```
(* INFIX FUNCTIONS *)
(+) 3 4;; (* 7 *)
let (+++) x y = x + 2 * y;;

(* FLOAT *)
3.0 +. 1.5 ;;
3.0 -. 1.5 ;;
3.0 *. 1.5 ;;
3.0 /. 1.5 ;;

(* BOOLEAN OPERATOR & CONCATENATION*)
   not && ||          "foo" ^ "bar"
   <> (*different*)

(* CONVERTIONS *)
float_of_int 3 ;;      (* 3.0 *)
int_of_float 3.0 ;;    (* 3 *)
string_of_int 3;;      (* "3" *)

(* LIRE UN BYTES *)
let s = "abc" ;;
s.[0] ;;               (* return a char *)

(* PRINT *)
print_endline "Ok." ;;
print_int 50 ;;
print_newline ;;
Printf.printf "%d,%s,%.3f \n %!" 99 "ok" 4.0 ;;

(* TUPLES *)
let (a,b,c) = (100, 200, 300) ;;
(false, true, -10) ;;  (* it work *)

(* LIST -> SAME TYPE *)
let l0 = []
let l1 = [ 5 ; 4 ; 3 ; 2 ; 1 ] ;;
let l2 = 7 :: 6 :: l1 ;;

(* UNCURRIED FUNCTIONS *)
let f ((x:int), (y:int)):int = x+y;;
let f ((x,y) : int*int):int = x+y;;
let f = function ((x,y) : int*int):int -> x+y;;

(* CURRIED FUNCTIONS *)
let fp (x:int) (y:int) = x+y;;
let fp = function x -> function y -> x+y;;
let fp = fun x -> fun y -> x+y;;
let fp = fun x y -> x+y;;

(* LAMBDA FUNCTIONS *)
let f2 = fun x y -> x + y;;
let f2 x y = x+y;;          (* syntactic sugar *)

(* A OPTION *)
let a = None;;
let b = Some 10;;
let c = Some "OK";;
if x < 10 then None else Some x;;
```

```
(* PATTERN MATCHING  *)
let f x =
    match x with
    | [] -> []
    | a::rest -> a::(f rest);; (* useless *)
let calc = function
    | (x, y, "add") -> x + y
    | (x, y, "sub") -> x - y
    | _ -> 0   (* default *);;
let equal a b =
    match b with
    | x when x = 0 -> true
    | _ -> false;;

(* ENUMERATED DATA TYPES *)
type color = White | Yellow | Green | Blue | Red;;
type role = Player of color * int | Referee;;
let role1 = Referee;;
let role2 = Player (Green, 8);;

(* INNER LET *)
let x = expr1 in expr2;;

(* INVISIBLE ACU *)
let rec rev lst =
    let rec perm acu lst =
        match lst with
        | [] -> acu
        | x::rest -> perm (x::acu) rest
    in perm [] lst;;

(* RECORD *)
type coordinates = { long: float ; lat: float };;
let point1 = { long = -0.3 ; lat = 42.5 };;
let point2 = { point1 with long = -1.0 };;
let get_lat c = c.lat
let get_lat { lat ; _ } = lat (* syntactic sugar *)

(* MUTABLE TYPES *)
type player = { name: string ; mutable points: int };
let foo = { name = "IGS" ; points = 10000};;
foo.points <- 800;;

(* MUTABLES *)
let x = ref "SC >> SI";;
x := "SI >> SC";;

(* SYS MODULE : MUST BE COMPILED *)
let ls = Sys.readdir (Sys.getcwd ())        (* ~ls *)

(* EXCEPTIONS *)
let a = Not_found;;  (* does not raise an exception*)
let a_r = raise Not_found;;  (* raise an exception *)
exception Horrible_error;;
exception Bad_bad_thing of int * string;;

failwith "yo" (* <=> *) raise (Failure "yo")
```

```
(* LIST *)
List.map f [1;4;5];;              (* [f 1; f 4; f 5] *)
List.find (fun x -> x>3) [1;5;80];;   (* return 5 *)
List.nth [3;4;10] 2                   (* return 10*)
List.concat [ [1;2] ; [3;4] ]        (* [1;2;3;4] *)
[1;2]@[3;4]                          (* [1;2;3;4] *)
List.exists (fun x -> x = 6) [1;4;5;6;7]  (* true *)
List.mem 6 [1;4;5;6;7]                    (* true *)
List.for_all (fun x -> x > 0) [1;4;5;6;7] (* true *)
List.filter (fun x -> x > 2) [1;4;5;6] (* [4;5;6] *)
List.length [1;3;4]                       (* 3 *)
List.is_empty [3]                        (* false *)
List.hd [1;3;4]                  (* first elem : 1 *)
List.tl [1;3;4]                   (* last elem : 4 *)
List.rev [1;3;4]                        (* [4;3;1] *)

(* Trouver premier elem qui satisfait un prédicat *)
List.find_opt (fun x -> x>3) [1;5;80]        (* 5 *)
(* -> return None si trouve rien *)
List.find (fun x -> x>3) [1;5;80]            (* 5 *)
(* -> same mais raise Not_found si trouve rien *)

(* associations *)
List.assoc 22 [(11,"bjr");(22,"yo")]      (* "yo" *)
List.assoc_opt 22 [(11,"bjr");(22,"yo")]  (* "yo" *)
(* -> same mais return None si trouve rien *)
List.mem_assoc 22 [(11,"bjr");(22,"yo")]   (* true*)
(* -> regarde si une association existe *)
List.remove_assoc 22 [(11,"bjr");(22,"yo")]
(* -> return [(11, "bjr")] *)

List.split [(1,2);(3,4)]      (* ([1; 3], [2; 4]) *)
List.combine [1; 3] [2; 4]    (* [(1, 2); (3, 4)] *)

(* trier *)
List.sort compare [3; 1; 5; 4]    (* [1; 3; 4; 5] *)
(* -> NB : changer "compare" par une fonction *)
List.sort_uniq compare [3; 1; 3; 1]      (* [1;3] *)
(* -> idem mais supprime les doublons *)

List.init 3 (fun x -> 2*x)           (* [0; 2; 4] *)
List.iter (fun x -> Printf.printf "%d" x) [2;3]
List.map2 (fun x y -> (x,y)) ["A";"C"] ["B";"D"];;
(* -> return [("A", "B"); ("C", "D")] *)

(* List.fold_left *)
let product lst =
  List.fold_left (fun acc x -> acc * x) 1 lst;;

let concatenate strings =
  List.fold_left (fun acc x -> acc ^ x) "" strings;;

let max_elem lst =
  List.fold_left
    (fun acc x -> if x > acc then x else acc)
    min_int
    lst
;;
```

```
(* ARRAYS MODULE *)
let john = Array.make 100 true;; (* 100 elem. true *)
print_int(john.(3));;               (* lire un elem *)
john.(4) <- 0;;                      (* set un elem *)
Array.to_list john;;          (* convertir en liste *)

(* DELETE DUPLICATES *)
let remove_duplicates lst =
  let rec aux seen = function
    | [] -> List.rev seen
    | a::rest ->
        if List.mem a seen then
            aux seen rest
        else aux (a::seen) rest
  in
  aux [] lst;;

(* ERROR HANDLING *)
try (* some code... *) with
    | Not_found -> "Pas trouvé"
    | Failure s -> "Erreur"
    | e -> raise e;;

(* ERROR PRINTING *)
Printf.printf "%s \n" (Printexc.to_string Not_found)

(* HASHTBL *)
let cache f arg =
    let memo = Hashtbl.create 20 in
    fun arg ->
        if Hashtbl.mem memo arg then
            (Hashtbl.find memo arg)
        else
            let c = (f arg) in
            Printf.printf "Computing...\n%!";
            Hashtbl.add memo arg c;
            c

(* LAZINESS *)
(* cf. LAZY sur la doc j'ai la flemme il est 23h *)


(* REMOVE DUPLICATES *)
let remove_duplicate l =
    let insert x l =
        if List.mem x l then
            l
        else x::l
    in List.rev (
        List.fold_left
            (fun acu t -> insert t acu) [] l
    )
;;
```