Les enregistrements

Ce sont des n-uplets dont les champs sont nommés comme les `struct` du C. Les noms des champs doivent commencer par une lettre minuscule.

```
# type int_et_bool = { i : int; b : bool };;
type int_et_bool = { i : int; b : bool; }
#{i=3; b=true};;
- : int_et_bool = {i=3; b=true}
#{b=false; i=7};;
- : int_et_bool = {i=7; b=false}

# let f = function
   {b=true; i=x} -> x*x
 | {b=false; i=x} -> x+x;;
val f : int_et_bool -> int = <fun>

# let f c = let x=c.i in
      if c.b then x*x else x+x;;
val f : int_et_bool -> int = <fun>

# let g = function {i=x} -> x;;
val g : int_et_bool -> int = <fun>

# type ('a,'b) couple = { fst :'a; snd :'b };;
type ('a, 'b) couple = { fst : 'a; snd : 'b; }

# type 'a arbre = Vide | Noeud of 'a noeud
  and  'a noeud =
      { valeur :'a; d:'a arbre; g:'a arbre };;
type 'a arbre = Vide | Noeud of 'a noeud
```



```
type 'a noeud =
     { valeur : 'a; d : 'a arbre; g : 'a arbre; }

# let rec somme = function
   Vide -> 0
 | Noeud {d=t2; valeur=x; g=t1} ->
         x + (somme t1) + (somme t2);;
val somme : int arbre -> int = <fun>

# let rec somme = function
   Vide -> 0
 | Noeud n ->
     n.valeur + (somme n.g) + (somme n.g);;
val somme : int arbre -> int = <fun>
```

Les exceptions

Nous allons décrire le mécanisme d'exception d'Objective Caml.

```
# let tete = function
      [] -> failwith "tete"
  | x::_ -> x;;
val tete : 'a list -> 'a = <fun>
# failwith;;
- : string -> 'a = <fun>

# tete [];;
Uncaught exception: Failure "tete".
```

En Objective Caml, les exceptions appartiennent à un type prédéfini `exn`. Ce type a une particularité. C'est un type somme qui est extensible en ce sens que l'on peut étendre l'ensemble de ses valeurs en déclarant de nouveaux constructeurs. On définit comme cela de nouvelles exceptions. La syntaxe pour définir de nouvelles exceptions est

```
# exception Erreur;;
exception Erreur
# Erreur;;
- : exn = Erreur
```

Le constructeur `Failure` est un constructeur d'exception.

```
# Failure "hello !";;
- : exn = Failure "hello !"
```


La levée d'une exception se fait à l'aide de la fonction `raise`. Remarquez son type.

```
# let tete = function
      [] -> raise (Failure "tete")
  | x::_ -> x;;
val tete : 'a list -> 'a = <fun>
# raise;;
- : exn -> 'a = <fun>
# let tete = function
      [] -> raise Erreur
  | x::_ -> x;;
val tete : 'a list -> 'a = <fun>

# exception Trouve of int;;
exception Trouve of int
# Trouve 5;;
- : exn = Trouve 5
# raise (Trouve 7);;
Uncaught exception: Trouve 7.

# exception Erreur_fatale of string;;
exception Erreur_fatale of string
# raise (Erreur_fatale "cas non prevu");;
Uncaught exception: Erreur_fatale "cas non prevu".
```

La construction `try ... with`

En cas de levée d'une exception, la capture de l'exception permet de continuer un calcul. On peut utiliser les captures d'exception comme un véritable style de programmation. La construction `try ... with` permet de le réaliser les captures. Les différentes étapes à l'exécution sont les suivantes.

- Essayer de calculer l'expression.
- Si aucune erreur n'est déclenchée, on retourne l'expression.
- Si cette évaluation déclenche une erreur qui tombe dans un cas de filtrage, alors on retourne la valeur correspondante de la clause sélectionnée par le filtrage.
- En cas d'échec sur le filtrage, la valeur exceptionnelle est propagée.

Dans ce qui suit, `p` est un prédicat de type `'a -> bool`.

```
# let rec ch p n = function
   [] -> raise Erreur
|x::l -> if (p x) then
         raise (Trouve n) else ch p (n+1) l;;
val ch : ('a -> bool) -> int -> 'a list -> 'b = <fun>

# let cherche p l = try (ch p 1 l) with
  Erreur -> raise (Failure "cherche: rien trouve")
| Trouve n -> n;;
val cherche : ('a -> bool) -> 'a list -> int = <fun>
# cherche pair [1;5;3;4;6;7];;
- : int = 4
# cherche pair [1;5;3];;
Uncaught exception: Failure "cherche: rien trouve".
```


Les aspects impératifs d'Objective Caml

(à utiliser avec modération)

Les enregistrements à champs modifiables

Les champs d'un enregistrement peuvent être déclarés modifiables à l'aide du mot clé `mutable`.

```
# type ex={ a:int; mutable b:(bool*int);
            mutable c:int->int };;
type ex = { a : int; mutable b : bool * int;
            mutable c : int -> int; }

# let r = {b=(false, 7); a=1; c=function x->2*x+1};;
val r : ex = {a=1; b=false, 7; c=<fun>}
# r.b;;
- : bool * int = false, 7
# r.c 3;;
- : int = 7

# let (x,y) = r.b in r.b<-(true, y+7);;
- : unit = ()
# r;;
- : ex = {a=1; b=true, 14; c=<fun>}
# r.c <- (function x->x*x);;
- : unit = ()
# r.c 3;;
- : int = 9
```

Les expressions peuvent être séquencées. Si $e_1, e_2, \ldots, e_n$ sont des expressions, alors $e = e_1; e_2; \ldots; e_n$ est une expression de type, le type de e_n, et de valeur, la valeur de e_n.

```
# type compteur = { mutable cpt:int };;
type compteur = { mutable cpt : int; }
# let incr = let c = { cpt=0 } in
    function () -> c.cpt <- c.cpt+1; c.cpt;;
val incr : unit -> int = <fun>
# incr ();;
- : int = 1
# incr ();;
- : int = 2
```

Les références

Objective Caml fournit un type polymorphe `ref` qui peut être vu comme le type des pointeurs sur une valeur quelconque. On parle de référence sur une valeur. Le type `ref` est défini comme un enregistrement à un champ modifiable.

```
# type 'a ref = {mutable contents: 'a}
```

Ce type est muni de raccourcis syntaxiques prédéfinis.

```
# let compteur = ref 0;;
val compteur : int ref = {contents=0}
# !compteur;;
- : int = 0
# compteur := 2;;
- : unit = ()
# compteur := !compteur +1;;
- : unit = ()
# !compteur;;
- : int = 3

# let incr c = c := !c + 1;;
val incr : int ref -> unit = <fun>
# incr compteur; !compteur;;
```



```
- : int = 4
```

Les boucles

Les différents types de boucles sont

- `while` expression `do` expression `done`
- `for` ident=expression `to|downto` expression
- `do` expression `done`

```
# let imprime_chiffres () =
   for i=0 to 9 do
       print_int i ;
       print_string " "
   done;
   print_newline ();;
val imprime_chiffres : unit -> unit = <fun>
# imprime_chiffres ();;
0 1 2 3 4 5 6 7 8 9
- : unit = ()
```

Les vecteurs

Les vecteurs sont des tableaux à une dimension. Beaucoup de fonctions de manipulations sont définies dans la librairie `Array`. Les indices commencent à 0.

```
# [|1;2;3|];;
- : int array = [|1; 2; 3|]
# let v = Array.create 4 3.14;;
val v : float array = [|3.14; 3.14; 3.14; 3.14|]
# v.(0);;
- : float = 3.14
# v.(4);;
Uncaught exception: Invalid_argument "Array.get".
#  v.(4) <- 5.;;
Uncaught exception: Invalid_argument "Array.set".
#  v.(3) <- 8.;;
- : unit = ()
# v;;
- : float array = [|3.14; 3.14; 3.14; 8|]
# Array.length v;;
- : int = 4
#  let w = Array.copy v;;
val w : float array = [|3.14; 3.14; 3.14; 8|]
# v = w;;
- : bool = true
# v == w;;
- : bool = false
# Array.of_list;;
- : 'a list -> 'a array = <fun>
```



```
# Array.of_list [3;2;8];;
- : int array = [|3; 2; 8|]

# let invt t=
   let n = Array.length t in
    (let rec f i =
       if i=n/2  then ()
       else let x=t.(i) in
              (t.(i)<-t.(n-i-1);
               t.(n-i-1)<-x;
               f(i+1)
               )
     in f 0
    );;
val invt : 'a array -> unit = <fun>

# let t=[|8; 10; 2; 1|];;
val t : int array = [|8; 10; 2; 1|]
# invt t;;
- : unit = ()
# t;;
- : int array = [|1; 2; 10; 8|]
```

Des tableaux à deux dimensions peuvent être créés de la façon suivante.

```
# Array.create_matrix;;
- : dimx:int -> dimy:int ->'a ->'a array array=<fun>
# Array.create_matrix 2 3 "a";;
# let m = Array.create_matrix 2 3 "a";;
val m : string array array =
        [|[|"a"; "a"; "a"|]; [|"a"; "a"; "a"|]|]
#  m.(1).(2);;
- : string = "a"
#  m.(1).(2) <- "toutou";;
- : unit = ()
# m;;
- : string array array =
    [|[|"a"; "a"; "a"|]; [|"a"; "a"; "toutou"|]|]
# Array.length m;;
- : int = 2
# Array.length m.(0);;
- : int = 3
```

Pour d'autres fonctions de la librairie `Array`, se reporter au chapitre 8 du livre de référence.