Programmation Fonctionnelle (LU2IN019) : COURS 2

Programmation Fonctionnelle

Emmanuel Chailloux

25 juin 2021

Plan

Plan du cours 2:

- les listes
- définitions de types
  - abbréviations
  - enregistrements
  - union discriminante
  - types paramétrés
  - types récursifs
- labels, bibliothèques labellisée
- exceptions

Listes homogènes

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

```
# [] ;;
- : 'a list
# 1::2::3::[] ;;
- : int list
# [1; 2; 3;] ;;
- : int list
# [1; "hello"; 3] ;;
erreur de typage
# List.hd [1.1; 1.2; 1.3] ;;
- : float = 1.1
# List.hd [] ;;
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 :

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

```
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  # length [2;4;7];;
6  - : int = 3
7  # length [];;
8  - : int = 0
9  # length ['A'; 'G'];;
10 - : 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 `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 :

$$\begin{cases} (mem\ z\ []) & = & false & \\ (mem\ z\ (x :: xs)) & = & true & si\ x\ =\ z \\ (mem\ z\ (x :: xs)) & = & (mem\ z\ xs) & sinon \end{cases}$$

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

Implantation: : 2 versions

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

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

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

$$\begin{array}{lcl} (append[][y1;...;ym] & = & [y1;..;ym]) \\ (append[x1;...;xn][y1;...;ym] & = & [x1;...;xn;y1;..;ym]) \end{array}$$

On peut noter que :

$$(append(x1 :: [x2;..;xn])[y1;..;ym]) = x1 :: [x2;..;xn;y1;..;ym]$$

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

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

append : concaténation de deux listes (2)

Implantation: :

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

Exemple (schématique) d'application :

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

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

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

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

$$map\ f\ [x1\ ;\ \ldots\ ;\ xn] \quad = \quad [f\ x1\ ;\ \ldots\ ;\ f\ xn]$$

```
# let rec map f l =
  if l = [] then []
  else
    let t = List.hd l
    and q = List.tl l in
      (f t) :: (map f q) ;;
val map : ('a -> 'b) -> 'a ↩
     list -> 'b list = <↩
     fun>
```

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

```
# let add2 x = x + 2 ;;
val add2 : int -> int = <fun>
# let l1 = map add2 [7; 2; 9] ;;
val l1 : int list = [9; 4; 11]

# let l2 = map length [['z';'a';'m']; ↩
     ['.';'n';'e';'t']] ;;
val l2 : int list = [3; 4]
```

```
#  map (append [10;  20])
       [[3;4;5];  [8; 3; ↩
            1]] ;;
- : int list list =
       [[10; 20; 3; 4; 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 `xi` si $0 \leq i \leq n$.

`nth` doit satisfaire les équations conditionnelles suivantes :

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

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

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

Implantation:

```
# let rec nth (xs : 'a list) (i:int) : 'a =
  match xs with
  | [] -> raise (Failure "nth")
  | x::xs -> if (i=0) then x else (nth xs (i-1))      ;;
val nth : 'a list -> int -> 'a = <fun>

# nth ['a' ; 'm' ;  'l' ] 2;;
- : char = 'l'
# nth ['a' ; 'm' ;  'l' ] 3;;
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

```
let rec fold_left f accu l = (* fold_left f r [e1;e2;e3]=f(f(f r e1)e2)e3 *)
  match l with
    [] -> accu
  | a::l -> fold_left f (f accu a) l

let rec fold_right f l accu = (* fold_right f [e1;e2;e3] r=f e1(f e2( fe3 r))↩
      *)
  match l with
    [] -> accu
    | a::l -> f a (fold_right f l accu)

(* val fold_left :   ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
   val fold_right :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun> *)
```

```
List.fold_left (+) 0 [8;4;10];;
(* - : int = 22 *)
List.fold_right (+) [8; 4; 10] 0;;
(* - : int = 22  *)
List.fold_left (/) 0 [8;4;10];;
(* - : int = 0 *)
List.fold_right (/) [8;4;10] 0;;
(* 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 $=\{f1 : t1 ; f2 : t2 ; \ldots ; fn : tn\}$

- 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`

```
# let add_complex c1 c2 =
   {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex :
   complex -> complex -> complex = <fun>

# add_complex c c;;
- : complex = {re=4; im=6}
```

Unions discriminantes

sommes avec constructeurs:

Syntaxe:

type *t* = C1 | *C2* **of** *t1* | ... | *Cm* **of** *t2* | *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]
```

Constructeurs avec paramètres

```
# type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = | Pique | Coeur | Carreau | Trefle
# type carte = As of couleur
           | Roi of couleur
           | Dame of couleur
           | Valet of couleur
           | Autre of couleur * int ;;
          type carte =...
# (As Pique, Autre(Coeur,9)) ;;
- :  carte * carte : ...
# let valeur couleur_atout cr = match cr with
    | As _ -> 11
    | Roi _ -> 4
    | Dame _ -> 3
    | Valet c -> if c = couleur_atout then 20 else 2
    | Autre (_,10) -> 10
    | Autre (c,9)  -> if c = couleur_atout then 14 else 0
    | _ -> 0 ;;
val valeur : couleur -> carte -> int = <fun>
# valeur Pique (Autre(Coeur,9));;
- : int = 0
# valeur Coeur (Autre(Coeur,9));;
- : int = 14
```

Types récursifs

Les déclarations de types sont récursives

```
# type intPile =  IntVide
                | IntPleine of int * intPile ;;
type intPile = IntVide | IntPleine of int * intPile

# let ma_pile =  IntPleine ( 33, IntPleine (44, IntPleine (55,  IntVide))) ;;
val ma_pile : intPile =
  IntPleine (33, IntPleine (44, IntPleine (55, IntVide)))

# let rec nb_elt p = match p with
  | IntVide -> 0
  | IntPleine (_, ps) -> 1 + (nb_elt ps) ;;
val nb_elt : intPile -> int = <fun>

# nb_elt ma_pile ;;
- : int = 3
```

Types paramétrés

Les déclarations de types peuvent être paramétrées

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

piles homogènes :

```
#  type 'a pile = Vide | Pleine of 'a * 'a pile;;
  type 'a pile = Vide | Pleine of 'a * 'a pile
# let empiler x p = Pleine(x, p) ;;
val empiler : 'a -> 'a pile -> 'a pile = <fun>
# let depiler p = match p with Pleine(_,p') -> p' ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Vide
val depiler : 'a pile -> 'a pile = <fun>
#   let sommet p = match p with Pleine(x,_) -> x
  ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Vide
val sommet : 'a pile -> 'a = <fun>
```

même structure que les listes mais opérations différentes.

Exemple : type option

```
# type 'a option = None
                 | Some of 'a;;

# let x = Some Pique;;
val x : couleur option = Some Pique

# let y = None;;
val y : 'a option = None

# let create_as oc = match oc with
  None -> As Pique
| Some  coul -> As coul;;
val create_as : couleur option -> carte

# create_as None;;
- : carte = As Pique

# create_as (Some Coeur);;
- : 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 :

```
# let add ~op1:x ~op2:y = x + y;;
val add : op1:int -> op2:int -> int = <fun>

# let mk_triplet ~arg1:x ~arg2:y ~arg3:z = (x,y,z);;
val mk_triplet :
  arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>

# let mk3 ~arg1:x ~y ~arg3:z = (x,y,z);;
val mk3 :
  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) :

```
# let mk3 ~arg1:x ~y ~arg3:z = (x,y,z);;
val mk3 :
  arg1:'a -> y:'b -> arg3:'c -> 'a * 'b * 'c = <fun>

# mk_triplet '1' 2 3.0 ;;
- : char * int * float = ('1', 2, 3.)

#  mk_triplet ~arg1:'A' ~arg2:2 ~arg3:3.0 ;;
- : char * int * float = ('A', 2, 3.)
```

incluant les cas d'applications partielles :

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

Labels (4)

y compris dans un autre ordre :

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

tout en conservant l'évaluation immédiate :

```
# let g  =
  mk_triplet ~arg2:(print_int 44; print_newline();444);;
44
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
```

Bibliothèques

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
  $\Rightarrow$ 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 ⇒ 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)

```
# exception Echec of string;;
exception Echec of string

# let declenche_echec s = raise (Echec s);;
val declenche_echec : string -> 'a = <fun>

# declenche_echec "argument invalide";;
Exception: Echec "argument invalide".
```

la fonction `failwith` s'écrit :

```
let failwith s = raise (Failure s);;

let invalid_argument s =
   raise (Invalid_argument s);;
```

`Failure` et `Invalid_argument` sont prédéfinies.

Déclarations et déclenchements (2)

```
# exception OrthoExn  of int * int * string;;
exception OrthoExn of int * int * string

# raise (OrthoExn (3, 6, "le caml"));;
Exception: OrthoExn (3, 6, "le caml").

# exception FuncTreat of (int -> int);;
exception FuncTreat of (int -> int)

# raise (FuncTreat (fun x -> x + 1));;
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).
```

Déclarations et déclenchements (4)

```
# exception Found_zero;;
exception Found_zero

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

Récupération d'exceptions

Syntaxe:

try expr **with** filtrage

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

```
# let mult_list l = match l with
  [] -> 0
| lo ->  (try mult_aux lo with
          | Found_zero -> 0
          | e -> raise e ) ;;
val mult_list : int list -> int = <fun>

# mult_list [1;2;3;0;5;6];;
- : 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

```
# fold_left (/) 1000 [3;5;11];;
- : int = 6

# fold_left (/) 1000 [3;0;11];;
Exception: Division_by_zero.

# let idiv a b = b / a;;
val idiv : int -> int -> int = <fun>

# fold_right idiv [3;5;11]  1000;;
- : int = 6

# fold_right idiv  [3;0;11] 1000;;
Exception: Division_by_zero.
```

Exemple : filtrage d'une liste

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

```
# exception Identity ;;
exception Identity
# let share f x = try f x with Identity -> x ;;
val share : ('a -> 'a) -> 'a -> 'a = <fun>
# let filter f l =
   let rec fil l = match l with
   | [] -> raise Identity
   | h :: t ->
      if f h then h :: fil t else share fil t in
 share fil l ;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>

# let l = [2; 4; 6 ; 8] ;;
val l : int list = [2; 4; 6; 8]
# let l2 = filter (function x -> x mod 4 = 0) l;;
val l2 : int list = [4; 8]
# let l3 = filter (function x -> x mod 2 = 0) l;;
val l3 : int list = [2; 4; 6; 8]
# l == l3 ;;
- : bool = true
```

Utilisation des exceptions

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

Attention au coût du `try`