TYPES / INPUTS / OUTPUTS

int_of_float | float_of_int | int_of_char …

type action = | Avancer | Appel of string | Exemple of (int*int)
type monde = {
grille: (int list) list;
mutable etoile: (int*int) ;}
//mutable signifie que l'on peut modifier l'élément ainsi : m.etoile <- (0,0)
Accéder aux champs de la structure : monde.grille / monde.etoiles ….

INPUTS:
read_int () | read_line () | read_float ()
OUTPUTS
print_char c | print_float f | print_int i | print_string "" | print_endline

LIST / ARRAY

```
val length : 'a list -> int
val iter : ('a -> unit) -> 'a list -> unit
val iteri : (int -> 'a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
```

(uniquement LIST)

```
val nth : 'a list -> int -> 'a
      Return the n-th element of the given list. The first element (head of the
      list) is at position 0. Raise Failure "nth" if the list is too short.
      Raise Invalid_argument "List.nth" if n is negative.
val rev : 'a list -> 'a list
val concat/flatten : 'a list list -> 'a list | ex:[[1;2];[3];[5;4]]->[1;2;3;5;4]
val find : ('a -> bool) -> 'a list -> 'a
val filter : ('a -> bool) -> 'a list -> 'a list
val assoc : 'a -> ('a * 'b) list -> 'b
```

(uniquement ARRAY)
Accéder à une valeur: tab.(i) | matrice.(i).(j)
Modifier une valeur : tab.(i) <- 5

```
val set : 'a array -> int -> 'a -> unit
val concat : 'a array list -> 'a array
val append : 'a array -> 'a array -> 'a array
val copy : 'a array -> 'a array
val to_list : 'a array -> 'a list
val of_list : 'a list -> 'a array
```

STACK / QUEUE

let s = Stack.create () in
let q = Queue.create () in

```
val push : 'a -> 'a t -> unit          ajoute un élément en tête
val pop : 'a t -> 'a                   supprime un element en queue
val top : 'a t -> 'a                   retourne le debut (de la stack/queue)
val clear : 'a t -> unit
val copy : 'a t -> 'a t
val is_empty : 'a t -> bool
val length : 'a t -> int
val iter : ('a -> unit) -> 'a t -> unit
val fold : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b
```

BTREE / GTREE

```
type ‘a btree =
            | Empty
            | Node of  ‘a * ‘a btree * ‘a btree

Let rec taille abr = match abr with
                        | Empty -> 0
                        | Node (x,g,d) -> 1 + taille g + taille d

Let rec hauteur abr = match abr with
                        | Empty -> 0
                        | Node (x,g,d) -> 1 + max (hauteur g) (hauteur d)

Let rec insert abr x = match abr with
                           | Empty -> Node (x, Empty, Empty)
                           | Node (x,g,d) -> if x<= e
                                                  then Node(e, insert g x, d)
                                                  else Node(e, g, insert d x)


type (‘a, ’b) gtree =
            | Empty
            | Node of ‘a * (‘b * (‘a, ‘b) gtree) list

let rec hauteur t =
            match t with
            | Node (_,[]) -> 0
            | Node (_,l) -> 1+(hauteur_liste l)
      and hauteur_liste l =
            match l with
            | [] -> 0
            | (c,a)::xs -> max (hauteur a) (hauteur_liste xs)


let rec taille t =
            match t with
            | Node (_,[]) -> 1
            | Node (_,l) -> 1+(somme_taille l)
      and somme_taille l =
            match l with
            | [] -> 0
            | x ::xs -> taille x + somme_taille xs




Type ‘a option =
            | None
            | Some of ‘a
```