

Memento OCaml

TYPES

unit rien, singleton ()
int entier de 31 ou 63 bits
float flottant double précision
bool booléen **true** ou **false**
char caractère ASCII simple, 'A'
string chaîne de caractères
'a **list** liste, head :: tail ou [1;2;3]
'a **array** tableau, [1;2;3]
t1 * t2 tuple
int option **None** ou **Some** 3 type optionnel entier

TYPES ALGÈBRIQUES

```
(* type enregistrement *)
type record = {
  v : bool; (* booléen immuable *)
  mutable e : int; (* entier muable *)}

(* usage *)
let r = { v = true; e = 3; }
r.e <- r.e + 1;

(* type somme *)
type sum =
| Constante (* Constructeur de constante, arité 0 *)
| Param of int (* Constructeur avec paramètre *)
| Paire of string * int (* avec deux paramètres *)

let c = Constant
let c = Param 42
let c = Paire ("Jean", 3)
```

VARIABLES GLOBALES ET LOCALES

```
let x = 21 * 2 (* variable globale *)
let s b h = let d = h/2 in b*d (* d est locale à s *)
```

OPÉRATEURS

+ - * / **mod** abs (* entiers *)
+. -. *. /. (* flottants *)
= <= >= < > != (* égalité et comparaison *)
&&, ||, not (* et, ou, non *)
Int.logand 5 3 (* renvoie 1, et bits à bits *)
Int.shift_left 1 3 (* renvoie 2³, décalage à gauche *)

STRUCTURES CONDITIONNELLES

Attention ci-dessous :
expr1 et expr2 doivent être du même type.

```
if condition then expr1 else expr2
if condition then expr
```

Sans le **else**, il faut que le expr soit **unit**.

BOUCLES

```
while cond do
  expr (* évaluée à unit *)
done;
for var = min_value to max_value do
  expr (* évaluée à unit *)
done;
for var = max_value downto min_value do
  expr (* évaluée à unit *)
done;
```

RÉFÉRENCES

L'affectation est un effet de bord.
L'affectation renvoie donc **unit**.

```
let a = ref 3 (* Init. référence *)
a := 42 (* Affectation -> unit *)
let b = !a-3 (* Accès à la valeur *)
```

FILTRAGE DE MOTIFS

```
match expression with
(* exemples de motifs *)
| 42 -> expr (* constante *)
| x when x = 0 -> expr (* condition *)
| (a,b) -> expr (* tuple *)
| Constructeur(a,b) -> expr
| [] -> expr (* liste vide *)
(* déconstruction de liste *)
| head :: tail -> expr
| (a,b,c) :: tail -> expr
| a :: b :: c :: tail -> expr
| (a,_) :: tail -> expr
| _ :: tail -> expr
| [a] -> expr (* liste à un élément *)
| [a;b] -> expr (* liste à deux éléments *)
| _ -> expr (* par défaut *)
```

EXCEPTIONS

```
failwith "Message d'erreur"
exception Boum
raise Boum
try expr with
| Boum -> "Oups..."
```

LISTES (INDUCTIVES, IMMUALES)

```
let lst = [1;2;3;4;5]
let lst = List.init 10 (fun x -> x)
let n = List.length lst
let h = List.hd lst
let t = List.tl lst
let fourth = List.nth lst 3
let rl = List.rev lst
let nl = x::lst (* O(1) *)
let cl = lst @ x (* O(n) *)
let r = List.iter
  (fun e -> print_int e) lst
let r = List.map (fun e -> e*e) lst
let r = List.filter (fun e -> e = 0) lst
let r = List.fold_left
  (fun a e -> 3*e + a) 0 lst
let r = List.fold_left
  max (List.hd lst) lst
let r = List.forall (fun e -> e < 0) lst
let r = List.exists (fun e -> e = 0) lst
let r = List.mem 3 lst
let elem = List.find
  (fun e -> e > 0) lst
```

Un bon entraînement est de parvenir rapidement à écrire ces fonctions (mem, iter, filter, map, find) en OCaml.

TABLEAUX (MUABLES)

```
let a = [1;2;3]
let n = Array.length t
let a = Array.make 10 0
let a = Array.init 10 (fun i -> 10 - i)
let first = a.(0)
a.(3) <- 5 (* affectation -> unit *)
let m = Array.make_matrix 3 3 0
```

CHAÎNES DE CARACTÈRES (IMMUABLES)

```
let s = "Hello"
let s = String.make 10 'z' (* "zzzzzzzzzz" *)
s.[2] (* accès renvoie -> char = 'z' *)
let n = String.length s
let t = s ^ " my friend !" (* concaténation *)
let test = String.equal s t
let test = String.contains 'z' s
let subs = String.sub s debut longueur
(* extraction d'une sous-chaîne *)
```

FONCTIONS

```
let f x = expr          fonction à un paramètre
let rec f x = expr       fonction récursive
f a                      application de f à a
let f x y = expr         deux paramètres
f a b                    application de f à a et b
let f (x : int) =        type contraint
  (fun x -> -x*x)         fonction anonyme
```

```
let f a b = match a mod b with
| 0 -> true  (* filtrage de motif *)
| _ -> false
```

```
let f x =
  (* avec fonction interne récursive *)
  let rec aux param = ...
  in aux x
```

```
let (a,b,c) = (1,2,3) in ...
let (a,b,c) = f n in ...
(* déconstruction d'un tuple *)
```

```
(* filtrage de motif implicite *)
(* un seul paramètre omis *)
```

```
let f = function
| None -> 0 (* filtre un type option *)
| Some(a) -> -a
```

FONCTIONS À CONNAÎTRE

```
let rec length l = (* longueur d'une liste *)
  match l with
  | [] -> 0
  | _::t -> 1 + length t
let rec mem x l = (* à connaître absolument *)
  match l with
  | [] -> false
  | h::_ when h = x -> true
  | _::t -> mem x t
```

(nième élément, exception *)*

```
let rec at k l =
  match l with
  | [] -> failwith "List too short !"
  | h::t when k = 0 -> h
  | _::t -> at (k - 1) t
```

(nième élément, retour optionnel *)*

```
let rec option_at k l =
  match l with
  | [] -> None
  | h::t when k = 0 -> Some h
  | _::t -> option_at (k - 1) t
```

```
let rec iter f l =
  (* f renvoie obligatoirement unit *)
```

```
  match l with
  | [] -> []
  | h::t -> f(h); iter f t;;
iter (fun x -> print_int l) (* usage *)
```

```
let rec map f l = (* à connaître absolument *)
  match l with
  | [] -> []
  | h::t -> f(h)::(map f t);;
```

```
map (fun x -> x*x) l (* usage *)
```

```
let rec last_two l =
  match l with
  | [] | [_] -> failwith "not enough elements"
  | [a; b] -> (a,b)
  | _::t -> last_two t
```

```
let rev list = (* récursive terminale *)
  let rec aux built l =
    match l with
    | [] -> built
    | h::t -> aux (h::built) t in
  aux [] list
```

FONCTIONS À CONNAÎTRE (SUITE)

```
let rec rm e l = (* supprime un élément *)
  match l with
  | [] -> []
  | h::t when h=e -> rm e t
  | h::t -> h::(rm e t);;
```

```
let rm e l = List.filter ((!=) e) l;; (* idem *)
```

```
let rm_dup s = (* supprime les doublons *)
  let rec aux sleft acc =
    match sleft with
    | [] -> acc
    | h::t when List.mem h acc -> aux t acc
    | h::t -> aux t (h::acc)
  in aux s [];;
```

```
let rec filter f to_filter =
  match to_filter with
  | [] -> []
  | h::t when f h -> h::(filter f t)
  | _::t -> filter f t;;
```

LOGIQUE

```
type formule =
| T (* vrai *)
| F (* faux *)
| Var of int (* variable propositionnelle *)
| Not of formule (* négation *)
| And of formule * formule (* conjonction *)
| Or of formule * formule (* disjonction *)
```

(v x renvoie la valeur de vérité de x *)*

```
let rec evaluation v f = match f with
| T -> true
| F -> false
| Var x -> v x
| Not p -> not (evaluation v p)
| And (p, q) -> evaluation v p && evaluation v q
| Or (p, q) -> evaluation v p || evaluation v q
```

GRAPHES

```
(* parcours en largeur *)
let bfs g v0 =
  let visited = Array.make (Array.length g) false in
  let rec explore queue = (* FIFO *)
    match queue with
    | [] -> []
    | v::t when visited.(v) -> explore t
    | v::t -> visited.(v) <- true; v::(explore (t @ g.(v)))
  in explore [v0] ;;
bfs g 0 ;;      (* usage *)
```

ARBRES

```
type 'a tree = Nil | Node of 'a tree * 'a * 'a tree

let rec h a = (* hauteur de l'arbre, O(n) *)
  match a with
  | Nil -> -1
  | Node(fg,_,fd) -> 1 + max (h fg) (h fd)

let rec size a =
  match a with
  | Nil -> 0
  | Node(fg, x, fd) -> 1 + size fg + size fd
```

REGEXP ET AUTOMATES

```
type regexp =      (* expression régulière *)
EmptySet
| Epsilon
| Letter of char
| Sum of  regexp * regexp
| Concat of regexp * regexp
| Kleene of regexp

type ndfsm =        (* automate non déterministe *)
{
  states : int list;
  alphabet : char list;
  initial : int list;
  transitions : (int * char * int) list;
  accepting : int list}

```

TRIS

```
(* tri par insertion *)
let rec insert_elem sorted e =
  match sorted with
  | [] -> [e]
  | h::t when h < e -> h::(insert_elem t e)
  | h::t -> e::h::t
let rec insert_sort l =
  match l with
  | [] -> []
  | e::t -> insert_elem (insert_sort t) e
```

```
(* tri fusion *)
let rec divide_en_2 l =
  match l with
  | [] -> ([],[])
  | [a] -> ([a],[])
  | a::b::t -> let (l1,l2) = divide_en_2 t
               in (a::l1, b::l2)
let rec fusion l1 l2 =
  match (l1,l2) with
  | ([],l2) -> l2
  | (l1,[]) -> l1
  | (a1::t1, a2::_) when a1 < a2 ->
    a1::(fusion t1 l2)
  | (_,a2::t2) -> a2::(fusion l1 t2)
let rec tri_fusion l =
  match l with
  | [] -> []
  | [a] -> [a]
  | l -> let (l1,l2) = divide_en_2 l
        in fusion (tri_fusion l1) (tri_fusion l2)
```

```
(* tri rapide *)
let rec partition l pivot=
  match l with
  | [] -> [],[]
  | h::t when h < pivot ->
    let (l1,l2) = partition t pivot in (h::l1,l2)
  | h::t ->
    let (l1,l2) = partition t pivot in (l1,h::l2)
  (* on pourrait choisir aléatoirement le pivot *)
let rec tri_rapide l =
  match l with
  | [] -> []
  | pivot::t -> let (l1,l2) = partition t pivot in
    (tri_rapide l1)@(pivot::(tri_rapide l2))
```

RECHERCHE DICHOTOMIQUE

```
(* Impératif *)
let dico_mem x tab =
  let n = Array.length tab in
  let b = ref false in
  let g = ref 0 in
  let d = ref (n - 1) in
  (* indices de gauche et de droite *)
  while (not !b) && !g <= !d do
    let m = ((!g) + (!d)) / 2 in
    if tab.(m) = x then
      b := true
    else if tab.(m) < x then
      g := m + 1
    else
      d := m - 1
  done ;
!b

(* Récursif *)
let rec_dicho_mem x tab =
  let rec aux g d =
    if g > d
    then false
    else let m = (g+d)/2 in
         if tab.(m) = x
         then true
         else if tab.(m) < x
         then aux (m+1) d
         else aux g (m-1)
  in aux 0 (Array.length tab - 1)
```