

MÉMENTO, OPTION INFORMATIQUE MPSI/MP

a Types

unit	rien, seule valeur : ()
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

b Types algébriques

(* Définir un type enregistrement nommé record *)

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

```
let r = { v = true; e = 3; }  
r.e <- r.e + 1;
```

(* Définition d'un type somme nommé sum *)

```
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 = Pair ("Jean", 3)
```

c 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 *)
```

d 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 2^3, décalage à gauche *)

e Structure conditionnelle

Attention ci-dessous : expr1 et expr2 **doivent** être du même type. De plus, la syntaxe sans le **else** exige que le **then** envoie unit.

```
if condition then expr1 else expr2  
if condition then expr (* possible si expr = unit () *)
```

f Boucles

```
while cond do ... done;  
for var = min_value to max_value do ... done;  
for var = max_value downto min_value do ... done;
```

g Références

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

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

h Pattern-matching

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

i Exceptions

```
failwith "Message d'erreur"  
exception Paf  
exception Boum of string  
raise Paf  
try expr with  
| Boum "boum boum" -> expr  
| Paf -> expr  
| _ -> expr
```

j Listes (immuables)

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

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

k Tableaux (muables)

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

l Chaînes de caractères (immuables)

```
let s = "Hello"  
let s = String.make 10 'z'  
let n = String.length s  
let t = s ^ " my friend !"  
let test = String.equal s t  
let test = String.contains 'z' s  
let subs = String.sub spos len t
```

m Fonctions

```
let f x = expr      fonction à un paramètre
let rec f x =       fonction récursive
  expr
  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 constraint
(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) = f(n) in ...
(* déconstruction d'un tuple *)
```

```
let f = function (* par filtrage de motif implicite *)
  (* un seul paramètre omis *)
| None -> 0      (* filtre un type option *)
| Some(a) -> -a
```

n Fonctions à connaître

```
let rec length l = (* longueur d'une liste *)
  match l with
  | [] -> 0
  | _::t -> 1 + length t;;
```

```
(* idem mais récursif terminal *)
let length l =
  let rec aux n mylist =
    match mylist with
    | [] -> n
    | _::t -> aux (n + 1) t
  in aux 0 l;;
```

```
(* nième élément, exception si pas trouvé *)
let rec at k l =
  match l with
  | [] -> failwith "not found !"
  | h::_ when k = 0 -> h
  | _::t -> at (k - 1) t;;
```

```
(* nième élément, retourne un type option *)
let rec at k l = match l with
| [] -> None
| h :: t -> if k = 0 then Some h else at (k - 1) t;;
```

```
let rec mem x l = (* à connaître absolument *)
  match l with
  | [] -> false
  | h::_ when h = x -> true
  | _::t -> mem x t;;
```

```
let rec iter f l = (* f renvoie obligatoirement unit *)
  match l with
  | [] -> []
  | h::t -> f(h); iter f t;;
iter (fun x -> Printf.printf "%d\n" x) 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 = (* récursice terminale *)
  let rec aux built l =
    match l with
    | [] -> built
    | h::t -> aux (h::built) t in
  aux [] list;;
```

```
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 [];;
```

o 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 *)
```

p Arbres

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

```
let rec h a = (* hauteur de l'arbre, 0(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
```

q 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
```

r 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;
}
```

s Recherche dichotomique

```
(* Impératif *)
let dico_mem x tab =
  let n = Array.length tab and b = ref false in
  let g = ref 0 and 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_dico_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)
```

t 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 slice l =
  match l with
  | [] -> ([],[])
  | [a] -> ([a],[])
  | a::b::t -> let (l1,l2) = slice t in
               (a::l1, b::l2)
```

```
let rec merge l1 l2 =
  match (l1,l2) with
  | ([],l2) -> l2
  | (l1,[]) -> l1
  | (a1::t1, a2::_) when a1 < a2 ->
    a1::(merge t1 l2)
  | (_::_, a2::t2) -> a2::(merge l1 t2)
```

```
let rec merge_sort l =
  match l with
  | [] -> []
  | [a] -> [a]
  | l -> let (l1,l2) = slice l
         in merge (merge_sort l1) (merge_sort 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)
```

```
let rec quick_sort l =
  match l with
  | []->[]
  | pivot::t -> let (l1,l2) = partition t pivot in
               (quick_sort l1)@(pivot::(quick_sort l2))
  (* on pourrait choisir aléatoirement le pivot... *)
```

u Emacs

i Généralités

M-	touche Meta (Alt ou Esc)
C-	touche Control
S-	touche Shift
C-x C-c	quitter
C-g	annuler la commande
M-	exécuter command
x command	
C-h b	aide sur les commandes

ii Fichiers

C-x C-f	ouvrir un nouveau fichier
C-x C-s	sauvegarder le fichier
C-x b	passer d'un fichier ouvert à un autre
C-x k	fermer le fichier

iii Fenêtres

C-x o	passer sur la fenêtre suivante
C-x 0	fermer la fenêtre

iv Copier coller

C-Space	sélectionner
M-w	copier
C-w	couper
C-y	coller

v Tuareg

C-c C-b	évaluer le code
C-x C-e	évaluer la phrase
C-c C-e	évaluer la phrase
C-M-x	évaluer la phrase
C-c C-k	tuer le processus ocaml
C-c C-t	trouver le type (curseur)
C-c C-s	lancer ocaml