

Mise à Niveau et Ouverture (STL)



Emmanuel Chailloux

Un mot sur OCaml (1)

- ▶ développé à l'Inria depuis 30 ans (ocaml.org)
- ▶ Langage de programmation **multi-paradigmes** (fonctionnel, impératif, objet, modulaire), de haut niveau avec des constructions **riches** et une **expressivité** accrue (foncteurs, types algébriques, lambdas, objets, exceptions ...)
- ▶ **Sûreté** augmentée par le typage statique avec inférence

```
1 # let rec map(f,l) = match l with
2   [] -> []
3   | t::q -> let nt = f(t) in
4             let nq = map(f,q) in
5             nt :: nq ;;
6 val map : ('a -> 'b) * 'a list -> 'b list = <fun>
7
8 # let compose(f,g,x) = f(g(x));;
9 val compose : ('a -> 'b) * ('c -> 'a) * 'c -> 'b = <fun>
10
11 # let addNL o = o#toString() ^ "\n";;
12 val addNL : < toString : unit -> string; .. > -> string = <fun>
```

Un mot sur OCaml (2)

- ▶ Ayant une réelle influence : F#, Swift, ReasonML
- ▶ des applications phares sur la fiabilité logicielle, l'implantation de langages, mais aussi en système, web et finance
- ▶ une communauté active, avec gestionnaire de paquets (opam)
- ▶ des compilateurs natifs et une machine virtuelle **légère**

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

Logiciels du cours

- ▶ Langages
 - ▶ OCaml 4.13 (Inria)
 - ▶ pré-installé à la PPTI
 - ▶ à installer à la maison à partir du site <http://ocaml.org>
 - ▶ dans un navigateur avec TryOCaml : <http://try.ocamlpro.com/>
 - ▶ F# 4.xx (Microsoft) <http://fsharp.org/>
 - ▶ Swift 4.xx (Apple) : <https://swift.org/>
 - ▶ Reason (Facebook) : <https://reasonml.github.io/>
- ▶ Environnements de développement
 - ▶ mode Tuareg pour Emacs : <http://tuareg.forge.ocamlcore.org/>
 - ▶ Emacs et Merlin : <https://github.com/ocaml/merlin>
 - ▶ plug-in Ocaide pour Eclipse : <http://www.algo-prog.info/ocaide/>
 - ▶ VisualStudioCode : <https://code.visualstudio.com>
 - ▶ Atom script : <https://atom.io/packages/script>
 - ▶ Xamarin : <https://xamarin.com/studio>
 - ▶ Netbean : <http://ocamlplugin.loki-a.com/index.php>
- ▶ Environnement pédagogique
 - ▶ learn-ocaml

Bibliographie sur la partie programmation (1)

▶ livres sur OCaml

- ▶ Xavier Leroy et al. The OCaml system : documentation and user's manual Inria, 2018.
- ▶ Emmanuel Chailloux, Pascal Manoury et Bruno Pagano. Développement d'Applications avec Objective Caml. O'Reilly, 2000.
- ▶ Guy Cousineau et Michel Mauny. Approche fonctionnelle de la programmation. Dunod, 1995.
- ▶ Philippe Nardel. Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml. Vuibert, 2005
- ▶ Pascal Manoury. Programmation de droite à gauche, et vice-versa. Paracampus, 2012.
- ▶ Sylvain Conchon et Jean-Christophe Filliâtre. Apprendre à programmer avec OCaml. Eyrolles, 2013.
- ▶ Yaron Minsky, Anil Madhavapeddy, Jason Hickey - Real World OCaml - O'Reilly - 2013

Bibliographie sur la partie programmation (2)

- ▶ cartes de référence OcamlPro :
 - ▶ langage :
<http://www.ocamlpro.com/wp-content/uploads/2019/09/ocaml-lang.pdf>
 - ▶ bibliothèque :
<http://www.ocamlpro.com/wp-content/uploads/2019/09/ocaml-stdlib.pdf>
- ▶ cours :
 - ▶ programmation fonctionnelle (LU2IN019) :
<http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2019/ue/LU2IN019-2019oct/>
 - ▶ modèles de programmation et interopérabilité des langages (LU3IN008) :
<http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I008-2019fev/>
- ▶ MOOC OCAML :
<https://www.fun-mooc.fr/courses/course-v1:parisdiderot+56002+session04/about>
- ▶ autres références sur : <https://ocaml.org/learn/books.html>,
http://ocaml.org/docs/cheat_sheets.html,

Programmes en Objective Caml (ou F#)

Seuls les programmes **correctement typés** peuvent être exécutés!!!



un programme est :

- ▶ le calcul d'une expression
- ▶ c'est-à-dire l'application d'une fonction à ses arguments
- ▶ avec évaluation immédiate des arguments
- ▶ et rupture d'évaluation si calcul impossible

Types de base, valeurs et fonctions

- ▶ nombres
 - ▶ *int* ($[-2^{30}, 2^{30} - 1]$ sur 32 bits ou $[-2^{62}, 2^{62} - 1]$ sur 64 bits)
voir `min_int` et `max_int`
 - ▶ *float* (IEEE 754) mantisse 53bits, exposant $\in [-1022, 1023]$
- ▶ caractères : *char*
- ▶ chaînes de caractères : *string*
- ▶ booléens : *bool*

Opérations sur les nombres

entiers		flottants	
+	addition	+.	addition
-	soustraction	-.	soustraction
*	multiplication	*.	multiplication
/	division entière	/.	division
mod	modulo	**	exponentiation

opérateurs infixes !!!

```
1 # 1 + 3;;  
2 - : int = 4  
3 # 1.8 *. 9.1;;  
4 - : float = 16.38
```

Fonctions sur les nombres

ceil		cos	cosinus
floor		sin	sinus
sqrt	racine carrée	tan	tangente
exp	exponentielle	acos	arccosinus
log	log népérien	asin	arcsinus
log10	log en base 10	atan	arctangente

angles en radian!!!

Calculs sur les nombres (1)

```
1 # 1 + 2;;  
2 - : int = 3  
3  
4 # 9/2;;  
5 - : int = 4  
6  
7 # max_int + 1;;  
8 - : int = -4611686018427387904  
9  
10 # 9.1 /. 2.2;;  
11 - : float = 4.13636363636  
12  
13 # 1. /. 0.;;  
14 - : float =          Inf
```

Calculs sur les nombres (2)

```
1 # 2 + 2.1;;
2 This expression has type float but
3 is here used with type int
```

Un *int* ne peut pas remplacer un *float*!!!

```
1 # sin;;
2 - : float -> float = <fun>
3
4 # asin 1.;;
5 - : float = 1.57079632679
```

Calculs sur les chaînes

```
1 # 'B';;  
2 - : char = 'B'  
3  
4 # int_of_char 'B';;  
5 - : int = 66  
6  
7 # "est une chaîne";;  
8 - : string = "est une chaîne"  
9  
10 # (string_of_int 1987)^" est l'année de CAML";;  
11 - : string = "1987 est l'année de CAML"
```

Opérations sur les booléens

Type des constantes et opérateurs:

true : bool

false : bool

not : bool -> bool

&& : bool -> bool -> bool

|| : bool -> bool -> bool

si e_1 et e_2 sont des expressions booléennes alors true, false, not e_1 , (e_1 || e_2), (e_1 && e_2) sont des expressions booléennes et réciproquement.

```
1 # true && false ;;
2 - : bool = false
3 # not (true && false) ;;
4 - : bool = true
```

Relations d'égalité et d'inégalité

=	égalité structurelle	<	inférieur
==	égalité physique	>	supérieur
<>	négation de =	<=	inférieur ou égal
!=	négation de ==	>=	supérieur ou égal

égalités et inégalités sont génériques (polymorphes)
elles s'appliquent à des arguments de n'importe quel type (les deux arguments doivent être du même type)!!!

```
= : 'a -> 'a -> bool
```

```
== : 'a -> 'a -> bool
```

```
1 # 3 == (1 + 2);;
2   - : bool = true
3 # not(true) = false ;;
4   - : bool = true
```

Produits cartésiens, n-uplets

- ▶ constructeur de valeurs : ,
 - ▶ constructeur de types : *
 - ▶ accesseurs (couples) : fst et snd
-

```
1 # (28, "janvier");;
2 - : int * string
3 # (20, "janvier", 2021);;
4 - : int * string * int
5 # fst (20, "janvier");;
6 - : int = 20
7 # snd (20, "janvier", 2021);;
```

erreur de typage sur la dernière ligne : la fonction snd est appliquée à un triplet et non pas à un couple.

Listes homogènes

- ▶ liste vide : []
 - ▶ constructeur ::
 - ▶ type *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".
```

voir cours 2.

Expression conditionnelle

Syntaxe:

if *expr1* **then** *expr2* **else** *expr3*

- ▶ *expr1* de type *bool*
 - ▶ *expr2* et *expr3* de même type
-

```
1 # if 3 = 4 then 0 else 4;;
2 - : int = 4
3 # if 5 = 6 then 1 else "Non";;
4 - : erreur de typage
5 # (if 3 = 5 then 8 else 10) + 5;;
6 - : int = 15
7 # 5 = 6 || 7 = 9;;
8 - : bool = false
```

Déclarations de valeurs (1)

Déclarations globales:

Syntaxe:

`let p = e`

```
1 # let pi = 3.14159;;  
2 val pi : float = 3.14159  
3  
4 # sin (pi /. 2.0);;  
5 - : float = 0.999999999999
```

Déclarations de valeurs (2)

Déclarations locales:

Syntaxe:

`let p = e1 in e2`

```
1 # let x = 3 in
2     let b = x < 10 in
3         if b then 0 else 10;;
4 - : int = 0
5
6 # b;;
7 Unbound value b
```

Déclarations combinées

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$... **and** $p_n = e_n$; ;

```
1 # let x = 1 and y = 2;;
2 val x : int = 1
3 val y : int = 2
4 # x + y;;
5 - : int = 3
```

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$... **and** $p_n = e_n$ **in** expr ; ;

```
1 # let a = 3.0 and b = 4.0 in sqrt(a*.a+.b*.b);;
2 - : float = 5
```

Valeurs fonctionnelles, fonctions, déclarations

Syntaxe:

function p -> e

- ▶ fonction anonyme : **function** p -> e
 - ▶ de type : `typede(p) -> typede(e)`
-

```
1 # function x -> x + 1;;
2 - : int -> int
3 # (function x -> x + 1) 2019 ;;
4 - : int = 2020
5 # function x -> if x < 0 then -x else x;;
6 - : int -> int
7 # function x -> (function y -> 2*x + 3*y);;
8 - : int -> int -> int
```

```
1 # let succ = function x -> x + 1;;
2 succ : int -> int
3 # succ 420;;
4 - : int = 421
```

Déclaration de valeurs fonctionnelles

Syntaxe:

let f = **function** p1 -> e
ou **let** f = **fun** p -> e
ou **let** f p = e *ou* **let** f (p) = e

```
1 # let pred(x) = x -1;;
2 pred : int -> int = <fun>
3 # let f c = 2*(fst c) + 3*(snd c);;
4 f : int * int -> int = <fun>
5 # f(1,2);;
6 - : int = 8
7 # let g = function x -> (function y -> 2*x + 3*y);;
8 val g : int -> int -> int = <fun>
9 # g 1 2;;
10 - : int = 8
11 # let h = g 1 ;;
12 val h : int -> int = <fun>
13 # h 2 ;;
14 - : int = 8
```

```
1 # let ng = fun x y -> 2 * x + 3 * y ;;
2 val ng : int -> int -> int = <fun>
3 # let nh = ng 1 ;;
4 val nh : int -> int = <fun>
```

Explicitation des types des paramètres des fonctions

Coercicion de types:

Syntaxe:

$(e : t)$

let $p : t = e$

```
1 # let u = [];;  
2 val u : 'a list = []  
3 # let v = (u : int list);;  
4 val v : int list = []
```

C'est principalement utilisé pour les fonctions, on peut écrire, et on écrira

```
1 let xor (x:bool) (y:bool) : bool = (x || y) && (not (x && y))
```

indiquant ainsi que le paramètre x est de type `bool`, le paramètre y de type `bool`, et la fonction retourne un booléen.

Portée des déclarations

► portée lexicale (liaison statique)

```
1 # let p = q + 1;;          (* erreur q inconnue *)
2
3 # let p = p + 1;;        (* erreur p inconnue *)
4
5 # let p = 10;;          p : int = 10
6
7 # let addp x = x + p;;   addp : int ->int = <fun>
8
9 # addp 8;;              - : int = 18
10
11 # let p = 40;;          p : int = 40
12
13 # addp 8;;              - : int = 18
14
15 # p;;                   - : int = 40
```

Appels de fonction

```
1 let fois (a, b) = a * b ;;
2 let moins (a, b) = a - b ;;
3 let carre x = fois (x, x) ;;
4 let delta (a, b, c) = moins (carre b, fois (4, fois (a, c))) ;;
```

empilement dans le cadre d'appel des arguments et des adresses de retour pour chaque appel de fonction

```
1 # #trace moins ;;
2 moins is now traced.
3 # #trace fois ;;
4 fois is now traced.
5 # #trace carre ;;
6 carre is now traced.
7 # #trace delta;;
8 delta is now traced.
```

dépilement des arguments empilés au retour de la fonction

```
1 # delta (2,5,2) ;;
2 delta <-- (2, 5, 2)
3 fois <-- (2, 2)
4 fois --> 4
5 fois <-- (4, 4)
6 fois --> 16
7 carre <-- 5
8 fois <-- (5, 5)
9 fois --> 25
10 carre --> 25
11 moins <-- (25, 16)
12 moins --> 9
13 delta --> 9
14 - : int = 9
```

Curryfication

Passage d'une fonction à plusieurs paramètres en une fonction à un paramètre qui retourne une nouvelle fonction sur le reste des paramètres, cela revient à écrire uniquement des fonctions à un paramètre,

exemple en OCaml : passage d'une fonction prenant un couple d'entiers $(x, y) \rightarrow 2x + 3y$ en une fonction prenant un paramètre entier x et retournant une fonction prenant un paramètre y et calculant $2x + 3y : x \rightarrow (y \rightarrow 2x + 3y)$

```
1 # let f (x,y) = 2*x + 3*y ;;
2 val f : int * int -> int = <fun>
3 # f (1,0) ;;
4 - : int = 2
5 # f (1,1) ;;
6 - : int = 5
7 # f (1,2) ;;
8 - : int = 8
9 # let g = function x ->
10     (function y -> 2*x + 3*y);;
11 val g : int -> int -> int = <fun>
```

```
1 # let h = g 1 ;;
2 val h : int -> int = <fun>
3 # h 0 ;;
4 - : int = 2
5 # h 1 ;;
6 - : int = 5
7 # h 2 ;;
8 - : int = 8
```

Récurtivité (1) : définitions

Syntaxe:

let rec p = *expr*

```
1 # let rec sigma x = if x = 0 then 0
2                       else x + sigma(x-1);;
3 sigma : int -> int = <fun>
4
5 # sigma 10;;
6 - : int = 55
```

réursion mutuelle:

```
1 # let rec odd x = if x = 0 then false else even(x-1)
2   and even x = if x = 0 then true else odd(x-1);;
3 val odd : int -> bool = <fun>
4 val even : int -> bool = <fun>
5
6 # odd 27;;
7 - : bool = true
```

Récurtivité (2) : trace et schéma d'évaluation

trace au toplevel

```
1 # #trace sigma;;
2 sigma is now traced.
3 # sigma 4 ;;
4 sigma <-- 4
5 sigma <-- 3
6 sigma <-- 2
7 sigma <-- 1
8 sigma <-- 0
9 sigma --> 0
10 sigma --> 1
11 sigma --> 3
12 sigma --> 6
13 sigma --> 10
14 - : int = 10
15 # #untrace sigma;;
16 sigma is no longer ←
    traced.
17 # sigma 4;;
18 - : int = 10
```

schéma d'évaluation

$$\begin{aligned}(\text{sigma } 4) &= 4 + (\text{sigma } 3) \\ &= 4 + (3 + (\text{sigma } 2)) \\ &= 4 + (3 + (2 + (\text{sigma } 1))) \\ &= 4 + (3 + (2 + (1 + (\text{sigma } 0)))) \\ &= 4 + (3 + (2 + (1 + 0))) \\ &= 4 + (3 + (2 + 1)) \\ &= 4 + (3 + 3) \\ &= 4 + 6 \\ &= 10\end{aligned}$$

2 phases :

- ▶ la descente réursive : empilement des additions tant qu'il y a des appels à sigma
- ▶ remontée réursive : effectuer les additions jusqu'au résultat final

Récurtivité (3) : terminale

```
1 # let rec sigma_aux (n,a) = if n = 0 then a else sigma_aux (n-1,a+n);;
2 val sigma_aux : int * int -> int = <fun>
3 # let sigma n = sigma_aux (n,0);;
4 val sigma : int -> int = <fun>
```

```
1 # #trace sigma_aux;;
2 sigma_aux is now traced.
3 # #trace sigma;;
4 sigma is now traced.
5 # sigma 4;;
6 sigma <-- 4
7 sigma_aux <-- (4, 0)
8 sigma_aux <-- (3, 4)
9 sigma_aux <-- (2, 7)
10 sigma_aux <-- (1, 9)
11 sigma_aux <-- (0, 10)
12 sigma_aux --> 10
13 sigma_aux --> 10
14 sigma_aux --> 10
15 sigma_aux --> 10
16 sigma_aux --> 10
17 sigma --> 10
18 - : int = 10
```

un appel récursif dans lequel la fonction n'exécute aucune instruction après l'appel est un appel récursif terminal.

$$\begin{aligned}(\text{sigma } 4) &= \text{sigma_aux } (4,0) \\ &= \text{sigma_aux}(3,4) \\ &= \text{sigma_aux}(2,7) \\ &= \text{sigma_aux}(1,9) \\ &= \text{sigma_aux}(0,10) \\ &= 10\end{aligned}$$

dans un appel récursif terminal il est possible de ré-utiliser le cadre d'appel précédent, ce qui permet de ne pas consommer de mémoire supplémentaire.

Récurtivité (4) : définition locale

fonction auxiliaire locale curryfiée.

```
1 # let sigma n =
2     let rec aux n a =
3         if n = 0 then a
4         else aux (n-1) (a+n)
5     in
6         aux n 0 ;;
7     val sigma : int -> int =<←>
8         <fun>
9 # #trace sigma;;
10     sigma is now traced.
11
12 # sigma 4 ;;
13 sigma <-- 4
14 sigma --> 10
15 - : int = 10
```

schéma d'évaluation de (sigma 4) :

```
(sigma 4) = aux 4 0
           = aux 3 4
           = aux 2 7
           = aux 1 9
           = aux 0 10
           = 10
```

pas de trace de fonctions locales

le premier cadre d'appel de aux est réutilisé dans les autres appels à aux. Ces appels récursifs sont en taille mémoire constante.

Récurtivité (5) : définition locale

Une fonction locale peut utiliser l'ensemble de l'environnement accessible.

```
1 # let pow x n =  
2   let rec aux n a =  
3     if n = 0 then a  
4     else aux (n-1) (x * a)  
5   in  
6     aux n 1 ;;  
7 val pow : int -> int -> int = <fun>  
8 # pow 2 5;;  
9 - : int = 32
```

ici à la ligne 4 le paramètre x de pow est utilisé dans aux.

C'est un style camélien assez classique.

Polymorphisme paramétrique (1)

appelé « généralité » dans d'autres langages

- ▶ même code pour des arguments de types différents
- ▶ la fonction n'utilise pas la structure de l'argument

```
1 # let mp a b = a,b;;
2 val mp : 'a -> 'b -> 'a * 'b = <fun>
3
4 # let x = mp 3 6.8;;
5 val x : int * float = 3, 6.8
6
7 # let y = mp true [1];;
8 val y : bool * int list = true, [1]
9
10 # fst x;;
11 - : int = 3
```

```
1 # let id x = x;;
2 val id : 'a -> 'a = <fun>
3 # let app x y = x y;;
4 val app : ('a -> 'b) -> 'a -> 'b = <fun>
5 # app id 1;;
6 - : int = 1
```

OCaml (et F#) : Résumé des expressions rencontrées

```
expr ::= constante
      | ( expr )
      | ident
      | Mod.ident
      | op expr
      | expr infix-op expr
      | if expr then expr else expr
      | function ident -> expr
      | expr expr
      | expr , expr
      | epxr :: epxr
      | let [rec] ident = expr
          [ and ident = expr ]*           // pas en F#
      in expr
```

OCaml : List.hd et List.tl

F# : List.head et List.tail

Filtrage de motif

permet l'accès aux structures de données

- ▶ en testant une valeur
- ▶ en nommant une partie de la structure

motif:

- ▶ assemblage correct (syntaxe et type) d'objets
 - ▶ de types de base (*int*, *bool*, ...)
 - ▶ de paires, listes et constructeurs
 - ▶ d'identificateurs
 - ▶ du motif « ramasse tout » (*_*)
- ▶ ce n'est pas une expression (il n'a pas de calcul)

Syntaxe du filtrage de motif

Syntaxe:

match e **with** $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \dots \mid p_n \rightarrow e_n$

- ▶ filtrage séquentiel de l'expression e par les différents motifs p_i .
- ▶ Si un des motifs correspond à la valeur de e , alors sa branche e_i est évaluée.
- ▶ tous les e_i sont du même type, idem pour les p_i et le type de e
- ▶ motif linéaire (motif non-linéaire comme (x,x)) interdit)
- ▶ détection d'un filtrage non exhaustif
- ▶ détection de branches inutiles

Exemple : fonction imply

► Enumération des cas

```
1 # let imply v =  
2     match v with  
3     | (true,true) -> true  
4     | (true,false) -> false  
5     | (false,true) -> true  
6     | (false,false) -> true;;  
7 val imply : bool * bool -> bool = <fun>
```

► Version plus compacte

```
1 # let imply v =  
2     match v with  
3     | (true,x) -> x  
4     | (false,-) -> true;;  
5 val imply : bool * bool -> bool = <fun>
```

Warnings

▶ filtrage non exhaustif :

```
1 # let f x =
2     match x with
3     | (true, x) -> true
4     | (false, false) -> true;;
5 Warning: this pattern-matching is not exhaustive.
6 Here is an example of a value that is not matched:
7 (false, true)
8 val f : bool * bool -> bool = <fun>
```

▶ cas inutile :

```
1 # let f x =
2     match x with
3     | (a,b) -> true
4     | (true,false) -> false;;
5 Warning: this match case is unused.
6 val f : bool * bool -> bool = <fun>
```

Style camélien : définition par cas

en utilisant le filtrage de motifs sur les différents constructeurs d'une structure de données :

```
1 let rec long l =
2   match l with
3   | [] -> 0
4   | _::q -> 1 + long q ;;
5 val long : 'a list -> int = <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) -> 'a list -> 'b list = <fun>
```

```
1 let rec mem_assoc x l =
2   match l with
3   | [] -> false
4   | (y,_)::q -> if x = y then true else mem_assoc x q ;;
5 val mem_assoc : 'a -> ('a * 'b) list -> bool = <fun>
```

Motif dans les déclarations (1)

► Formes équivalentes:

Syntaxe:

`match e with filtrage` \equiv `(function filtrage) e`

```
1 # match (2,1)
2   with (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
3 - : int = 2
4 # function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
5 - : int * int -> int = <fun>
6 # (function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d)
7   (2,1) ;;
8 - : int = 2
```


Motif dans les déclarations (2)

► Déclarations destructurantes:

Syntaxe:

`let p = e` p est un motif

```
1 # let (x,y) = (3,true);;
2 val x : int = 3
3 val y : bool = true
```

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* = *nom*2

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 : `{ ... }`

▶ accesseurs : `.re` et `.im`

```
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 add_complex c1 c2 =
8     {re=c1.re+.c2.re; im=c1.im+.c2.im};;
9 val add_complex :
10     complex -> complex -> complex = <fun>
11
12 # add_complex c c;;
13 - : complex = {re=4; im=6}
```

Enregistrements (2)

pouvant aussi définir un motif pour un filtrage :

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

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

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

Types rékursifs

Les déclarations de types sont rékursives

```
1 # type intArbre = IntEmpty
2   | IntNode of intArbre * int * intArbre ;;
3 type intArbre = ...
4
5 # let monarbre =
6   IntNode ( IntNode ( IntEmpty, 4, IntEmpty ),
7             2,
8             IntNode ( IntEmpty, 1, IntEmpty ) ) ;;
9 val monarbre : intArbre = ...
10
11 # let rec nb_noeuds a = match a with
12   IntEmpty -> 0
13 | IntNode (fg, _, fd) -> 1 + (nb_noeuds fg) + (nb_noeuds fd);;
14 val nb_noeuds : intArbre -> int = <fun>
15
16 # nb_noeuds monarbre;;
17 - : int = 3
```


Types paramétrés

Les déclarations de types peuvent être paramétrées

```
1 # type 'a arbre = Empty
2     | Node of 'a arbre * 'a * 'a arbre;;
3 type 'a arbre = ...
4
5 # Empty;;
6 - : 'a arbre = Empty
7 # Node (Empty, 1, Empty);;
8 - : int arbre = Node (Empty, 1, Empty)
9 # Node (Empty, 1.0, Empty);;
10 - : float arbre = Node (Empty, 1., Empty)
11 # Node (Node (Empty, 1, Empty), 4, Empty);;
12 - : int arbre = Node (Node (Empty, 1, Empty), 4, Empty)
13
14 # let rec long_ma a = match a with
15     Empty -> 0
16     | Node(fg,_,fd) -> 1 + long_ma fg + long_ma fd;;
17 val long_ma : 'a arbre -> int = <fun>
18
19 # long_ma (Node (Empty, 2, Node (Empty, 3, Empty)));;
20 - : int = 2
21 # long_ma (Node (Empty, 'a', Node (Empty, 'z', Empty)));;
22 - : int = 2
```

Constructeurs et enregistrements en OCaml

L'argument d'un constructeur d'un type somme peut être défini (et utilisé) à la manière d'un enregistrement.

```
1 # type 'a arbreB = Vide
2   | Feuille of 'a
3   | Noeud of {etiq : 'a; fg : 'a arbreB; fd : 'a arbreB} ;;
4
5 # let ma = Noeud {
6   etiq = 4;
7   fg = Noeud {etiq = 2; fg = Feuille 1; fd = Vide};
8   fd = Feuille 8} ;;
9
10 # let rec nombre_noeud a = match a with
11   Vide -> 0
12   | Feuille _ -> 1
13   | Noeud n -> 1 + (nombre_noeud n.fg) + (nombre_noeud n.fd) ;;
```

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
```

Exemples sur les 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 :

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

```
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 :

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

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>
```

```
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 :

$$(append [] [y1; ...; ym]) = [y1; ..; ym]$$

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

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 :

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

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) -> 'a list -> '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) -> 'a list -> '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 [['z'; 'a'; 'm'];
7                       ['.'; 'n'; 'e'];
8                       ['t'];
9                       []] ;;
10 val l2 : int list = [3; 4]
```

```
1 # map (append [10; 20]) ←
2   [[3;4;5]; [8; 3; 1]];;
3 - : int list list = [[10; 20; 3;
4   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 x_i si $0 \leq i \leq n$.

`nth` doit satisfaire les équations conditionnelles suivantes :

$$(nth(x :: xs)i) = xs[i] = 0(nth(x :: xs)i) = (nthxs(i - 1))\text{sinon}$$

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:

```
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)

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

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

Exemple : arbres (1)

Représentation des arbres:

```
1 # type 'a arbre = Empty
2 | Node of 'a * 'a arbre list;;
3 type 'a arbre = ...
4
5 # let rec nombre_noeud a = match a with
6   Empty -> 0
7 | Node (_,[]) -> 1
8 | Node (_,h::t) ->
9   1 + (List.fold_left (+) (nombre_noeud h) (List.map nombre_noeud t));;
10 val nombre_noeud : 'a arbre -> int = <fun>
11
12 # let rec hauteur a = match a with
13   Empty -> 0
14 | Node (_,[]) -> 1
15 | Node (_,h::t) ->
16   1 + (List.fold_left (max) (hauteur h) (List.map hauteur t));;
17 val hauteur : 'a arbre -> int = <fun>
```


Exemple arbres (2)

```
1 # let rec somme_noeud a = match a with
2   Empty -> 0
3   | Node (e,[]) -> e
4   | Node (e,h::t) ->
5     e + (List.fold_left (+) (somme_noeud h) (List.map somme_noeud t));;
6 val somme_noeud : int arbre -> int = <fun>
7
8 # let rec app_arbre e a =
9   let rec app_s_arbre l = match l with
10     [] -> false
11     | h::t -> app_arbre e h || app_s_arbre t
12   in
13     match a with Empty -> false
14     | Node (ne,l) ->
15       ne = e || app_s_arbre l;;
16 val app_arbre : 'a -> 'a arbre -> bool =
```

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 *)
```

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 = 10) 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);;
```

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).
```

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 val mult_list : int list -> int = <fun>
6
7 # mult_list [1;2;3;0;5;6];;
8 - : int = 0
```

Module List (1)

```
1
2 let hd = function
3   [] -> failwith "hd"
4   | a::l -> 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.
```

Exemple : filtrage d'une liste

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

```
1 # exception Identity;;
2 exception Identity
3
4 # let share f x = try f x with Identity -> x;;
5 val share : ('a -> 'a) -> 'a -> 'a = <fun>
6
7 # let filter f l =
8   let rec fil l = match l with
9     | [] -> raise Identity
10    | h :: t ->
11      if f h then h :: fil t else share fil t in
12   share fil l;;
13 val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

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`