a list) : ('a list) = match xs with
2 | [] -> ys
3 | x :: zs -> x :: (append zs ys) ;;
4 val append : 'a list -> 'a list -> 'a list = <fun>
5 # append [1 ; 2] [3 ; 4] ;;
6 - : int list = [1; 2; 3; 4]
```

Exemple (schématisque) d'application :

$$\begin{aligned} & (\text{append}[x1; x2; x3][y1; y2; y3; y4]) \\ &= x1 :: (\text{append}[x2; x3][y1; y2; y3; y4]) \\ &= x1 :: x2 :: (\text{append}[x3][y1; y2; y3; y4]) \\ &= x1 :: x2 :: x3 :: (\text{append}[][y1; y2; y3; y4]) \\ &= x1 :: x2 :: x3 :: [y1; y2; y3; y4] \end{aligned}$$

à noter qu'en OCaml la fonction List.append se note aussi par le symbole infixe @

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


map : application d'une fonction sur les éléments d'une liste

$$\text{map } f [x_1 ; \dots ; x_n] = [f x_1 ; \dots ; f x_n]$$

```
1 # let rec map f l =
2   if l = [] then []
3   else
4     let t = List.hd l
5     and q = List.tl l in
6     (f t) :: (map f q) ;;
7 val map : ('a -> 'b) ->
8   'a list ->
9   'b list = <fun>
```

```
1 # let rec map f l =
2   match l with
3   | [] -> []
4   | t::q -> (f t) :: map f q ;;
5 val map : ('a -> 'b) ->
6   'a list ->
7   'b list = <fun>
```

```
1 # let add2 x = x + 2 ;;
2 val add2 : int -> int = <fun>
3 # let l1 = map add2 [7; 2; 9] ;;
4 val l1 : int list = [9; 4; 11]
5
6 # let l2 = map length
7   [['z'; 'a'; 'm']; ['.'; 'n'; 'e'; 't']] ;;
8 val l2 : int list = [3; 4]
```

```
1 # map (append [10; 20])
2   [[3;4;5]; [8; 3; 1]] ;;
3 - : int list list =
4   [[10; 20; 3; 4; 5];
5    [10; 20; 8; 3; 1]]
```

nth : recherche du n-ième élément d'une liste (1)

fonction partielle :

l'appel `nth [x0 ; x1 ; ... ; xn] i` retourne la valeur x_i si $0 \leq i \leq n$.

nth doit satisfaire les équations conditionnelles suivantes :

$$\begin{cases} (\text{nth } (x :: xs) i) = x \text{ si } i = 0 \\ (\text{nth } (x :: xs) i) = (\text{nth } xs (i - 1)) \text{ sinon} \end{cases}$$

Attention : si l'indice i n'est pas dans l'intervalle $[0, (\text{length } l) - 1]$.

nth : recherche du n-ième élément d'une liste (2)

Implantation:

```
1 # let rec nth (xs : 'a list) (i:int) : 'a =
2   match xs with
3     | [] -> raise (Failure "nth")
4     | x::xs -> if (i=0) then x else (nth xs (i-1))    ;;
5 val nth : 'a list -> int -> 'a = <fun>
6
7 # nth ['a' ; 'm' ; 'l' ] 2;;
8 - : char = 'l'
9 # nth ['a' ; 'm' ; 'l' ] 3;;
10 Exception: Failure "nth".
```

Fonctions sur les listes - module List (1)

2 constructeurs (infixes) :

- ▶ [] : liste vide
- ▶ :: (cons) : liste non vide

```
1 let rec length_aux len l = match l with
2   [] -> len
3   | a::l -> length_aux (len + 1) l
4
5 let length l = length_aux 0 l
6
7 let rec rev_append l1 l2 =
8   match l1 with
9     [] -> l2
10    | a :: l -> rev_append l (a :: l2)
11
12 let rev l = rev_append l []
13
14 (*
15   val length_aux : int -> 'a list -> int = <fun>
16   val length : 'a list -> int = <fun>
17   val rev_append : 'a list -> 'a list -> 'a list = <fun>
18   val rev : 'a list -> 'a list = <fun>
19 *)
```

Fonctions sur les listes - module List (2)

```
1  let rec map f = function
2      [] -> []
3      | a::l -> let r = f a in r :: map f l
4
5  let rev_map f l =
6      let rec rmap_f accu = function
7          | [] -> accu
8          | a::l -> rmap_f (f a :: accu) l
9      in
10     rmap_f [] l
11 ;;
12
13 (*
14     val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
15     val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
16 *)
```

Fonctions sur les listes - module List (3)

voir cours 3 sur les itérateurs

```
1 let rec fold_left f accu l = (* fold_left f r [e1;e2;e3]=f(f(f r e1)e2)e3 *)
2   match l with
3     [] -> accu
4     | a::l -> fold_left f (f accu a) l
5
6 (* fold_right f [e1;e2;e3] r=f e1(f e2( fe3 r)) *)
7 let rec fold_right f l accu = match l with
8   [] -> accu
9   | a::l -> f a (fold_right f l accu)
10
11 (* val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
12    val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun> *)
```

```
1 List.fold_left (+) 0 [8;4;10];;          (* (+) 1 2 donne 3 *)
2 (* - : int = 22 *)
3 List.fold_right (+) [8; 4; 10] 0;;
4 (* - : int = 22 *)
5 List.fold_left (/) 0 [8;4;10];;
6 (* - : int = 0 *)
7 List.fold_right (/) [8;4;10] 0;;
8 (* Exception: Division_by_zero. *)
```

Déclarations de types en OCaml (1)

- ▶ produit cartésien : enregistrement

Syntaxe:

```
type nom = enregistrement
```

- ▶ union discriminante : somme avec constructeurs

Syntaxe:

```
type nom = union
```

- ▶ abréviation

Syntaxe:

```
type nom = nom2
```

Déclarations de types en OCaml (2)

- ▶ déclarations combinées

Syntaxe:

```
type nom_1 = ...
```

```
and nom_2 = ...
```

```
and nom_n = ...
```

- ▶ avec paramètres

Syntaxe:

```
type (p1, p2, ..., pn) nom = ...
```


Enregistrements (1)

Syntaxe:

type $t = \{f1 : t1; f2 : t2; \dots; fn : tn\}$

▶ constructeur : { ... }

```
1 # type complex = {re:float;im:float} ;;
2 type complex = { re: float; im: float}
3
4 # let c = {re=2.;im=3.};;
5 val c : complex = {re=2; im=3}
6
7 # let mult_complex c1 c2 =
8     match (c1,c2) with
9         ({re=x1;im=y1}, {re=x2;im=y2}) ->
10             {re=x1*.x2-.y1*.y2;im=x1*.y2+.x2*.y1};;
11 val mult_complex :
12     complex -> complex -> complex = <fun>
13
14 # mult_complex c c;;
15 - : complex = {re=-5; im=12}
```

Enregistrements (2)

- ▶ constructeur : { ... }
 - ▶ accesseurs : .re et .im
-

```
1 # let add_complex c1 c2 =  
2   {re=c1.re+c2.re; im=c1.im+c2.im};;  
3 val add_complex :  
4   complex -> complex -> complex = <fun>  
5  
6 # add_complex c c;;  
7 - : complex = {re=4; im=6}
```

Unions discriminantes

sommes avec constructeurs:

Syntaxe:

$$\text{type } t = C1 \mid C2 \text{ of } t1 \mid \dots \mid Cm \text{ of } t2 \mid Cn$$

Constructeurs constants:

```
1 # type piece = Pile | Face;;
2 type piece = | Pile | Face
3
4 # Pile;;
5 - : piece = Pile
6
7 # [Pile; Face; Face; Pile];;
8 - : piece list = [Pile; Face; Face; Pile]
```

Tous les constructeurs commencent par une majuscule. Cela permet de les distinguer des variables dans le filtrage par motifs.

Constructeurs avec paramètres

```
1 # type couleur = Pique | Coeur | Carreau | Trefle;;
2 type couleur = | Pique | Coeur | Carreau | Trefle
3 # type carte = As of couleur
4     | Roi of couleur
5     | Dame of couleur
6     | Valet of couleur
7     | Autre of couleur * int ;;
8     type carte = ...
9 # (As Pique, Autre(Coeur,9)) ;;
10 - : carte * carte : ...
11 # let valeur couleur_atout cr = match cr with
12     | As _ -> 11
13     | Roi _ -> 4
14     | Dame _ -> 3
15     | Valet c -> if c = couleur_atout then 20 else 2
16     | Autre (_,10) -> 10
17     | Autre (c,9) -> if c = couleur_atout then 14 else 0
18     | _ -> 0 ;;
19 val valeur : couleur -> carte -> int = <fun>
20 # valeur Pique (Autre(Coeur,9));;
21 - : int = 0
22 # valeur Coeur (Autre(Coeur,9));;
23 - : int = 14
```

Les déclarations de types sont récurives

```
1 # type intPile = IntVide
2     | IntPleine of int * intPile ;;
3 type intPile = IntVide | IntPleine of int * intPile
4
5 # let ma_pile = IntPleine ( 33, IntPleine (44, IntPleine (55, IntVide))) ;;
6 val ma_pile : intPile =
7     IntPleine (33, IntPleine (44, IntPleine (55, IntVide)))
8
9 # let rec nb_elt p = match p with
10     | IntVide -> 0
11     | IntPleine (_, ps) -> 1 + (nb_elt ps) ;;
12 val nb_elt : intPile -> int = <fun>
13
14 # nb_elt ma_pile ;;
15 - : int = 3
```

Types paramétrés

Les déclarations de types peuvent être paramétrées

```
1 type 'a list = [] | (::) of 'a * 'a list
```

pires homogènes :

```
1 # type 'a pile = Vide | Pleine of 'a * 'a pile;;
2 type 'a pile = Vide | Pleine of 'a * 'a pile
3 # let empiler x p = Pleine(x, p) ;;
4 val empiler : 'a -> 'a pile -> 'a pile = <fun>
5 # let depiler p = match p with Pleine(_,p') -> p' ;;
6 Warning 8: this pattern-matching is not exhaustive.
7 Here is an example of a value that is not matched:
8 Vide
9 val depiler : 'a pile -> 'a pile = <fun>
10 # let sommet p = match p with Pleine(x,_) -> x
11 ;;
12 Warning 8: this pattern-matching is not exhaustive.
13 Here is an example of a value that is not matched:
14 Vide
15 val sommet : 'a pile -> 'a = <fun>
```

même structure que les listes mais opérations différentes.

Exemple : type option

```
1 # type 'a option = None
2     | Some of 'a;;
3
4 # let x = Some Pique;;
5 val x : couleur option = Some Pique
6
7 # let y = None;;
8 val y : 'a option = None
9
10 # let create_as oc = match oc with
11     None -> As Pique
12     | Some coul -> As coul;;
13 val create_as : couleur option -> carte
14
15 # create_as None;;
16 - : carte = As Pique
17
18 # create_as (Some Coeur);;
19 - : carte = As Coeur
```

Types fonctionnels

```
1 type 'a listf =
2   Val of 'a
3   | Fun of ('a -> 'a) * 'a listf ;;
4 (* type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf *)
5
6 let huit_div = (/) 8 ;;
7 (* val huit_div : int -> int = <fun> *)
8
9 let gl = Fun (succ, (Fun (huit_div, Val 4))) ;;
10 (* val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))*
11
12 let rec compute = function
13   Val v -> v
14   | Fun(f, x) -> f (compute x) ;;
15 (* val compute : 'a listf -> 'a = <fun> *)
16 compute gl;;
17 (* - : int = 3 *)
```


Labels (1)

Label :

*annotation portée aux paramètres (formels ou d'appel)
d'une fonction*

Syntaxe :

▶ let

Syntaxe:

let fonction ~label:p = exp

▶ fun

Syntaxe:

fun ~label:p -> exp

Labels (2)

dans les paramètres d'appel d'une fonction :

Syntaxe:

(*fonction* ~*label:exp*)

dans les types :

```
1 # let add ~op1:x ~op2:y = x + y;;
2 val add : op1:int -> op2:int -> int = <fun>
3
4 # let mk_triplet ~arg1:x ~arg2:y ~arg3:z = (x,y,z);;
5 val mk_triplet :
6   arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
7
8 # let mk3 ~arg1:x ~y ~arg3:z = (x,y,z);;
9 val mk3 :
10  arg1:'a -> y:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

Labels (3)

les labels peuvent être décrits pour l'application (ce n'est pas obligatoire) :

```
1 # let mk3 ~arg1:x ~y ~arg3:z = (x,y,z);;
2 val mk3 :
3   arg1:'a -> y:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
4
5 # mk_triplet '1' 2 3.0 ;;
6 - : char * int * float = ('1', 2, 3.)
7
8 # mk_triplet ~arg1:'A' ~arg2:2 ~arg3:3.0 ;;
9 - : char * int * float = ('A', 2, 3.)
```

incluant les cas d'applications partielles :

```
1 # let f = mk_triplet ~arg1:44;;
2 val f : arg2:'_a -> arg3:'_b -> int * '_a * '_b = <fun>
```

Labels (4)

y compris dans un autre ordre :

```
1 # let g = mk_triplet ~arg2:22;;  
2 val g : arg1:'a -> arg3:'b -> 'a * int * 'b = <fun>
```

tout en conservant l'évaluation immédiate :

```
1 # let g =  
2   mk_triplet ~arg2:(print_int 44; print_newline());44);;  
3 44  
4 val g : arg1:'a -> arg3:'b -> 'a * int * 'b = <fun>
```

Conventions

sur les noms de label

label signification

pos: une position (dans une liste, une chaîne, un tableau, etc.)

len: une taille (length)

buf: une chaîne utilisée comme tampon (buffer)

src: la source d'une opération

dst: la destination d'une opération

fun: une fonction à appliquer

pred: un prédicat

acc: un accumulateur

out: un canal de sortie (out_channel)

key: une clé utilisée dans un index (liste d'associations, etc.)

data: une valeur associée utilisée dans un index

mode: un mode ou une liste d'options

perm: des permissions de fichiers

plusieurs bibliothèques dont `ListLabels` peuvent utiliser des labels dont `List`

ListLabels:

```
val map : f:( '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.

Typage et domaine de définition

type inféré \neq domaine de définition:

- ▶ c'est une approximation
- ▶ exemple : division entière, tête de liste vide
- ▶ provient souvent d'un filtrage non exhaustif

Que faire ?:

- ▶ utiliser une valeur spéciale (ou plusieurs)

```
1 # asin 2.;;  
2 - : float =      NaN
```

- ▶ effectuer une rupture de calcul jusqu'à un récupérateur d'une telle rupture
⇒ exceptions

Exceptions

Une exception est une rupture de calcul.
utilisée :

- ▶ pour éviter les erreurs de calcul
 - ▶ division par zéro
 - ▶ accès à la référence `null`
 - ▶ ouverture d'un fichier inexistant
 - ▶ ...
- ▶ comme style de programmation
 - ▶ sortie de boucles
 - ▶ remontée directe d'appels imbriqués
 - ▶ ...

En OCaml une exception est une valeur de type `exn`

Exceptions

Syntaxe:

```
exception E1 ;;
```

```
exception E1 of t1 ;;
```

- ▶ une exception est une valeur de type *exn*
 - ▶ le type *exn* est un type somme monomorphe **extensible**
-

```
1 # exception A_MOI;;  
2 exception A_MOI  
3  
4 # A_MOI;;  
5 - : exn = A_MOI  
6  
7 # exception Depth of int;;  
8 exception Depth of int  
9  
10 # Depth 4;;  
11 - : exn = Depth(4)
```

Déclenchement d'une exception

`raise : exn -> 'a`

- ▶ impossible à écrire \Rightarrow primitive
 - ▶ l'expression `(raise E1)` n'a pas de contrainte de type car elle ne calcule pas de valeur
-

```
1 # raise A_MOI ;;
2 Exception: A_MOI
3
4 # let x = 18 ;;
5 val x : int = 18
6
7 # if (x = 0) then raise A_MOI else x ;;
8 - : int = 18
9 # if (x = 18) then raise A_MOI else x ;;
10 Exception: A_MOI.
```

Déclarations et déclenchements (1)

```
1 # exception Echec of string;;
2 exception Echec of string
3
4 # let declenche_echec s = raise (Echec s);;
5 val declenche_echec : string -> 'a = <fun>
6
7 # declenche_echec "argument invalide";;
8 Exception: Echec "argument invalide".
```

la fonction failwith s'écrit :

```
1 let failwith s = raise (Failure s);;
2
3 let invalid_argument s =
4   raise (Invalid_argument s);;
```

Failure et Invalid_argument sont prédéfinies.

Déclarations et déclenchements (2)

```
1 # exception OrthoExn of int * int * string;;
2 exception OrthoExn of int * int * string
3
4 # raise (OrthoExn (3, 6, "le caml"));
5 Exception: OrthoExn (3, 6, "le caml").
6
7 # exception FuncTreat of (int -> int);;
8 exception FuncTreat of (int -> int)
9
10 # raise (FuncTreat (fun x -> x + 1));;
11 Exception: FuncTreat <fun>.
```

Déclarations et déclenchements (3)

Filtrage de motifs incomplet:

```
1 # let tete l = match l with t::q -> t;;
2 Warning: this pattern-matching is not exhaustive.
3 Here is an example of a value that is not matched:
4 []
5 val tete : 'a list -> 'a = <fun>
6
7 # tete [1;2;3];;
8 - : int = 1
9
10 # tete [];;
11 Exception: Match_failure ("", 13, 35).
```

```
1 # let tete l =
2     match l with t::q -> t
3         | _ -> raise (Match_failure (" ",13,35)) ;;
4 val tete : 'a list -> 'a = <fun>
```

Déclarations et déclenchements (4)

```
1 # exception Found_zero;;
2 exception Found_zero
3
4 # let rec mult_aux l= match l with
5     h::[] -> h
6     | 0::t -> raise Found_zero
7     | h::t -> h * mult_aux t ;;
8 Warning 8: this pattern-matching is not exhaustive.
9 Here is an example of a value that is not matched:
10 []
11 val mult_aux : int list -> int = <fun>
```

Récupération d'exceptions

Syntaxe:

try expr **with** filtrage

Le type des motifs du *filtrage* doit être *exn*.

```
1 # let mult_list l = match l with
2   [] -> 0
3 | lo -> (try mult_aux lo with
4         | Found_zero -> 0
5         | e -> raise e ) ;;
6 val mult_list : int list -> int = <fun>
7
8 # mult_list [1;2;3;0;5;6];;
9 - : int = 0
```

```
m [1;2;3;0;5;6] -> try m_aux [1;2;3;0;5;6]
                    1 * m_aux [2;3;0;5;6]
                    2 * m_aux [3;0;5;6]
                    3 * m_aux[0;5;6]
                    raise Found_zero
                    with Found_zero -> 0
```

Module List (1)

```
1                                     (* forme e'quivalente *)
2 let hd = function                   (* let hd l = match l with *)
3   [] -> failwith "hd"               (* | [] -> failwith "hd" *)
4   | a::l -> a                       (* | a::_ -> a *)
5
6 let tl = function
7   [] -> failwith "tl"
8   | a::l -> l
9
10 let rec nth l n =
11   match l with
12     [] -> failwith "nth"
13     | a::l ->
14       if n = 0 then a else
15       if n > 0 then nth l (n-1) else
16       invalid_arg "List.nth"
```


Module List (2)

```
1
2 #let rec fold_left f accu l =
3   match l with
4     [] -> accu
5     | a::l -> fold_left f (f accu a) l
6 val fold_left :
7   ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
8
9 # let rec fold_right f l accu =
10  match l with
11    [] -> accu
12    | a::l -> f a (fold_right f l accu)
13 val fold_right :
14   ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Exemples fonctionnels

```
1 # fold_left (/) 1000 [3;5;11];;
2 - : int = 6
3
4 # fold_left (/) 1000 [3;0;11];;
5 Exception: Division_by_zero.
6
7 # let idiv a b = b / a;;
8 val idiv : int -> int -> int = <fun>
9
10 # fold_right idiv [3;5;11] 1000;;
11 - : int = 6
12
13 # fold_right idiv [3;0;11] 1000;;
14 Exception: Division_by_zero.
```

filtrage d'une liste:

```
1 # let rec filter (p : 'a -> bool) (xs : 'a list) : ('a list) =
2   match xs with
3   | [] -> []
4   | x::xs -> if (p x) then x::(filter p xs)
5             else (filter p xs)      ;;
6 val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Exemple : filtrage d'une liste

- ▶ filtrage des éléments d'une liste par un prédicat
- ▶ sans recopie inutile

```
1 # exception Identity ;;
2 exception Identity
3 # let share f x = try f x with Identity -> x ;;
4 val share : ('a -> 'a) -> 'a -> 'a = <fun>
5 # let filter f l =
6     let rec fil l = match l with
7     | [] -> raise Identity
8     | h :: t ->
9         if f h then h :: fil t else share fil t in
10    share fil l ;;
11 val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
12
13 # let l = [2; 4; 6 ; 8] ;;
14 val l : int list = [2; 4; 6; 8]
15 # let l2 = filter (function x -> x mod 4 = 0) l;;
16 val l2 : int list = [4; 8]
17 # let l3 = filter (function x -> x mod 2 = 0) l;;
18 val l3 : int list = [2; 4; 6; 8]
19 # l == l3 ;;
20 - : bool = true
```

Utilisation des exceptions

- ▶ Gestion de situations exceptionnelles où le calcul ne peut pas se poursuivre → rupture du calcul
- ▶ style de programmation : exemple précédent (`filter`)

Attention au coût du `try`