- **Devolver el último elemento de una lista:**

```
# let rec last = function
  | [] -> None
  | [ x ] -> Some x
  | _ :: t -> last t;;
val last : 'a list -> 'a option = <fun>
```

- **Devolver los dos últimos elementos de una lista:**

```
# let rec last_two = function
    | [] | [_] -> None
    | [x; y] -> Some (x,y)
    | _ :: t -> last_two t;;
val last_two : 'a list -> ('a * 'a) option = <fun>
```

- **Devolver el n-ésimo elemento de una lista:**

```
# let rec at k = function
    | [] -> None
    | h :: t -> if k = 0 then Some h else at (k - 1) t;;
val at : int -> 'a list -> 'a option = <fun>
```

- **Longitud de una lista:**

```
# let length list =
    let rec aux n = function
      | [] -> n
      | _ :: t -> aux (n + 1) t
    in
    aux 0 list;;
val length : 'a list -> int = <fun>
```

- **Invertir una lista:**

```
# let rev list =
    let rec aux acc = function
      | [] -> acc
      | h :: t -> aux (h :: acc) t
    in
    aux [] list;;
val rev : 'a list -> 'a list = <fun>
```

- **Saber si una lista es un palíndromo**

```
# let is_palindrome list =
    (* One can use either the rev function from the previous problem, or the built-in List.rev *)
    list = List.rev list;;
val is_palindrome : 'a list -> bool = <fun>
```

- **“Aplanar una lista”**

 # flatten [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"]

```
# type 'a node =
    | One of 'a
    | Many of 'a node list;;
type 'a node = One of 'a | Many of 'a node list
# (* This function traverses the list, prepending any encountered elements
    to an accumulator, which flattens the list in inverse order. It can
    then be reversed to obtain the actual flattened list. *);;
# let flatten list =
    let rec aux acc = function
      | [] -> acc
      | One x :: t -> aux (x :: acc) t
      | Many l :: t -> aux (aux acc l) t
    in
    List.rev (aux [] list);;
val flatten : 'a node list -> 'a list = <fun>
```

- **Eliminar duplicados de una lista**

```
# let rec compress = function
    | a :: (b :: _ as t) -> if a = b then compress t else a :: compress t
    | smaller -> smaller;;
val compress : 'a list -> 'a list = <fun>
```

- **Agrupar duplicados de una lista en sublistas**

```
# let pack list =
    let rec aux current acc = function
      | [] -> []    (* Can only be reached if original list is empty *)
      | [x] -> (x :: current) :: acc
      | a :: (b :: _ as t) ->
        if a = b then aux (a :: current) acc t
        else aux [] ((a :: current) :: acc) t  in
    List.rev (aux [] [] list);;
val pack : 'a list -> 'a list list = <fun>
```

- **Contar cuantos elementos hay iguales en una lista**

```
# let encode list =
    let rec aux count acc = function
      | [] -> [] (* Can only be reached if original list is empty *)
      | [x] -> (count + 1, x) :: acc
      | a :: (b :: _ as t) -> if a = b then aux (count + 1) acc t
                          else aux 0 ((count + 1, a) :: acc) t in
    List.rev (aux 0 [] list);;
val encode : 'a list -> (int * 'a) list = <fun>
```

```
# encode ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e"; "e"];;
- : (int * string) list =
[(4, "a"); (1, "b"); (2, "c"); (2, "a"); (1, "d"); (4, "e")]
```

- **Como el anterior pero diferente salida**

```
# encode ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e"; "e"];;
- : string rle list =
[Many (4, "a"); One "b"; Many (2, "c"); Many (2, "a"); One "d";
 Many (4, "e")]


# let encode l =
    let create_tuple cnt elem =
      if cnt = 1 then One elem
      else Many (cnt, elem) in
    let rec aux count acc = function
      | [] -> []
      | [x] -> (create_tuple (count + 1) x) :: acc
      | hd :: (snd :: _ as tl) ->
          if hd = snd then aux (count + 1) acc tl
          else aux 0 ((create_tuple (count + 1) hd) :: acc) tl in
      List.rev (aux 0 [] l);;
val encode : 'a list -> 'a rle list = <fun>
```

- **Al revés que la anterior, nos pasan cantidades y creamos la lista**

```
# let decode list =
    let rec many acc n x =
      if n = 0 then acc else many (x :: acc) (n - 1) x
    in
    let rec aux acc = function
      | [] -> acc
      | One x :: t -> aux (x :: acc) t
      | Many (n, x) :: t -> aux (many acc n x) t
    in
      aux [] (List.rev list);;
val decode : 'a rle list -> 'a list = <fun>
```

- **No creamos sublistas, solo contamos repetidos**

```
# let encode list =
    let rle count x = if count = 0 then One x else Many (count + 1, x) in
    let rec aux count acc = function
      | [] -> [] (* Can only be reached if original list is empty *)
      | [x] -> rle count x :: acc
      | a :: (b :: _ as t) -> if a = b then aux (count + 1) acc t
                              else aux 0 (rle count a :: acc) t
    in
```

```
   List.rev (aux 0 [] list);;
val encode : 'a list -> 'a rle list = <fun>
```

- **Duplicar elementos de una lista**

```
# let rec duplicate = function
    | [] -> []
    | h :: t -> h :: h :: duplicate t;;
val duplicate : 'a list -> 'a list = <fun>
```

- **Duplicar todos os elementos de una lista x número de veces**

```
# let replicate list n =
    let rec prepend n acc x =
      if n = 0 then acc else prepend (n-1) (x :: acc) x in
    let rec aux acc = function
      | [] -> acc
      | h :: t -> aux (prepend n acc h) t in
    (* This could also be written as:
       List.fold_left (prepend n) [] (List.rev list) *)
    aux [] (List.rev list);;
val replicate : 'a list -> int -> 'a list = <fun>
```

- **Elimina el elemento de la posición que le pasamos en la lista**

```
# let drop list n =
    let rec aux i = function
      | [] -> []
      | h :: t -> if i = n then aux 1 t else h :: aux (i + 1) t  in
    aux 1 list;;
val drop : 'a list -> int -> 'a list = <fun>
```

- **Dividir una lista en dos partes, la partimos en la posición que le pasamos**

```
# let split list n =
    let rec aux i acc = function
      | [] -> List.rev acc, []
      | h :: t as l -> if i = 0 then List.rev acc, l
                    else aux (i - 1) (h :: acc) t
    in
      aux n [] list;;
val split : 'a list -> int -> 'a list * 'a list = <fun>
```

- **Eliminar antes del primer valor y después del ultimo (solo dejar los valores entre los 2 que le pasamos)**

```
# let slice list i k =
    let rec take n = function
      | [] -> []
      | h :: t -> if n = 0 then [] else h :: take (n - 1) t
    in
    let rec drop n = function
```

```
    | [] -> []
    | h :: t as l -> if n = 0 then l else drop (n - 1) t
  in
  take (k - i + 1) (drop i list);;
val slice : 'a list -> int -> int -> 'a list = <fun>
```

- **Pasar los x primeros elementos de una lista para atrás**

```
# let rec remove_at n = function
  | [] -> []
  | h :: t -> if n = 0 then t else h :: remove_at (n - 1) t;;
val remove_at : int -> 'a list -> 'a list = <fun>
```

- **Insertar un elemento en una posición determinada**

```
# let rec insert_at x n = function
  | [] -> [x]
  | h :: t as l -> if n = 0 then x :: l else h :: insert_at x (n - 1) t;;
val insert_at : 'a -> int -> 'a list -> 'a list = <fun>
```

- **Crear una lista que contenga todos los enteros en un rango**

```
# let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a + 1) b
  in
    if a > b then List.rev (aux b a) else aux a b;;
val range : int -> int -> int list = <fun>
```

Recursiva de cola:

```
# let range a b =
  let rec aux acc high low =
    if high >= low then
      aux (high :: acc) (high - 1) low
    else acc
  in
    if a < b then aux [] b a else List.rev (aux [] a b);;
val range : int -> int -> int list = <fun>
```

- **Extraer x números aleatorios de una lista**

```
# let rand_select list n =
  let rec extract acc n = function
    | [] -> raise Not_found
    | h :: t -> if n = 0 then (h, acc @ t) else extract (h :: acc) (n - 1) t
  in
  let extract_rand list len =
    extract [] (Random.int len) list
  in
  let rec aux n acc list len =
    if n = 0 then acc else
```

```
      let picked, rest = extract_rand list len in
      aux (n - 1) (picked :: acc) rest (len - 1)
  in
  let len = List.length list in
    aux (min n len) [] list len;;
val rand_select : 'a list -> int -> 'a list = <fun>
```

- **Extrae n números aleatorios del 1 a M**

```
# (* [range] and [rand_select] defined in problems above *)
  let lotto_select n m = rand_select (range 1 m) n;;
val lotto_select : int -> int -> int list = <fun>
```

- **Generar una permutación aleatoria de elementos de una lista**

```
# let rec permutation list =
  let rec extract acc n = function
    | [] -> raise Not_found
    | h :: t -> if n = 0 then (h, acc @ t) else extract (h :: acc) (n - 1) t
  in
  let extract_rand list len =
    extract [] (Random.int len) list
  in
  let rec aux acc list len =
    if len = 0 then acc else
      let picked, rest = extract_rand list len in
      aux (picked :: acc) rest (len - 1)
  in
  aux [] list (List.length list);;
val permutation : 'a list -> 'a list = <fun>
```

- **Todas las combinaciones de n elementos de una lista en sublistas**

```
# let rec extract k list =
  if k <= 0 then [[]]
  else match list with
       | [] -> []
       | h :: tl ->
         let with_h = List.map (fun l -> h :: l) (extract (k - 1) tl) in
         let without_h = extract k tl in
         with_h @ without_h;;
val extract : int -> 'a list -> 'a list list = <fun>
```

- **Agrupar elementos de un conjunto en subconjuntos:**

```
# group ["a"; "b"; "c"; "d"] [2; 1];;
- : string list list list =
[[["a"; "b"]; ["c"]]; [["a"; "c"]; ["b"]]; [["b"; "c"]; ["a"]];
 [["a"; "b"]; ["d"]]; [["a"; "c"]; ["d"]]; [["b"; "c"]; ["d"]];
```

```
 [["a"; "d"]; ["b"]]; [["b"; "d"]; ["a"]]; [["a"; "d"]; ["c"]];
 [["b"; "d"]; ["c"]]; [["c"; "d"]; ["a"]]; [["c"; "d"]; ["b"]]]
```

```
let group list sizes =
    let initial = List.map (fun size -> size, []) sizes in
  let prepend p list =
    let emit l acc = l :: acc in
    let rec aux emit acc = function
      | [] -> emit [] acc
      | (n, l) as h :: t ->
        let acc = if n > 0 then emit ((n - 1, p :: l) :: t) acc
                  else acc in
        aux (fun l acc -> emit (h :: l) acc) acc t
    in
    aux emit [] list
  in
  let rec aux = function
    | [] -> [initial]
    | h :: t -> List.concat_map (prepend h) (aux t)
  in
  let all = aux list in
  (* Don't forget to eliminate all group sets that have non-full
     groups *)
  let complete = List.filter (List.for_all (fun (x, _) -> x = 0)) all in
    List.map (List.map snd) complete;;
val group : 'a list -> int list -> 'a list list list = <fun>
```

- **Ordenar una lista de listas según la longitud de las sublistas:**

```
let rec insert cmp e = function
  | [] -> [e]
  | h :: t as l -> if cmp e h <= 0 then e :: l else h :: insert cmp e t

let rec sort cmp = function
  | [] -> []
  | h :: t -> insert cmp h (sort cmp t)

(* Sorting according to length : prepend length, sort, remove length *)
let length_sort lists =
  let lists = List.map (fun list -> List.length list, list) lists in
  let lists = sort (fun a b -> compare (fst a) (fst b)) lists in
  List.map snd lists
;;


let rle list =
  let rec aux count acc = function
    | [] -> [] (* Can only be reached if original list is empty *)
    | [x] -> (x, count + 1) :: acc
```

```
    | a :: (b :: _ as t) ->
      if a = b then aux (count + 1) acc t
      else aux 0 ((a, count + 1) :: acc) t in
  aux 0 [] list

let frequency_sort lists =
  let lengths = List.map List.length lists in
  let freq = rle (sort compare lengths) in
  let by_freq =
    List.map (fun list -> List.assoc (List.length list) freq , list) lists in
  let sorted = sort (fun a b -> compare (fst a) (fst b)) by_freq in
  List.map snd sorted
```

- **Determinar si un entero dado es primo**

```
# let is_prime n =
    let n = abs n in
    let rec is_not_divisor d =
      d * d > n || (n mod d <> 0 && is_not_divisor (d + 1)) in
    n <> 1 && is_not_divisor 2;;
val is_prime : int -> bool = <fun>
```

- **Determinar el max comun divisor de 2 enteros positivos**

```
# let rec gcd a b =
    if b = 0 then a else gcd b (a mod b);;
val gcd : int -> int -> int = <fun>
```

- **Determinar si 2 enteros positivos son coprimos (max comun divisor=1)**

```
# (* [gcd] is defined in the previous question *)
  let coprime a b = gcd a b = 1;;
val coprime : int -> int -> bool = <fun>
```

- **Función totien de Euler (nº de enteros positivos coprimos con m)**

```
# (* [coprime] is defined in the previous question *)
  let phi n =
    let rec count_coprime acc d =
      if d < n then
        count_coprime (if coprime n d then acc + 1 else acc) (d + 1)
      else acc
    in
      if n = 1 then 1 else count_coprime 0 1;;
val phi : int -> int = <fun>
```

Mejora:

```
(* Naive power function. *)
let rec pow n p = if p < 1 then 1 else n * pow n (p - 1)

(* [factors] is defined in the previous question. *)
```

```
let phi_improved n =
  let rec aux acc = function
    | [] -> acc
    | (p, m) :: t -> aux ((p - 1) * pow p (m - 1) * acc) t
  in
    aux 1 (factors n)
```

- **Determinar los factores primos de un entero positivo dado**

```
# (* Recall that d divides n iff [n mod d = 0] *)
  let factors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then d :: aux d (n / d) else aux (d + 1) n
    in
      aux 2 n;;
val factors : int -> int list = <fun>
```

- **Igual que el anterior pero construye una lista que contiene los factores primos y su multiplicidad**

```
# let factors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then
          match aux d (n / d) with
          | (h, n) :: t when h = d -> (h, n + 1) :: t
          | l -> (d, 1) :: l
        else aux (d + 1) n
    in
      aux 2 n;;
val factors : int -> (int * int) list = <fun>
```

- **Dado un rango (limites), construye una lista de primos en ese rango**

```
# let is_prime n =
    let n = max n (-n) in
    let rec is_not_divisor d =
      d * d > n || (n mod d <> 0 && is_not_divisor (d + 1))
    in
      is_not_divisor 2

  let rec all_primes a b =
    if a > b then [] else
      let rest = all_primes (a + 1) b in
      if is_prime a then a :: rest else rest;;
val is_prime : int -> bool = <fun>
val all_primes : int -> int -> int list = <fun>
```

- **Función que dado un entero nos proporciona dos enteros primso cuya suma es el que le proporcionamos**

```
# (* [is_prime] is defined in the previous solution *)
  let goldbach n =
    let rec aux d =
      if is_prime d && is_prime (n - d) then (d, n - d)
      else aux (d + 1)
    in
      aux 2;;
val goldbach : int -> int * int = <fun>
```

- **Dado un rango de números enteros por su límite inferior y superior, imprima una lista de todos los números pares y su composición Goldbach (anterior)**

```
# (* [goldbach] is defined in the previous question. *)
  let rec goldbach_list a b =
    if a > b then [] else
      if a mod 2 = 1 then goldbach_list (a + 1) b
      else (a, goldbach a) :: goldbach_list (a + 2) b

  let goldbach_limit a b lim =
    List.filter (fun (_, (a, b)) -> a > lim && b > lim) (goldbach_list a b);;
val goldbach_list : int -> int -> (int * (int * int)) list = <fun>
val goldbach_limit : int -> int -> int -> (int * (int * int)) list = <fun>
```

- **Defina table de una manera que table variables expr devuelva la tabla de verdad para la expresión expr, que contiene las variables lógicas enumeradas en variables.**

```
# (* [val_vars] is an associative list containing the truth value of
    each variable.  For efficiency, a Map or a Hashtlb should be
    preferred. *)

  let rec eval val_vars = function
    | Var x -> List.assoc x val_vars
    | Not e -> not (eval val_vars e)
    | And(e1, e2) -> eval val_vars e1 && eval val_vars e2
    | Or(e1, e2) -> eval val_vars e1 || eval val_vars e2

  (* Again, this is an easy and short implementation rather than an
    efficient one. *)
  let rec table_make val_vars vars expr =
    match vars with
    | [] -> [(List.rev val_vars, eval val_vars expr)]
    | v :: tl ->
        table_make ((v, true) :: val_vars) tl expr
      @ table_make ((v, false) :: val_vars) tl expr

  let table vars expr = table_make [] vars expr;;
val eval : (string * bool) list -> bool_expr -> bool = <fun>
val table_make :
  (string * bool) list ->
  string list -> bool_expr -> ((string * bool) list * bool) list = <fun>
```

```
val table : string list -> bool_expr -> ((string * bool) list * bool) list =
  <fun>
```

- **Reglas de construcción de un código gris**

```
# let gray n =
    let rec gray_next_level k l =
      if k < n then
        let (first_half,second_half) =
          List.fold_left (fun (acc1,acc2) x ->
              (("0" ^ x) :: acc1, ("1" ^ x) :: acc2)) ([], []) l
        in
        gray_next_level (k + 1) (List.rev_append first_half second_half)
      else l
    in
      gray_next_level 1 ["0"; "1"];;
val gray : int -> string list = <fun>
```

- **Construir árbol binario balanceado para un número dado de nodos (x como info en cada nodo)**

```
# (* Build all trees with given [left] and [right] subtrees. *)
  let add_trees_with left right all =
    let add_right_tree all l =
      List.fold_left (fun a r -> Node ('x', l, r) :: a) all right in
    List.fold_left add_right_tree all left

  let rec cbal_tree n =
    if n = 0 then [Empty]
    else if n mod 2 = 1 then
      let t = cbal_tree (n / 2) in
      add_trees_with t t []
    else (* n even: n-1 nodes for the left & right subtrees altogether. *)
      let t1 = cbal_tree (n / 2 - 1) in
      let t2 = cbal_tree (n / 2) in
      add_trees_with t1 t2 (add_trees_with t2 t1 []);;
val add_trees_with :
  char binary_tree list ->
  char binary_tree list -> char binary_tree list -> char binary_tree list =
  <fun>
val cbal_tree : int -> char binary_tree list = <fun>
```

- **Árbol binario simétrico**

```
# let rec is_mirror t1 t2 =
    match t1, t2 with
    | Empty, Empty -> true
    | Node(_, l1, r1), Node(_, l2, r2) ->
      is_mirror l1 r2 && is_mirror r1 l2
    | _ -> false
```

```
  let is_symmetric = function
    | Empty -> true
    | Node(_, l, r) -> is_mirror l r;;
val is_mirror : 'a binary_tree -> 'b binary_tree -> bool = <fun>
val is_symmetric : 'a binary_tree -> bool = <fun>
```

- **Construir un árbol de búsqueda binaria a partir de una lista de enteros**

```
# let rec insert tree x = match tree with
    | Empty -> Node (x, Empty, Empty)
    | Node (y, l, r) ->
      if x = y then tree
      else if x < y then Node (y, insert l x, r)
      else Node (y, l, insert r x)
  let construct l = List.fold_left insert Empty l;;
val insert : 'a binary_tree -> 'a -> 'a binary_tree = <fun>
val construct : 'a list -> 'a binary_tree = <fun>
```

- **Paradigma de generación y prueba**

```
# let sym_cbal_trees n =
    List.filter is_symmetric (cbal_tree n);;
val sym_cbal_trees : int -> char binary_tree list = <fun>
```

- **Construir un árbol binario de altura equilibrada para una altura dada**

```
# let rec hbal_tree n =
    if n = 0 then [Empty]
    else if n = 1 then [Node ('x', Empty, Empty)]
    else
    (* [add_trees_with left right trees] is defined in a question above. *)
      let t1 = hbal_tree (n - 1)
      and t2 = hbal_tree (n - 2) in
      add_trees_with t1 t1 (add_trees_with t1 t2 (add_trees_with t2 t1 []));;
val hbal_tree : int -> char binary_tree list = <fun>
```

- **Nº mínimo de nodos**

```
# let rec min_nodes_loop m0 m1 h =
    if h <= 1 then m1
    else min_nodes_loop m1 (m1 + m0 + 1) (h - 1)
    let min_nodes h =
    if h <= 0 then 0 else min_nodes_loop 0 1 h;;
val min_nodes_loop : int -> int -> int -> int = <fun>
val min_nodes : int -> int = <fun>
```

- **Altura mínima**

```
# let min_height n = int_of_float (ceil (log (float(n + 1)) /. log 2.));;
val min_height : int -> int = <fun>
```

- **Contar las hojas de un árbol binario**

```
# let rec count_leaves = function
   | Empty -> 0
   | Node (_, Empty, Empty) -> 1
   | Node (_, l, r) -> count_leaves l + count_leaves r;;
val count_leaves : 'a binary_tree -> int = <fun>
```

- **Recopilar las hojas de un árbopl binario en una lista**

```
let leaves t =
  let rec leaves_aux t acc = match t with
    | Empty -> acc
    | Node (x, Empty, Empty) -> x :: acc
    | Node (x, l, r) -> leaves_aux l (leaves_aux r acc)
  in
  leaves_aux t [];;
val leaves : 'a binary_tree -> 'a list = <fun>
```

- **Nodos internos de un árbol binario en una lista**

```
let internals t =
  let rec internals_aux t acc = match t with
    | Empty -> acc
    | Node (x, Empty, Empty) -> acc
    | Node (x, l, r) -> internals_aux l (x :: internals_aux r acc)
  in
  internals_aux t [];;
val internals : 'a binary_tree -> 'a list = <fun>
```

- **Nodos de un nivel dado de una lista**

```
let at_level t level =
  let rec at_level_aux t acc counter = match t with
    | Empty -> acc
    | Node (x, l, r) ->
      if counter=level then
        x :: acc
      else
        at_level_aux l (at_level_aux r acc (counter + 1)) (counter + 1)
  in
    at_level_aux t [] 1;;
val at_level : 'a binary_tree -> int -> 'a list = <fun>
```

- **Construir un árbol binario completo**

```
# let rec split_n lst acc n = match (n, lst) with
   | (0, _) -> (List.rev acc, lst)
```

```
  | (_, []) -> (List.rev acc, [])
  | (_, h :: t) -> split_n t (h :: acc) (n-1)

let rec myflatten p c =
  match (p, c) with
  | (p, []) -> List.map (fun x -> Node (x, Empty, Empty)) p
  | (x :: t, [y]) -> Node (x, y, Empty) :: myflatten t []
  | (ph :: pt, x :: y :: t) -> (Node (ph, x, y)) :: myflatten pt t
  | _ -> invalid_arg "myflatten"

let complete_binary_tree = function
  | [] -> Empty
  | lst ->
    let rec aux l = function
      | [] -> []
      | lst -> let p, c = split_n lst [] (1 lsl l) in
              myflatten p (aux (l + 1) c)
    in
     List.hd (aux 0 lst);;
val split_n : 'a list -> 'a list -> int -> 'a list * 'a list = <fun>
val myflatten : 'a list -> 'a binary_tree list -> 'a binary_tree list = <fun>
val complete_binary_tree : 'a list -> 'a binary_tree = <fun>
```

- **Representación de cadena de árboles binarios**

```
# let rec string_of_tree = function
  | Empty -> ""
  | Node(data, l, r) ->
    let data = String.make 1 data in
    match l, r with
    | Empty, Empty -> data
    | _, _ -> data ^ "(" ^ (string_of_tree l)
            ^ "," ^ (string_of_tree r) ^ ")";;
val string_of_tree : char binary_tree -> string = <fun>
```

- **Conversión inversa**

```
# let tree_of_string =
  let rec make ofs s =
    if ofs >= String.length s || s.[ofs] = ',' || s.[ofs] = ')' then
      (Empty, ofs)
    else
      let v = s.[ofs] in
      if ofs + 1 < String.length s && s.[ofs + 1] = '(' then
        let l, ofs = make (ofs + 2) s in (* skip "v(" *)
        let r, ofs = make (ofs + 1) s in (* skip "," *)
          (Node (v, l, r), ofs + 1) (* skip ")" *)
      else (Node (v, Empty, Empty), ofs + 1)
  in
```

```
    fun s -> fst (make 0 s);;
val tree_of_string : string -> char binary_tree = <fun>
```

- **Preorder e inorder**

```
# let rec preorder = function
    | Empty -> []
    | Node (v, l, r) -> v :: (preorder l @ preorder r)
   let rec inorder = function
    | Empty -> []
    | Node (v, l, r) -> inorder l @ (v :: inorder r)
   let rec split_pre_in p i x accp acci = match (p, i) with
    | [], [] -> (List.rev accp, List.rev acci), ([], [])
    | h1 :: t1, h2 :: t2 ->
       if x = h2 then
         (List.tl (List.rev (h1 :: accp)), t1),
         (List.rev (List.tl (h2 :: acci)), t2)
       else
         split_pre_in t1 t2 x (h1 :: accp) (h2 :: acci)
    | _ -> assert false
   let rec pre_in_tree p i = match (p, i) with
    | [], [] -> Empty
    | (h1 :: t1), (h2 :: t2) ->
       let (lp, rp), (li, ri) = split_pre_in p i h1 [] [] in
         Node (h1, pre_in_tree lp li, pre_in_tree rp ri)
    | _ -> invalid_arg "pre_in_tree";;
val preorder : 'a binary_tree -> 'a list = <fun>
val inorder : 'a binary_tree -> 'a list = <fun>
val split_pre_in :
  'a list ->
  'a list ->
  'a -> 'a list -> 'a list -> ('a list * 'a list) * ('a list * 'a list) =
  <fun>
val pre_in_tree : 'a list -> 'a list -> 'a binary_tree = <fun>
```

- **Construcción de árboles a partir de una cadena de nodos**

```
let rec add_string_of_tree buf (T (c, sub)) =
   Buffer.add_char buf c;
   List.iter (add_string_of_tree buf) sub;
   Buffer.add_char buf '^'
  let string_of_tree t =
   let buf = Buffer.create 128 in
   add_string_of_tree buf t;
   Buffer.contents buf;;
val add_string_of_tree : Buffer.t -> char mult_tree -> unit = <fun>
val string_of_tree : char mult_tree -> string = <fun>
```

- **Contar los nodos de un árbol de múltiples vías**

```
# let rec count_nodes (T (_, sub)) =
   List.fold_left (fun n t -> n + count_nodes t) 1 sub;;
val count_nodes : 'a mult_tree -> int = <fun>
```

- **Determinar la longitud interna de un árbol (suma total de las longitudes de camino de todos los nodos del árbol)**

```
# let rec ipl_sub len (T(_, sub)) =
   (* [len] is the distance of the current node to the root.  Add the
      distance of all sub-nodes. *)
   List.fold_left (fun sum t -> sum + ipl_sub (len + 1) t) len sub
  let ipl t = ipl_sub 0 t;;
val ipl_sub : int -> 'a mult_tree -> int = <fun>
val ipl : 'a mult_tree -> int = <fun>
```

- **Secuencia de orden ascendente de los nodos del árbol (secuencia de abajo hacia arriba de los nodos del árbol)**

```
# let rec prepend_bottom_up (T (c, sub)) l =
   List.fold_right (fun t l -> prepend_bottom_up t l) sub (c :: l)
  let bottom_up t = prepend_bottom_up t [];;
val prepend_bottom_up : 'a mult_tree -> 'a list -> 'a list = <fun>
val bottom_up : 'a mult_tree -> 'a list = <fun>
```

- **Ruta de un nodo a otro**

```
 let neighbors g a cond =
   let edge l (b, c) = if b = a && cond c then c :: l
                    else if c = a && cond b then b :: l
                    else l in
   List.fold_left edge [] g.edges
  let rec list_path g a to_b = match to_b with
   | [] -> assert false (* [to_b] contains the path to [b]. *)
   | a' :: _ ->
     if a' = a then [to_b]
     else
       let n = neighbors g a' (fun c -> not (List.mem c to_b)) in
         List.concat_map (fun c -> list_path g a (c :: to_b)) n

  let paths g a b =
   assert(a <> b);
   list_path g a [b];;
val neighbors : 'a graph_term -> 'a -> ('a -> bool) -> 'a list = <fun>
val list_path : 'a graph_term -> 'a -> 'a list -> 'a list list = <fun>
val paths : 'a graph_term -> 'a -> 'a -> 'a list list = <fun>
```

- **Devuelve un camino cerrado (ciclo)**

```
# let cycles g a =
   let n = neighbors g a (fun _ -> true) in
```

```
  let p = List.concat_map (fun c -> list_path g a [c]) n in
  List.map (fun p -> p @ [a]) p;;
val cycles : 'a graph_term -> 'a -> 'a list list = <fun>
```

- **Problema de las 8 reinas**

```
# let possible row col used_rows usedD1 usedD2 =
  not (List.mem row used_rows
       || List.mem (row + col) usedD1
       || List.mem (row - col) usedD2)
       let queens_positions n =
  let rec aux row col used_rows usedD1 usedD2 =
    if col > n then [List.rev used_rows]
    else
      (if row < n then aux (row + 1) col used_rows usedD1 usedD2
       else [])
      @ (if possible row col used_rows usedD1 usedD2 then
           aux 1 (col + 1) (row :: used_rows) (row + col :: usedD1)
             (row - col :: usedD2)
         else [])
  in aux 1 1 [] [] [];;
val possible : int -> int -> int list -> int list -> int list -> bool = <fun>
val queens_positions : int -> int list list = <fun>
```

- **Palabras numéricas en inglés**

```
# let full_words =
  let digit = [|"zero"; "one"; "two"; "three"; "four"; "five"; "six";
               "seven"; "eight"; "nine"|] in
  let rec words w n =
    if n = 0 then (match w with [] -> [digit.(0)] | _ -> w)
    else words (digit.(n mod 10) :: w) (n / 10)
  in
    fun n -> String.concat "-" (words [] n);;
val full_words : int -> string = <fun>
```