Toplevel

```
(*Si recibe un nombre (let)*)
val <name> : <type> = <val>
(*Si no*)
- : <type> = <val>

(* TIPOS *)
Primitivo : <type> = <val>
Funcion : <type> -> <type> (*-> ...*) = <func>
Unit : unit = ()
Listas : <type> list = [<val>; <val>; <val>]
Variables : <type> ref = {contents = <val>}
Array : <type> array = [|<val>; <val>; <val>|]
Records : <record name> = {<arg1> = <val>; <arg2> = <val>;}

(*EJEMPLOS*)

# [1;2;3;4];;
- : int list = [1; 2; 3; 4]

------------------------------------------------------

# List.map2;;
- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>

-------------------------------------------------------

let variable = ref 4

val variable : int ref = {contents = 4}
--------------------------------------------------------

# [|3;4;5|];;
- : int array = [|3; 4; 5|]

---------------------------------------------------------

type cosa = {a1 : int; cosa : char}

# let c = {a1 = 3; cosa = 'a'};;
val c : cosa = {a1 = 3; cosa = 'a'}
```

Notas

- Se puede usar pattern matching al nombrar valores

- ```
  let x,y = (3,4);;
  val x : int = 3
  val y : int = 4
  ```

- Entrada/salida
  - Salida:
    - output_char, output_string, output_byte ...
    - out_channel
      - stdout, stderr, open_out
    - flush
  - Entrada:
    - input_line, input_char ...
    - in_channel
      - stdin, open_in
- Exceptiones
  - ```
    raise (Invalid argument "string")
    ```

Codigo

Arboles

```
let rec size = function
    Gt (_,[]) -> 1
  | Gt (r,h::t) -> size h + size (Gt (r,t));;

let rec height = function
    Gt (_,[]) -> 1 |
    Gt (r,l) -> 1 + List.fold_left max 0 (List.map (height) l);;

let leaves tree =
    let rec aux acc = function
        Gt (r, []) -> r::acc |
        Gt (_, l) -> List.concat (List.map (aux acc) l)
    in aux [] tree;;

let rec mirror = function
    Gt (r, l) -> Gt (r, List.rev (List.map mirror l));;

let rec preorder = function
        Gt (r, []) -> [r] |
        Gt (r, l) -> r::List.concat (List.map preorder l);;

let rec postorder = function
        Gt (r, []) -> [r] |
        Gt (r, l) -> List.concat (List.map postorder l) @ [r];;
```

Practica listas

```
let hd = function
  [] -> raise (Failure "hd") |
  h::_ -> h;;

let tl = function
  [] -> raise (Failure "tl") |
  _::t -> t;;

let length l =
  let rec suma_length i = function
    [] -> i |
    _::t -> suma_length (i+1) t
  in suma_length 0 l;;

let rec compare_lengths l1 l2 = match (l1, l2) with
  [], [] -> 0 |
  [], _ -> -1 |
  _, [] -> 1  |
  _::t1, _::t2 -> compare_lengths t1 t2;;

let rec nth l n =
  if n < 0 then raise (Invalid_argument "nth")
  else match (l, n) with
    ([], _) -> raise (Failure "nth") |
    (h::_, 0) -> h |
    (_::t, _) -> nth t (n-1);;

let rec append l1 l2 = match l1 with
 [] -> l2|
 h::t -> h::(append t l2);;

let rec find f l= match l with
  h::t -> ((function true -> h|
                false -> find f t )(f h)) |
  [] -> raise Not_found;;

let rec for_all f l = match l with
  h::t -> f h &&  for_all f t |
  [] -> true;;

let rec exists f l = match l with
h::t -> (f h || exists f t) |
[] -> false;;

let rec mem x l = match l with
  h::t -> ((x=h) || mem x t) |
  [] -> false;;

let rev l =
  let rec aux v = function
    [] -> v |
    h::t -> aux (h::v) t
  in aux [] l;;
```

```
let filter f l =
  let rec aux acc f l = match l with
    h::t -> if (f h) then aux (h::acc) f t  else aux acc f t |
    [] -> rev acc
  in aux [] f l;;

let find_all f l =
  let rec aux acc f l = match l with
    h::t -> if (f h) then aux (h::acc) f t  else aux acc f t |
    [] -> rev acc
  in aux [] f l;;

let partition f l =
  let rec aux tacc facc f l = match l with
    h::t -> if (f h) then aux (h::tacc) facc f t  else aux tacc (h::facc) f t |
    [] -> (rev tacc, rev facc)
  in aux [] [] f l;;

let split l=
  let rec aux acc1 acc2 l = match l with
    h::t -> aux (fst h::acc1) (snd h::acc2) t |
    [] -> (rev acc1,rev acc2)
  in aux [] [] l;;

let rec combine l1 l2 = match l1, l2 with
    ([], []) -> [] |
    h1::t1, h2::t2 -> (h1, h2):: combine t1 t2 |
    _,_ -> raise (invalid_arg "combine");;

let init n f =
  if n < 0 then raise (Invalid_argument "init")
  else
    let rec aux acc i n f =
      if i < n then aux ((f i)::acc) (i+1) n f
      else (rev acc) in
    aux [] 0 n f;;

let rev l =
  let rec aux v = function
    [] -> v |
    h::t -> aux (h::v) t
  in aux [] l;;

let rec rev_append l1 l2 = match l1 with
    []->l2 |
    h::t -> rev_append t (h::l2);;

let concat l=
  let rec aux acc l = match l with
    h::t -> aux (rev_append h acc) t |
    [] -> rev acc
  in aux [] l;;

let flatten l= concat 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 v = function
    [] -> v |
    h::t -> aux (f h::v) t
  in aux [] l;;

let rec map2 f l1 l2 = match l1, l2 with
    ([], []) -> [] |
    (h1::t1, h2::t2) -> (f h1 h2)::map2 f t1 t2 |
    _ , _ -> raise (Invalid_argument "map2");;

let rec fold_left f n l = match l with
  [] -> n |
  h::t -> fold_left f (f n h) t;;

let rec fold_right op l e = match l with
  [] -> e |
  h::t -> op h (fold_right op t e)
```

Tour

```
let possible m n path (x, y)=
    not (List.mem (x, y) path) && x > 0 && x <= m && y > 0 && y <= n ;;

let next_jump m n path (x, y) =
    let jumps = [(x+1, y-2);(x+1, y+2);(x-1, y-2);(x-1, y+2);
                 (x+2, y-1);(x+2, y+1);(x-2, y-1);(x-2, y+1)]
    in List.filter (possible m n path) jumps

let tour m n (xi,yi) (xf,yf) =
    let rec aux1 path (xs,ys) =
        if (xs, ys) = (xf, yf) then List.rev ((xs, ys)::path)
        else let next_jumps = next_jump m n path (xs, ys) in
            let rec aux2 = function
                [] -> raise (Not_found)
              | h::t -> try aux1 ((xs, ys)::path) h
                        with Not_found -> aux2 t
            in aux2 next_jumps
    in aux1 [] (xi, yi);;
```

Qsort

```
let rec qsort1 ord = function
    [] -> []
    | h::t -> let after, before = List.partition (ord h) t
              in qsort1 ord before @ h :: qsort1 ord after;;

let rec qsort2 ord =
    let append' l1 l2 = List.rev_append (List.rev l1) l2 in
    function
    [] -> []
  | h::t -> let after, before = List.partition (ord h) t
            in append' (qsort2 ord before) (h :: qsort2 ord after);;
```

Msort

```
let rec divide l = match l with
    h1::h2::t -> let t1, t2 = divide t in (h1::t1, h2::t2)
  | _ -> l, [];;

let rec merge f = function
    [], l | l, [] -> l
  | h1::t1, h2::t2 -> if (f h1) h2 then h1 :: merge f (t1, h2::t2)
                      else h2 :: merge f (h1::t1, t2);;

let rec msort1 f l = match l with
    [] | _::[] -> l
  | _ -> let l1, l2 = divide l
         in merge f (msort1 f l1, msort1 f l2);;

(*Msort 2*)

let divide' l =
    let rec aux acc1 acc2 l1 = match l1 with
        [] -> List.rev acc1, List.rev acc2 |
        h1::[] -> aux (h1::acc1) (acc2) [] |
        h1::h2::t -> aux (h1::acc1) (h2::acc2) t
    in aux [] [] l;;

let merge' f (l1, l2) =
    let rec aux acc = function
        [],[] -> List.rev acc |
        [], h::t | h::t, [] -> aux (h::acc) ([], t) |
        h1::t1, h2::t2 -> if f h1 h2 then aux (h1::acc) (t1,(h2::t2))
        else aux (h2::acc) (h1::t1, t2)
    in aux [] (l1,l2);;


let rec msort2 f l = match l with
    [] | _::[] -> l
  | _ -> let l1, l2 = divide' l
         in merge'  (f) (msort2 f l1, msort2 f l2);;
```

Logic

```
(* IMPLIMENTACION 1 *)

type log_exp =
    Const of bool
  | Var of string
  | Neg of log_exp
  | Disj of log_exp * log_exp
  | Conj of log_exp * log_exp
  | Cond of log_exp * log_exp
  | BiCond of log_exp * log_exp;;

let rec eval ctx = function
    Const b -> b
  | Var s -> List.assoc s ctx
  | Neg e -> not (eval ctx e)
  | Disj (e1, e2) -> (eval ctx e1) || (eval ctx e2)
  | Conj (e1, e2) -> (eval ctx e1) && (eval ctx e2)
  | Cond (e1, e2) -> (not (eval ctx e1)) || (eval ctx e2)
  | BiCond (e1, e2) -> (eval ctx e1) = (eval ctx e2);;

(* IMPLIMENTACION 2*)

type oper = Not;;

type biOper = Or | And | If | Iff;;

type prop =
    C of bool
  | V of string
  | Op of oper * prop
  | BiOp of biOper * prop * prop;;

(* FUNCTIONS *)

(*A*)
let rec prop_of_log_exp = function
    Const c -> C c
  | Var v -> V v
  | Neg e -> Op (Not, (prop_of_log_exp e))
  | Disj (e1, e2) -> BiOp (Or, prop_of_log_exp e1, prop_of_log_exp e2)
  | Conj (e1, e2) -> BiOp (And, prop_of_log_exp e1, prop_of_log_exp e2)
  | Cond (e1, e2) -> BiOp (If, prop_of_log_exp e1, prop_of_log_exp e2)
  | BiCond (e1, e2) -> BiOp (Iff, prop_of_log_exp e1, prop_of_log_exp e2);;

let rec log_exp_of_prop = function
    C c -> Const c
  | V v -> Var v
  | Op(Not, prop) -> Neg (log_exp_of_prop prop)
  | BiOp (Or, e1, e2) -> Disj ((log_exp_of_prop e1), (log_exp_of_prop e2))
  | BiOp (And, e1, e2) -> Conj ((log_exp_of_prop e1), (log_exp_of_prop e2))
  | BiOp (If, e1, e2) -> Cond ((log_exp_of_prop e1), (log_exp_of_prop e2))
  | BiOp (Iff, e1, e2) -> BiCond ((log_exp_of_prop e1), (log_exp_of_prop e2));;

(*B*)
```

```
let opval = function
    Not -> (not);;

let biopval = function
    Or -> (||)
  | And -> (&&)
  | If -> (fun x y -> not x || y)
  | Iff -> (=);;

let rec peval ctx = function
    C b -> b
  | V v -> List.assoc v ctx
  | Op (p, e) -> (opval p) (peval ctx e)
  | BiOp (p, e1, e2) -> (biopval p) (peval ctx e1) (peval ctx e2);;

(*C*)
let rec combinations = function
    0 -> [[]]
  | n -> let comb_f = List.map (fun l -> false::l) (combinations (n-1)) in
         let comb_t = List.map (fun l -> true::l) (combinations (n-1)) in
          comb_t @ comb_f

let get_vars p =
    let rec aux acc = function
        V v-> if not (List.mem v acc) then v::acc else []
      | Op (_, p) -> aux acc p
      | BiOp (_, p1, p2) -> List.rev_append (aux acc p1) (aux acc p2)
      | C c -> []
    in aux [] p;;

let is_tau p =
    let vars = get_vars p in
    let all_ctx = List.map (List.combine vars) (combinations (List.length
vars)) in
    let flipped = Fun.flip peval in
    List.for_all (fun x -> x = true) (List.map  (flipped p) all_ctx);;
```

Perms

```
let rec descending = function
    h1::h2::t -> h1 >= h2 && descending (h2::t) |
    _ -> true;;

let rec ascending = function
    h1::h2::t -> h1 <= h2 && ascending (h2::t) |
    _ -> true;;

let rec qsort ord = function
    [] -> []
    | h::t -> let after, before = List.partition (ord h) t
              in qsort ord before @ h :: qsort ord after;;
```

```
let next_number l =
  (* Dada una lista l con cabeza h, devuelve una lista l' con cabeza h' igual
al siguiente
     elemento a h en orden topologico de los presentes en l. El resto de
elementos se situan
     en la cola sin ningun orden concreto *)
   let rec aux1 n acc1 l1 = (*Buscar el primer número mayor que h*)
        let rec aux2 r acc2 l2 = match l2 with (*Busca el siguiente en orden
topologico*)
            [] -> r::acc2 |
            h::t -> if h > n  && h < r then aux2 h (r::acc2) t
            else aux2 r (h::acc2) t
        in match l1 with
        [] -> raise (Invalid_argument "next_number") |
        h::t -> if h > n then aux2 h (n::acc1) t
            else aux1 n (h::acc1) t
    in aux1 (List.hd l) [] (List.tl l)

let prev_number l =
  (* Equivalente a next_number pero situa como h' al elemente previo en orden
topologico*)
    let rec aux1 n acc1 l1 =
        let rec aux2 r acc2 l2 = match l2 with
            [] -> r::acc2 |
            h::t -> if h < n  && h > r then aux2 h (r::acc2) t
            else aux2 r (h::acc2) t
        in match l1 with
        [] -> raise (Invalid_argument "prev_number") |
        h::t -> if h < n then aux2 h (n::acc1) t
            else aux1 n (h::acc1) t
    in aux1 (List.hd l) [] (List.tl l)

let reorder_next l =  let c = next_number l in
  (*Devuelve la siguiente permutacion de una lista l con una cola sin más
permutaciones siguientes*)
    (List.hd c):: qsort (<=) (List.tl c)

let reorder_prev l =  let c = prev_number l in
  (*Devuelve la anterior permutacion de una lista l con una cola sin más
permutaciones previas*)
    (List.hd c):: qsort (>=) (List.tl c)



let rec next l =
  if descending l then raise Not_found (*Si l esta en orden decentente no tiene
más permutaciones siguientes*)
  else match l with
    h::[] -> [h] |
    h1::h2::[] -> h2::[h1] |
    h1::t -> (try h1::next(t) with
      Not_found -> reorder_next l)|
    [] -> [];;

let rec prev l =
  if ascending l then raise Not_found  (*Si l esta en orden ascendente no tiene
más permutaciones previas*)
  else match l with
    h::[] -> [h] |
```

```
    h1::h2::[] -> h2::[h1] |
    h1::t -> (try h1::prev(t) with
      Not_found -> reorder_prev l)|
    [] -> [];;

let allperms l =
  let rec aux_next acc=
    try  aux_next (next (List.hd acc)::acc) with
      Not_found -> List.rev acc in
  let rec aux_prev acc =
    try  aux_prev (prev (List.hd acc)::acc) with
      Not_found -> acc
  in aux_prev [l] @ try let next_element = next l in aux_next [next_element]
with
(* En caso de que l sea el ultimo elemento, concatena
```

Imperativo

```
let output_multiples x a b =
    for i = a to b do
        if (i mod x) = 0 then
            Format.print_int i
    done;;
```