val <nombre> : <tipo_expresion> = <valor_expresion_evaluada> -> después de una definición con let
-: <tipo_expresion> = <valor_expresion_evaluada> -> evaluación de resultados
Una secuencia también puede conducir a una excepción, se demotan: Exception: <constructor_excepcion> “mensaje_o_nombre">

PROGRAMACION FUNCIONAL
MODULO LIST:

```
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val nth : 'a list -> int -> 'a (Devuelve el n-elemento, comenzando en 0)
val init : int -> (int -> 'a) -> 'a list (Crea lista de n elementos de f n, comenzando en 0)
val find : ('a -> bool) -> 'a list -> 'a (Encuentra el primer elemento que cumple un predicado)
val mem : 'a -> 'a list -> bool (Verifica si un elemento está en la lista)
val map : ('a -> 'b) -> 'a list -> 'b list (Aplica una función a cada elemento de la lista)
val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc (Aplica una función acumulativa de izquierda a derecha)
val fold_right : ('a -> 'acc -> 'acc) -> 'a list -> 'acc -> 'acc (No es recursiva terminal)
val filter : ('a -> bool) -> 'a list -> 'a list
val compare_lengths : 'a list -> 'b list -> int (0->iguales//-1->a<b//1->a>b)
val append : 'a list -> 'a list -> 'a list
val concat o flatten : 'a list list -> 'a list (No es recursiva terminal)
val length : 'a list -> int
val rev : 'a list -> 'a list
val exists : ('a -> bool) -> 'a list -> bool (false si esta vacia)
val for_all : ('a -> bool) -> 'a list -> bool (true si esta vacia)
val iter : ('a -> unit) -> 'a list -> unit (Aplica una función a cada elemento de la lista sin devolver nada)
```

PROGRAMACION IMPERATIVA
MODULO ARRAY:
(CUIDADO PORQUE SON MUTABLES)

```
val length : 'a array -> int
val get : 'a array -> int -> 'a (Lo mismo que v.(n))
val set : 'a array -> int -> 'a -> unit (Lo mismo que v.(n) <- x)
val make : int -> 'a -> 'a array (Inicializa el vector de tamaño int a 'a. Si 'a es mutable, cambian todos)
val init : int -> (int -> 'a) -> 'a array
val append : 'a array -> 'a array -> 'a array
val concat : 'a array list -> 'a array
val sub : 'a array -> int -> int -> 'a array (sub a pos len: array desde pos hasta pos+len-1)
val copy : 'a array -> 'a array
val fill : 'a array -> int -> int -> 'a -> unit (Cambia el valor a x desde pos hasta pos+len-1)
val to_list : 'a array -> 'a list
val of_list : 'a list -> 'a array
val iter : ('a -> unit) -> 'a array -> unit
val map : ('a -> 'b) -> 'a array -> 'b array
val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a array -> 'acc
val fold_right : ('a -> 'acc -> 'acc) -> 'a array -> 'acc -> 'acc
val for_all : ('a -> bool) -> 'a array -> bool
val exists : ('a -> bool) -> 'a array -> bool
val mem : 'a -> 'a array -> bool
```

REGISTROS/ESTRUCTURAS:

```
# type persona = {nombre: string; edad:int};;
type persona = { nombre : string; edad : int; }
# let p = {nombre = "luis";edad=20};;
val p : persona = {nombre = "luis"; edad = 20}
# p.edad;;
- : int = 20
# let mas_viejo p ={ p with edad = p.edad + 1};;
```

```
O tambien: # let mas_viejo p = {nombre = p.nombre;edad=p.edad+1};;
val mas_viejo : persona -> persona = <fun>
# type persona = {nombre: string; mutable edad:int};;
type persona = { nombre : string; mutable edad : int; }

REFERENCIAS:
# let n = ref 0;;
val n : int ref = {contents = 0}
# let turno () = n := !n +1; !n;;
val turno : unit -> int = <fun>
# turno();;
- : int = 1
# turno();;
- : int = 2

MODULOS/CLASES/OBJETOS:
# module Contador : sig
        val turno : unit -> int
        val reset : unit -> unit
end =
struct
        let n = ref 0
        let turno () = n := !n + 1; !n
        let reset () = n := 0
end;;
module Contador : sig val turno : unit -> int val reset : unit -> unit end
# class counter = object
        val mutable n = 0
        method next = n <- n+1;n
        method reset = n <- 0
end;;
class counter : object val mutable n : int method next : int method reset : unit end
# type counter = < next : int; reset : unit >;;
type counter = < next : int; reset : unit >
# let new_counter () : counter = object
        val mutable n = 0
        method next = n <- n+1;n
        method reset = n <- 0
end;;
val new_counter : unit -> counter = <fun>
# let c3 = new_counter ();;
val c3 : counter = <obj>
# let c1 = object
        val mutable n = 0
        method next = n <- n+1;n
        method reset = n <- 0
end;;
val c1 : < next : int; reset : unit > = <obj>
# class ['a] queue = object (self)
        val mutable front = []
        val mutable back = []
        method push (e: 'a)=
                back <- e::back
        method top = match front,back with
                h::_,_ -> Some h
                |[],[] -> None
                |[],_ -> front <- List.rev back;
                         back <- [];
                         self#top
        method pop = match front,back with
                h::t,_ -> front <- t; Some h
                |[],[] -> None
                |[],_ -> front <- List.rev back;
                                back <- [];
                                self#pop
```

```
end;;
class ['a] queue :
  object
    val mutable back : 'a list
    val mutable front : 'a list
    method pop : 'a option
    method push : 'a -> unit
    method top : 'a option
  end
# let q = new queue;;
val q : '_weak1 queue = <obj>
# q#top;;
- : '_weak1 option = None
# q#push "Hola";;
- : unit = ()
# q#top;;
- : string option = Some "Hola"
# let rec pushl l q = match l with
        |[] -> ()
        |h::t -> q#push h; pushl t q
;;
val pushl : 'a list -> < push : 'a -> 'b; .. > -> unit = <fun>
# let rec drain q = match q#pop with
        None -> ()
        |_ -> drain q
;;
val drain : < pop : 'a option; .. > -> unit = <fun>


PREGUNTAS VARIAS
# let print_newline ( ) = print_endline " ";;
val print_newline : unit -> unit = <fun>
# let lista = (1,2) :: (3,4)::(5,6)::[];;
val lista : (int * int) list = [(1, 2); (3, 4); (5, 6)]
# let _::lista = lista in List.tl lista;;
- : (int * int) list = [(5, 6)]
# let x,y = List.hd lista;;
val x : int = 1
val y : int = 2
# (2 < 2 / 1) && (1 < 1 /0);;
- : bool = false
# let iter f v = for i = 0 to Array.length v - 1 do f v.(i) done;;
val iter : ('a -> 'b) -> 'a array -> unit = <fun>
# let v = Array.init 3 (fun i -> object method number = i end);;
val v : < number : int > array = [|<obj>; <obj>; <obj>|]
# let pr_number v = iter (fun e -> print_int e#number) v;;
val pr_number : < number : int; .. > array -> unit = <fun>
# pr_number v; print_newline ();;
- : unit = ()
# let x,y = 2+1,0;;
val x : int = 3
val y : int = 0
# (function x -> function y -> 2*y) y x;;
- : int = 6
# let h = fun x y -> y::x;;
val h : 'a list -> 'a -> 'a list = <fun>
# h ['h'];;
- : char -> char list = <fun>
# h [] [0];;
- : int list list = [[0]]
# let x,y = y,x;;
val x : int = 0
val y : int = 3
# let v = ref x;;
val v : int ref = {contents = 0}
```

```
# v+1;;
Error: This expression has type int ref
       but an expression was expected of type int
# let w=v;;
val w : int ref = {contents = 0}
# w:=!w+1;!v,!w;;
- : int * int = (1, 1)
# List.map (fun x-> x,x) ['a';'e';'i'];;
- : (char * char) list = [('a', 'a'); ('e', 'e'); ('i', 'i')]
# let x = 1 in
        for i=1 t  o 3 do
        let x = 2*x in print_int x
        done;
        print_  int x;;
2221- : unit = ()
# let x y = y,y;;
val x : 'a -> 'a * 'a = <fun>
val x : float = 5.
val y : float = 10.
# 2 * min_int;;
- : int = 0
# let x::y = ['a';'e';'i';'o';'u'];;
val x : char = 'a'
val y : char list = ['e'; 'i'; 'o'; 'u']
# List. tl y;;
- : char list = ['i'; 'o'; 'u']
# x::y;;
- : char list = ['a'; 'e'; 'i'; 'o'; 'u']
# let x=[(1,2);(3,4)];;
val x : (int * int) list = [(1, 2); (3, 4)]
# let x::y = x in x::x::y;;
- : (int * int) list = [(1, 2); (1, 2); (3, 4)]
# let (x,y) :: _ = List.tl x;;
val x : int = 3
val y : int = 4
# x + (let x = x + y in x + y);;
- : int = 14
# (function x-> x, x) "hola";;
- : string * string = ("hola", "hola")
# List.fold_right (-) [4;3;2];;
- : int -> int = <fun>
# List.fold_right (-) [4;3;2] 1;;
- : int = 2
# List.map (function x -> 2 * x + 1);;
- : int list -> int list = <fun>
# let x = let x = 3 in x * x;;
val x : int = 9
# x + let x = x + 1 in x * x;;
- : int = 109
# function x -> x;;
- : 'a -> 'a = <fun>
# let x::y = let _::t = [1;2;3] in t;;
val x : int = 2
val y : int list = [3]
# List.fold_left (fun x f -> f x) 0 [(+) 1; (-) 10; fun x -> x*x];;
- : int = 81
# let l = let o1 = object method name = "01"
  end in o1::[];;
val l : < name : string > list = [<obj>]
# let sufix s1 s2 = s2 ^s1;;
val sufix : string -> string -> string = <fun>
# let past = sufix "ed";;
val past : string -> string = <fun>
# past "show";;
- : string = "showed"
```

```
# let rec p = function 0 -> true | n -> i (n-1) and i = function 0-> true | n-> p(n-1);;
val p : int -> bool = <fun>
val i : int -> bool = <fun>
# let x::y::z = [1]@[2];;
val x : int = 1
val y : int = 2
val z : int list = []
# z::[];;
- : int list list = [[]]
# let five _ = 5;;
val five : 'a -> int = <fun>
# let id x = x and apply x y = x y;;
val id : 'a -> 'a = <fun>
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# five 0, id 0, (id five) 0, id (five 0), apply five true;;
- : int * int * int * int * int = (5, 0, 5, 5, 5)
# let (|>) x f = f x;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
# -2 |> abs |> (+) 3 |> function x -> x * x;;
- : int = 25
# let x f = f,f;;
val x : 'a -> 'a * 'a = <fun>
# let a::b = [x 1; x 2] in (a,b);;
- : (int * int) * (int * int) list = ((1, 1), [(2, 2)])
# let f, x = (+), 0;;
val f : int -> int -> int = <fun>
val x : int = 0
# f x;;
- : int -> int = <fun>
# let y = x + 1,x - 1;;
val y : int * int = (1, -1)
# (function x -> x) (function s -> s ^ s);;
- : string -> string = <fun>
# let x, y = 1, 5;;
val x : int = 1
val y : int = 5
# let x y = y + 1 in x;;
- : int -> int = <fun>
# let x = let y = x < y in y;;
val x : bool = true
# let x y = y + 1 in x;;
- : int -> int = <fun>
# x;;
- : bool = true
# function x -> function y -> function z -> z y x;;
- : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c = <fun>
# let no f x = not (f x);;
val no : ('a -> bool) -> 'a -> bool = <fun>
# let apa x f = f x;;
val apa : 'a -> ('a -> 'b) -> 'b = <fun>
# List.map (apa 2) [(function x -> x * x); succ; (+)1; (-)1];;
- : int list = [4; 3; 3; -1]
# let x, y = -2.5, 2.5;;
val x : float = -2.5
val y : float = 2.5
# let dup f x = f (fst x), f (snd x);;
val dup : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>
# dup (+);;
- : int * int -> (int -> int) * (int -> int) = <fun>
# let p = dup floor (y, x);;
val p : float * float = (2., -3.)
# 2. ** 3.;;
- : float = 8.
# sqrt 2.;;
- : float = 1.41421356237309515
```

```
# sqrt 4;;
error de tipo
# sqrt 2. ** 2. = 2.;;
- : bool = false
# 0.1 +. 0.2 <= 0.3;;
- : bool = false
# 3.0 = float_of_int 3;;
- : bool = true
# truncate 2.1 + truncate (-2.9);;
- : int = 0
# ceil 2.1 +. ceil (-2.9);;
- : float = 1.
# int_of_float -2.9;;
error sintactico, falta parentesis
# "1999" + "1";;
error de tipo, deberia ser "1999" ^ "1";;
# "2" < "12";;
- : bool = true
# ();;
- : unit = ()
# (());;
- : unit = ()
# 0, ();;
- : int*unit = (0, ())
# Char.chr (Char.code 'M' + Char.code 'a' - Char.code 'A');;
- : char='m'
# let maximo = max_int;;
val maximo : int = 4611686018427387903
# let minimo = min_int;;
val minimo : int = -4611686018427387904
# minimo + maximo;;
- : int = -1
# maximo + 1;;
- : int = -4611686018427387904
# 2 * maximo;;
- : int = -2
# let maximo = 1. /. 0.;;
val maximo : float = infinity
# let minimo = -1.0 /. 0.;;
val minimo : float = neg_infinity
# 1. /. maximo;;
- : float = 0.
# 1. /. minimo;;
- : float = -0.
# 1. /. maximo = 1. /. minimo;;
- : bool = true
# 0. /. 0.;;
- : float = nan
# maximo -. maximo;;
- : float = nan
# maximo +. minimo;;
- : float = nan
# abs -1;;
error de tipo. Falta parentesis

ARBOLES
BINTREE:
type 'a bintree = Empty | BT of 'a bintree * 'a * 'a bintree
type 'a t = 'a bintree

let empty = Empty
let is_empty x = x = Empty
let leaftree x = BT (Empty, x, Empty)
let root = function
  | Empty -> raise (Failure "root")
  | BT (_, x, _) -> x
let left_b = function
  | Empty -> raise (Failure "left_b")
  | BT (l, _, _) -> l
let right_b = function
  | Empty -> raise (Failure "right_b")
  | BT (_, _, r) -> r
let root_replacement t x =
  match t with
  | Empty -> raise (Failure "root_replacement")
  | BT (l, _, r) -> BT (l, x, r)
let left_replacement t l =
  match t with
  | Empty -> raise (Failure "left_replacement")
  | BT (_, x, r) -> BT (l, x, r)
let right_replacement t r =
  match t with
  | Empty -> raise (Failure "right_replacement")
  | BT (l, x, _) -> BT (l, x, r)
let rec size = function
  | Empty -> 0
  | BT (l, _, r) -> 1 + size l + size r
let rec height = function
  | Empty -> 0
  | BT (l, _, r) -> 1 + max (height l) (height r)
let rec preorder t =
  let rec aux l_aux = function
    | Empty -> l_aux
    | BT (l, x, r) -> x :: aux (aux l_aux r) l
  in aux [] t
let rec inorder t =
  let rec aux l_aux = function
    | Empty -> l_aux
    | BT (l, x, r) -> aux (x :: aux l_aux r) l
  in aux [] t
let rec postorder t =
  let rec aux l_aux = function
    | Empty -> l_aux
    | BT (l, x, r) -> aux (aux (x :: l_aux) r) l
  in aux [] t
let breadth a =
  let rec aux l_aux = function
    | [] -> (
        match l_aux with
        | [] -> []
        | _ -> aux [] (List.rev l_aux))
    | Empty :: t -> aux l_aux t
    | BT (l, x, r) :: t -> x :: aux (r :: l :: l_aux) t
  in aux [] [ a ]
let leaves t =
  let rec aux l_aux = function
    | Empty -> l_aux
    | BT (Empty, x, Empty) -> x :: l_aux
    | BT (l, _, r) -> aux (aux l_aux r) l
  in aux [] t
let rec find_in_depth f = function
  | Empty -> raise Not_found
  | BT (l, x, r) -> (
      if f x then x
      else try find_in_depth f l with Not_found -> find_in_depth f r)
let rec find_in_depth_opt f = function
  | Empty -> None
  | BT (l, x, r) -> (
      if f x then Some x
      else match find_in_depth_opt f l with
```

```
        | Some x -> Some x
        | None -> find_in_depth_opt f r)
let rec exists f = function
  | Empty -> false
  | BT (l, x, r) -> if f x then true else exists f l || exists f r
let rec for_all f = function
  | Empty -> false
  | BT (l, x, r) -> if f x then true else for_all f l && for_all f r
let rec map f = function
  | Empty -> Empty
  | BT (l, x, r) -> BT (map f l, f x, map f r)
let rec mirror = function
  | Empty -> Empty
  | BT (l, x, r) -> BT (mirror r, x, mirror l)
let rec replace_when f t aux =
  match t with
  | Empty -> Empty
  | BT (l, x, r) -> if f x then aux else BT (replace_when f l aux, x, replace_when f r aux)
let rec cut_above f = function
  | Empty -> Empty
  | BT (l, x, r) -> if f x then Empty else BT (cut_above f l, x, cut_above f r)
let rec cut_below f = function
  | Empty -> Empty
  | BT (l, x, r) -> if f x then BT (Empty, x, Empty) else BT (cut_below f l, x, cut_below f
r)

STBINTREE:
type 'a st_bintree = Leaf of 'a | SBT of 'a st_bintree * 'a * 'a st_bintree
type 'a t = 'a st_bintree

let leaftree x = Leaf x
let is_leaf = function
  | Leaf _ -> true
  | _ -> false
let comb x l r = SBT (l, x, r)
let root = function
  | Leaf x -> x
  | SBT (_, x, _) -> x
let left_b = function
  | Leaf _ -> raise (Failure "left_b")
  | SBT (l, _, _) -> l
let right_b = function
  | Leaf _ -> raise (Failure "right_b")
  | SBT (_, _, r) -> r
let root_replacement t x =
  match t with
  | Leaf _ -> raise (Failure "root_replacement")
  | SBT (l, _, r) -> SBT (l, x, r)
let left_replacement t l =
  match t with
  | Leaf _ -> raise (Failure "left_replacement")
  | SBT (_, x, r) -> SBT (l, x, r)
let right_replacement t r =
  match t with
  | Leaf _ -> raise (Failure "right_replacement")
  | SBT (l, x, _) -> SBT (l, x, r)
let rec size = function
  | Leaf _ -> 1
  | SBT (l, _, r) -> 1 + size l + size r
let rec height = function
  | Leaf _ -> 1
  | SBT (l, _, r) -> 1 + max (height l) (height r)
let rec preorder t =
  let rec aux l_aux = function
    | Leaf x -> x :: l_aux
    | SBT (l, x, r) -> x :: aux (aux l_aux r) l
  in aux [] t
let rec inorder t =
  let rec aux l_aux = function
    | Leaf x -> x :: l_aux
    | SBT (l, x, r) -> aux (x :: aux l_aux r) l
  in aux [] t
let rec postorder t =
  let rec aux l_aux = function
    | Leaf x -> x :: l_aux
    | SBT (l, x, r) -> aux (aux (x :: l_aux) r) l
  in aux [] t
let breadth a =
  let rec aux l_aux = function
    | [] -> (
        match l_aux with
        | [] -> []
        | _ -> aux [] (List.rev l_aux))
    | Leaf x :: t -> x :: aux l_aux t
    | SBT (l, x, r) :: t -> x :: aux (r :: l :: l_aux) t
  in aux [] [ a ]
let leaves t =
  let rec aux l_aux = function
    | Leaf x -> x :: l_aux
    | SBT (l, _, r) -> aux (aux l_aux r) l
  in aux [] t
let rec find_in_depth f = function
  | Leaf x -> if f x then x else raise Not_found
  | SBT (l, x, r) -> (
      if f x then x
      else try find_in_depth f l with Not_found -> find_in_depth f r)
let rec find_in_depth_opt f = function
  | Leaf x -> if f x then Some x else None
  | SBT (l, x, r) -> (
      if f x then Some x
      else
        match find_in_depth_opt f l with
        | Some x -> Some x
        | None -> find_in_depth_opt f r)
let rec exists f = function
  | Leaf x -> f x
  | SBT (l, x, r) -> if f x then true else exists f l || exists f r
let rec for_all f = function
  | Leaf x -> f x
  | SBT (l, x, r) -> if f x then true else for_all f l && for_all f r
let rec map f = function
  | Leaf x -> Leaf (f x)
  | SBT (l, x, r) -> SBT (map f l, f x, map f r)
let rec mirror = function
  | Leaf x -> Leaf x
  | SBT (l, x, r) -> SBT (mirror r, x, mirror l)
let rec replace_when f t aux =
  match t with
  | Leaf x -> if f x then aux else Leaf x
  | SBT (l, x, r) ->
      if f x then aux else SBT (replace_when f l aux, x, replace_when f r aux)
let rec cut_below f = function
  | Leaf x -> Leaf x
  | SBT (l, x, r) ->
      if f x then Leaf x else SBT (cut_below f l, x, cut_below f r)

GTREE:
type 'a gtree = GT of 'a * 'a gtree list
type 'a t = 'a gtree
```

```
let leaftree x = GT (x, [])
let root (GT (x, _)) = x
let branches (GT (_, hijos)) = hijos
let rec size (GT (_, hijos)) = 1 + List.fold_left (fun suma l -> suma + size l) 0 hijos
let rec height (GT (_, hijos)) = 1 + List.fold_left (fun alt l -> max alt (height l)) 0 hijos
let preorder (GT (x, hijos)) =
  let rec aux l_aux = function
    | [] -> l_aux
    | GT (x, hijos) :: t -> x :: aux (aux l_aux t) hijos
  in x :: aux [] hijos
let postorder (GT (x, hijos)) =
  let rec aux l_aux = function
    | [] -> l_aux
    | GT (x, hijos) :: t -> aux (x :: aux l_aux hijos) t
  in List.rev (x :: aux [] hijos)
let breadth (GT (x, hijos)) =
  let rec aux acc = function
    | [] -> List.rev acc
    | GT (x, hijos) :: t -> aux (x :: acc) (t @ hijos)
  in aux [] [ GT (x, hijos) ]
let leaves a =
  let rec aux l_aux = function
    | GT (x, []) -> x :: l_aux
    | GT (_, hijos) -> List.fold_left aux l_aux hijos
  in List.rev (aux [] a)
let find_in_depth f a =
  let rec aux = function
    | [] -> raise Not_found
    | GT (x, hijos) :: t -> (
        if f x then x else try aux hijos with Not_found -> aux t)
  in aux [ a ]
let find_in_depth_opt f a =
  let rec aux = function
    | [] -> None
    | GT (x, hijos) :: t -> (
        if f x then Some x
        else
          match aux hijos with
          | Some x -> Some x
          | None -> aux t)
  in aux [ a ]
let breadth_find f a =
  let rec aux = function
    | [] -> raise Not_found
    | GT (x, hijos) :: t -> if f x then x else aux (t @ hijos)
  in aux [ a ]
let breadth_find_opt f a =
  let rec aux = function
    | [] -> None
    | GT (x, hijos) :: t -> if f x then Some x else aux (t @ hijos)
  in aux [ a ]
let exists f a =
  let rec aux = function
    | [] -> false
    | GT (x, hijos) :: t -> if f x then true else aux hijos || aux t
  in aux [ a ]
let for_all f a =
  let rec aux = function
    | [] -> false
    | GT (x, hijos) :: t -> if f x then true else aux hijos && aux t
  in aux [ a ]
let map f (GT (x, hijos)) =
  let rec aux = function
    | [] -> []
    | GT (x, hijos) :: t -> GT (f x, aux hijos) :: aux t
  in GT (f x, aux hijos)
let rec mirror (GT (x, hijos)) = GT (x, List.rev (List.map mirror hijos))
let rec replace_when f (GT (x, hijos)) aux =
  if f x then aux else GT (x, List.map (fun l -> replace_when f l aux) hijos)
let rec cut_below f (GT (x, hijos)) =
  if f x then GT (x, []) else GT (x, List.map (cut_below f) hijos)

FOLDING:
let i_prod = function
  | [] -> raise (Failure "i_prod: La lista esta vacia")
  | h :: t -> List.fold_left ( * ) 1 t
let f_prod = function
  | [] -> raise (Failure "i_prod: La lista esta vacia")
  | h :: t -> List.fold_left ( *. ) 1. t
let lmin = function
  | [] -> raise (Failure "lmin: La lista esta vacia")
  | h :: t -> List.fold_left min h t
let min_max = function
  | [] -> raise (Failure "min_max: La lista esta vacia")
  | h :: t -> (List.fold_left min h t, List.fold_left max h t)
let rev l = List.fold_left (fun a b -> b :: a) [] l
let rev_append l1 l2 = List.fold_left (fun a b -> b :: a) l2 l1
let rev_map f l = List.fold_left (fun a b -> f b :: a) [] l
let concat l = List.fold_left (fun a b -> a ^ b) "" l

LISTING:
let from0to n =
  if n < 0 then raise (Failure "from0to: n no puede ser negativo")
  else
    let rec aux n_aux l_aux =
      if n_aux > n then l_aux else n_aux :: aux (n_aux + 1) l_aux
    in
    aux 0 []
let to0from n =
  if n < 0 then raise (Failure "from0to: n no puede ser negativo")
  else
    let rec aux n_aux l_aux =
      if n_aux < 0 then l_aux else aux (n_aux - 1) (n_aux :: l_aux)
    in
    aux n []
let pair x l =
  let rec aux = function
    | [] -> []
    | h :: t -> (x, h) :: aux t
  in
  aux l
let remove_first x l =
  let rec aux = function
    | [] -> []
    | h :: t -> if x = h then t else h :: aux t
  in
  aux l
let remove_all x l =
  let rec aux = function
    | [] -> []
    | h :: t -> if x = h then aux t else h :: aux t
  in
  aux l
let ldif l1 l2 =
  let rec aux l_aux = function
    | [] -> List.rev l_aux
    | h :: t -> if List.mem h l2 then aux l_aux t else aux (h :: l_aux) t
  in
  aux [] l1
```

```
TAIL:
let concat l =
  let rec aux acc = function
    | [] -> List.rev acc
    | h :: t -> aux (h @ acc) t
  in
  aux [] l
let front l =
  let rec aux acc = function
    | [] -> raise (Failure "front")
    | [ _ ] -> List.rev acc
    | h :: t -> aux (h :: acc) t
  in
  aux [] l
let compress l =
  let rec aux acc = function
    | h1 :: (h2 :: _ as t) -> if h1 = h2 then aux acc t else aux (h1 :: acc) t
    | [ h ] -> List.rev (h :: acc)
    | [] -> List.rev acc
  in
  aux [] l
let ofo l =
  let rec aux acc = function
    | [] -> List.rev acc
    | h :: t -> if List.mem h acc then aux acc t else aux (h :: acc) t
  in
  aux [] l
let fold_right' f l x =
  let rec aux acc l =
    match l with
    | [] -> acc
    | h :: t -> aux (f h acc) t
  in
  aux x (List.rev l)
let sublists l =
  let rec aux acc = function
    | [] -> acc
    | h :: t ->
        let sublista = List.map (fun sub -> h :: sub) acc in
        aux (acc @ sublista) t
  in
  aux [ [] ] l

ARRAY LOOP:
let append a1 a2 =
  let tam1 = Array.length a1 in
  let tam2 = Array.length a2 in
  if tam1 + tam2 > Sys.max_array_length then raise (Invalid_argument "append")
  else
    Array.init (tam1 + tam2) (fun i ->
        if i < tam1 then a1.(i) else a2.(i - tam1))
let sub a pos tam =
  if pos < 0 || tam < 0 || pos + tam > Array.length a then
    raise (Invalid_argument "sub")
  else Array.init tam (fun i -> a.(pos + i))
let copy a =
  let tam = Array.length a in
  Array.init tam (fun i -> a.(i))
let fill a pos tam x =
  if pos < 0 || tam < 0 || pos + tam > Array.length a then
    raise (Invalid_argument "fill")
  else
    for i = pos to pos + tam - 1 do
      a.(i) <- x
    done
```

```
let blit a1 pos1 a2 pos2 tam =
  if
    pos1 < 0 || pos2 < 0 || tam < 0
    || pos1 + tam > Array.length a1
    || pos2 + tam > Array.length a2
  then raise (Invalid_argument "blit")
  else
    for i = 0 to tam - 1 do
      a2.(pos2 + i) <- a1.(pos1 + i)
    done
let to_list a =
  let tam = Array.length a in
  let l = ref [] in
  for i = tam - 1 to 0 do
    l := a.(i) :: !l
  done;
  !l
let iter f a =
  let len = Array.length a in
  for i = 0 to len - 1 do
    f a.(i)
  done
let fold_left f acc a =
  let acc_aux = ref acc in
  for i = 0 to Array.length a - 1 do
    acc_aux := f !acc_aux a.(i)
  done;
  !acc_aux

let for_all f a =
  let aux = ref true in
  let cont = ref 0 in
  while !cont < Array.length a && !aux do
    if not (f a.(!cont)) then aux := false else cont := !cont + 1
  done;
  !aux

let exists f a =
  let aux = ref false in
  let cont = ref 0 in
  while !cont < Array.length a && not !aux do
    if f a.(!cont) then aux := true else cont := !cont + 1
  done;
  !aux

let find_opt f a =
  let aux = ref None in
  let cont = ref 0 in
  while !cont < Array.length a && !aux = None do
    if f a.(!cont) then aux := Some a.(!cont) else cont := !cont + 1
  done;
  !aux

LIST LOOP:
let length l =
  let cont = ref 0 in
  let aux = ref l in
  while !aux <> [] do
    cont := !cont + 1;
    aux := List.tl !aux
  done;
  !cont
let last l =
  if l = [] then failwith "last"
  else
```

```
    let t = ref (List.tl l) in
    let h = ref (List.hd l) in
    while !t <> [] do
      h := List.hd !t;
      t := List.tl !t
    done;
    !h
let nth l pos =
  if pos < 0 then raise (Invalid_argument "nth") (*pos tiene que ser positiva*)
  else
    let cont = ref 0 in
    let l_aux = ref l in
    while !cont < pos && !l_aux <> [] do
      l_aux := List.tl !l_aux;
      cont := !cont + 1
    done;
    if !l_aux = [] then failwith "nth" (*No se ha encontrado el elemento*)
    else List.hd !l_aux
let rev l =
  match l with
  | [] -> []
  | _ ->
      let l_rev = ref [] in
      let l_aux = ref l in
      while !l_aux <> [] do
        l_rev := List.hd !l_aux :: !l_rev;
        l_aux := List.tl !l_aux
      done;
      !l_rev
let append l1 l2 =
  let l1_aux = ref l1 in
  let l2_aux = ref l2 in
  let l_rev = ref [] in
  while !l1_aux <> [] do
    l_rev := List.hd !l1_aux :: !l_rev;
    l1_aux := List.tl !l1_aux
  done;
  while !l_rev <> [] do
    l2_aux := List.hd !l_rev :: !l2_aux;
    l_rev := List.tl !l_rev
  done;
  !l2_aux
let concat l =
  let l_concat = ref [] in
  let l_aux = ref l in
  while !l_aux <> [] do
    let h = ref (List.hd !l_aux) in
    while !h <> [] do
      l_concat := List.hd !h :: !l_concat;
      h := List.tl !h
    done;
    l_aux := List.tl !l_aux
  done;
  let l_rev = ref [] in
  while !l_concat <> [] do
    l_rev := List.hd !l_concat :: !l_rev;
    l_concat := List.tl !l_concat
  done;
  !l_rev
let for_all f l =
  let l_aux = ref l in
  let aux = ref true in
  while !l_aux <> [] && !aux do
    if f (List.hd !l_aux) then l_aux := List.tl !l_aux else aux := false
  done;
  !aux
let exists f l =
  let l_aux = ref l in
  let aux = ref false in
  while !l_aux <> [] && not !aux do
    if f (List.hd !l_aux) then aux := true else l_aux := List.tl !l_aux
  done;
  !aux
let find_opt f l =
  let l_aux = ref l in
  let aux = ref None in
  while !l_aux <> [] && !aux = None do
    let h = List.hd !l_aux in
    if f h then aux := Some h else l_aux := List.tl !l_aux
  done;
  !aux
let iter f l =
  let l_aux = ref l in
  while !l_aux <> [] do
    f (List.hd !l_aux);
    l_aux := List.tl !l_aux
  done
let fold_left f acc l =
  let l_aux = ref l in
  let acc_aux = ref acc in
  while !l_aux <> [] do
    acc_aux := f !acc_aux (List.hd !l_aux);
    l_aux := List.tl !l_aux
  done;
  !acc_aux

let hd = function
  | [] -> raise (Failure "hd: Lista vacia")
  | h :: t -> h
let tl = function
  | [] -> raise (Failure "tl: Lista vacia")
  | h :: t -> t
let last l =
  let rec aux = function
    | [] -> raise (Failure "last: Lista vacia")
    | [ x ] -> x
    | _ :: t -> aux t
  in
  aux l
let rec length = function
  | [] -> 0
  | h :: t -> 1 + length t
let length' l =
  let rec aux count = function
    | [] -> count
    | h :: t -> aux (count + 1) t
  in
  aux 0 l
let compare_lengths l1 l2 =
  let rec aux l_aux1 l_aux2 =
    match (l_aux1, l_aux2) with
    | [], [] -> 0
    | [], _ -> -1
    | _, [] -> 1
    | _ :: t1, _ :: t2 -> aux t1 t2
  in
  aux l1 l2
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | h1 :: t1 -> h1 :: append t1 l2
let rec rev_append l1 l2 =
  let rec aux l_aux = function
    | [] -> l_aux
    | h :: t -> aux (h :: l_aux) t
  in
  aux l2 l1
let rev l =
  let rec aux l_aux = function
    | [] -> l_aux
    | h :: t -> aux (h :: l_aux) t
  in
  aux [] l
let rec concat = function
  | [] -> []
  | h :: t -> append h (concat t)
let flatten = concat
let init len f =
  if len < 0 then raise (Invalid_argument "init: lenght is negative")
  else
    let rec aux n l_aux = if n < 0 then l_aux else aux (n - 1) (f n :: l_aux) in
    aux (len - 1) []
let rec nth l n =
  if n < 0 then raise (Invalid_argument "nth: n is negative")
  else
    let rec aux cont = function
      | [] -> raise (Failure "nth: List is too short or n not found")
      | h :: t -> if n = cont then h else aux (cont + 1) t
    in
    aux 0 l
let rec map f l =
  match l with
  | [] -> []
  | h :: t -> f h :: map f t
let rec rev_map f l =
  let rec aux l_aux = function
    | [] -> l_aux
    | h :: t -> aux (f h :: l_aux) t
  in
  aux [] l
let rec map2 f l1 l2 =
  match (l1, l2) with
  | [], [] -> []
  | [], _ | _, [] ->
      raise (Invalid_argument "map2: Lists have different lengths")
  | h1 :: t1, h2 :: t2 -> f h1 h2 :: map2 f t1 t2
let rec combine l1 l2 =
  match (l1, l2) with
  | [], [] -> []
  | [], _ | _, [] ->
      raise (Invalid_argument "combine: Lists have different lengths")
  | h1 :: t1, h2 :: t2 -> (h1, h2) :: combine t1 t2
let rec split = function
  | [] -> ([], [])
  | (h1, h2) :: t ->
      let t1, t2 = split t in
      (h1 :: t1, h2 :: t2)
let find f l =
  let rec aux f_aux l_aux =
    match (f_aux, l_aux) with
    | _, [] -> raise Not_found
    | _, h :: t -> if f h then h else aux f_aux t
  in
  aux f l
let rec filter f l =
  match l with
  | [] -> l
  | h :: t -> if f h then h :: filter f t else filter f t
let filter' f l =
  let rec aux l_aux = function
    | [] -> l_aux
    | h :: t -> if f h then h :: aux l_aux t else aux l_aux t
  in
  aux [] l
let rec partition f l =
  match l with
  | [] -> ([], [])
  | h :: t ->
      let si, no = partition f t in
      if f h then (h :: si, no) else (si, h :: no)
let partition' f l =
  let rec aux l_aux = function
    | [] -> l_aux
    | h :: t ->
        let si, no = aux l_aux t in
        if f h then (h :: si, no) else (si, h :: no)
  in
  aux ([], []) l
let for_all f l =
  let rec aux = function
    | [] -> true
    | h :: t -> if f h then true && aux t else false
  in
  aux l
let exists f l =
  let rec aux = function
    | [] -> false
    | h :: t -> if f h then true else false || aux t
  in
  aux l
let mem n l =
  let rec aux = function
    | [] -> false
    | h :: t -> if h = n then true else aux t
  in
  aux l
let fold_left f x l =
  let rec aux x_aux = function
    | [] -> x_aux
    | h :: t -> aux (f x_aux h) t
  in
  aux x l
let rec fold_right f l x =
  match l with
  | [] -> x
  | h :: t -> f h (fold_right f t x)
let rec binstr_of_int x =
  if x = 0 then "0"
  else if x = 1 then "1"
  else binstr_of_int (x / 2) ^ string_of_int (x mod 2)
let rec int_of_binstr x =
  let n = String.length x in
  if n > Sys.int_size then
    (*Volvemos a llamarla, limitando el tamaño a 63(int_ize)*)
    int_of_binstr (String.sub x (n - Sys.int_size) Sys.int_size)
  else if n = 0 then 0
  else if x.[0] = '0' then
    (*Si el elemento es 0, lo ignoramos y pasamos al siguiente*)
    int_of_binstr (String.sub x 1 (n - 1))
  else
```

```
    (*Si el elemento es 1, elevamos 2 al respectivo tamaño del string y llamamos
      a la funcion con el resto del string*)
    int_of_float (2. ** float_of_int (n - 1))
    + int_of_binstr (String.sub x 1 (n - 1))
let rec int_of_binstr' x =
  (*Lo que hacemos es eliminar los espacios del string antes de pasarselo
    a la funcion que transforma a decimal*)
  let rec eliminar_espacios x =
    if String.contains x ' ' then
      (*Le pasamos el mismo string, pero quitando ese espacio*)
      eliminar_espacios
        (String.sub x 0 (String.index x ' ')
        ^ String.sub x
            (String.index x ' ' + 1)
            (String.length x - String.index x ' ' - 1))
    else x
  in
  let x = eliminar_espacios x in
  let n = String.length x in
  if n > Sys.int_size then
    (*Volvemos a llamarla, limitando el tamaño a 63(int_ize)*)
    int_of_binstr (String.sub x (n - Sys.int_size) Sys.int_size)
  else if n = 0 then 0
  else if x.[0] = '0' then
    (*Si el elemento es 0, lo ignoramos y pasamos al siguiente*)
    int_of_binstr (String.sub x 1 (n - 1))
  else
    (*Si el elemento es 1, elevamos 2 al respectivo tamaño del string y llamamos
      a la funcion con el resto del string*)
    int_of_float (2. ** float_of_int (n - 1))
    + int_of_binstr (String.sub x 1 (n - 1))

type 'a tree = Leaf of 'a | Node of ('a tree * 'a * 'a tree);;
let rec peso_maximo = function
        Leaf p -> p
        | Node (i, c, d) -> c +. max (peso_maximo i) (peso_maximo d);;

let rec caminos = function
        Leaf p -> [[p]]
        | Node (i, c, d) -> let f l = c::l in List.map f (caminos i) @ List.map f (caminos
d);;
let camino_maximo a =
        let rec aux = function
                Leaf a -> a, []
                | Node (i, n, d) ->
                        let (p1, c1) = aux i
                        and (p2, c2) = aux d
                                in if p1 > p2 then n +. p1, 'I'::C1
                                else n +. p2, 'D'::c2
        in snd (aux a);;
```

Define una función aBase: int -> int -> int List tal que aBase b n devuelva
la lista de enteros correspondiente a los dígitos de la representación en
base b del número n, y una función deBase: int -> int List -> int tal que
deBase b l devuelva el número cuya representación en base b corresponde a
la lista l:

```
let rec aBase a b =
        if b < a then [b]
        else [b mod a] @ aBase a (b/a);;

let rec debase b l = match l with
        [] -> raise (Failure "deBase")
        | h::[] -> h
        | h::t -> h + b * deBase h t;;
```

```
(*Version optimizada de fibonacci:*)
let fib n =
  (*Guaramos en fi el resultado de sumarse con fa(que es el fi anterior) y aumentamos el
contador j*)
  let rec aux j fi fa = if j = n then fi else aux (j + 1) (fi + fa) fi in
  aux 0 0 1
let rec fib_to i n =
  let x = fib i in
  if x < n then
    (*mostramos el numero y llamamos al siguiente de la serie*)
    let () = print_endline (string_of_int x) in
    fib_to (i + 1) n
  else print_newline ()
let () =
  (*Si el tamaño de la entrada es distinto de 2, mostramos error*)
  if Array.length Sys.argv <> 2 then
    print_endline "fibto: Invalid number of arguments"
  else fib_to 1 (int_of_string Sys.argv.(1))
```