# MdP : COURS 2

## Modèles de Programmation


**Emmanuel Chailloux**

# Plan

- Déclaration de types
  - produit cartésien
  - union discriminante
  - types récursifs
  - types paramétrés
- Exemples
- Typage et domaine de définition
- Exceptions en OCaml
- Exceptions dans d'autres langages

# Déclarations de types (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 (2)

- déclarations combinées

  **Syntaxe**:

  **type** *nom1* = . . .
  **and** *nom2* = . . .
  **and** *nomn* = . . . ;;

- avec paramètres

  **Syntaxe**:

  **type** ( p1,p2,. . . ,pn) *nom* = . . . ;;

# Enregistrements (1)

**Syntaxe**:

**type** $t$ ={f1 : t1; f2:t2; . . . ; fn:tn} ;;

---

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

# 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  ;;
9  type carte =...
10
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;;
```

# Types récursifs

## Les déclarations de types sont récursives

```
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 nodes a = match a with
12    IntEmpty -> 0
13 | IntNode (fg, _, fd) -> 1 + (nodes fg) + (nodes fd);;
14 val nodes : intArbre -> int = <fun>
15
16 # nodes monarbre;;
17 - : int = 3
```

# Types paramétrés

## Les déclarations de types peuvent être paramétrées

```
# type 'a arbre = Empty
                | Node of 'a arbre * 'a * 'a arbre;;
type 'a arbre = ...

# Empty;;
- : 'a arbre = Empty
# Node (Empty, 1, Empty);;
- : int arbre = Node (Empty, 1, Empty)
# Node (Empty, 1.0, Empty);;
- : float arbre = Node (Empty, 1., Empty)
# Node (Node (Empty, 1, Empty), 4, Empty);;
- : int arbre = Node (Node (Empty, 1, Empty), 4, Empty)

# let rec long_ma a = match a with
  Empty -> 0
| Node(fg,_,fd) -> 1 + long_ma fg + long_ma fd;;
val long_ma : 'a arbre -> int = <fun>

# long_ma (Node (Empty, 2, Node (Empty, 3, Empty)));;
- : int = 2
# long_ma (Node (Empty, 'a', Node (Empty, 'z', Empty)));;
- : int = 2
```

# 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
```

# Fonctions sur les listes (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 (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 (3)

```
1  let rec fold_left f accu l =
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 =
7    match l with
8      [] -> accu
9    | a::l -> f a (fold_right f l accu)
10
11 (*
12 val fold_left :
13   ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
14 val fold_right :
15   ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
16 *)
```

# Fonctions sur les listes (4)

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

# Exemple: arbres (1)

## Représentation des arbres:

```
# type 'a arbre =  Empty
| Node of 'a * 'a arbre list;;
type 'a arbre = ...

# let rec nombre_noeud a = match a with
   Empty -> 0
| Node (_,[]) -> 1
| Node (_,h::t) ->
     1 + (List.fold_left (+) (nombre_noeud h) (List.map ↩
         nombre_noeud t));;
val nombre_noeud : 'a arbre -> int = <fun>

# let rec hauteur a = match a with
   Empty -> 0
| Node (_,[]) -> 1
| Node (_,h::t) ->
     1 + (List.fold_left (max) (hauteur h) (List.map ↩
         hauteur t));;
val hauteur : 'a arbre -> int = <fun>
```

# Exemple arbres (2)

```
# let rec somme_noeud a = match a with
  Empty -> 0
| Node (e,[]) -> e
| Node (e,h::t) ->
    e + (List.fold_left (+) (somme_noeud h) (List.map ↩
        somme_noeud t));;
val somme_noeud : int arbre -> int = <fun>

# let rec app_arbre e a =
    let rec app_s_arbre  l = match l with
      [] -> false
    | h::t -> app_arbre e h || app_s_arbre t
    in
      match a with  Empty -> false
        | Node (ne,l) ->
            ne = e  || app_s_arbre l;;
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

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

- effectuer une rupture de calcul jusqu'à un récupérateur (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 dún fichier inexistant
  - ...
- comme style de programmation

En Java une exception est un objet

# Exceptions

**Syntaxe**:

**Exception** *E1;;* *ou* **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  # A_MOI;;
4  - : exn = A_MOI
5  # exception Depth of int;;
6  exception Depth of int
7  # Depth 4;;
8  - : 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

```
1 # raise A_MOI;;
2 Uncaught exception: A_MOI
3 # let x = 18;;
4 val x : int = 18
5 # if (x = 0) then raise A_MOI else x;;
6 - : int = 18
```

# 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  let invalid_argument s =
3     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: this pattern-matching ...
9  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 : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = ↩
       <fun>
7
8  # let rec fold_right f l accu =
9    match l with
10     [] -> accu
11   | a::l -> f a (fold_right f l accu)
12 val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =↩
       <fun>
```

# Exemples fonctionnels

```
 1
 2 # fold_left (/) 1000 [3;5;11];;
 3 - : int = 6
 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 # fold_right idiv [3;5;11]  1000;;
10 - : int = 6
11 # fold_right idiv  [3;0;11] 1000;;
12 Exception: Division_by_zero.
```

# Exemple : filtrage d'une liste

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

```
1
2  # exception Identity;;
3  exception Identity
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 $\rightarrow$ rupture du calcul
- style de programmation : exemple précédent (`filter`)

**Attention** au coût du `try`