PARADIGMAS DE LA PROGRAMACIÓN

EXAMEN PP FEBRERO 2008

1. Escribe el resultado tal y como lo haría OCaml:

- **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>

```
let rec p     = function      0 -> true        |n -> i(n-1)
and i         = function      0 -> false       |n -> p(n-1);;
   val p: int -> bool = <fun>
   val p: int -> bool = <fun>
```

```
p 2, i(-2);;
   (* recursividad infinita *)
```

```
let p n = p (abs n) and i n = i(abs n) in p 2, i(-2);;
   -: bool * bool = (true, false)
```

```
let x::y::Z = [1]@[2];;
   val x: int = 1
   val y: int = 2
   val z: int list = []
```

```
z::[];;
   -: int list list = [[]]
```

```
let x, y = y, x;;
   val x: int = 2
   val y: int = 1
```

```
let f z = z + 2 * y;;
   val f: int -> int = <fun>
```

```
f x + f y;;
   -: int = 7
```

```
let y = x + y;;
   val y: int = 3
```

```
f x + f y;;
   -: int = 9
```

```
let p = let x,y = x+y, x-y in y, x;;
   val p: int * int = (-1, 5)


let p = x+y, y-x in let x,y = p in y, x;;
   -: int * int = (1, 5)
```

2. Pasar la siguiente función sin referencias, de programación declarative a imperativa:

```
let f (x, y) =
   let x = abs x and y = abs y in
         let a = ref (max x y)
         and b = ref (min x y) in
               while !b <> 0 do
                     let temp = !a mod !b in a := !b;
                     b := temp
               done;
               !a;;

   let f (x, y) =
         let rec aux = function
                (a, 0) -> a
               |(a, b) -> aux(a*, a mod b)
         in aux( max (abs x) (abs y), min (abs x) (abs y));;
```

3.

```
type 'a arb = R of 'a
              |U of 'a * 'a arb
              |B of 'a * 'a arb * 'a arb;;

let anchura arbol =
   let rec aux = function
          []              -> []
         |R(x)::t         -> x:: aux t
         |U(x, i)::t      -> x:: aux (t@[i])
         |B(x, i, d)::t   -> x:: aux (t@[i;d])
   in aux[arbol];;
```

EXAMEN PP DICIEMBRE 2010

1. **Escribe el resultado de la compilación en ejecución de las siguientes frases, con tipos y valores, como lo haría el compilador interactivo de OCaml:**

```
let five _ = 5
  val five: 'a -> int = <fun>


let id x = x and apply x y = x y;;
  val apply: ('a -> 'b) -> 'a -> 'b


five 0, id 0, (id five) 0, id (five 0), apply five true;;
  -: int * int * int * int = (5, 0, 5, 5)


let mx3 x y z = max (max y z);;
  val mx3: 'a -> 'a -> 'a -> 'a = <fun> *


let rec fold x = function [] -> x | (op, y)::t -> fold (op x y) t;;
  'a -> (('a -> 'b -> 'a) * 'b) list *


fold 1 [(+), 2; (*), 3; (-) 1; (/) 3];;



let (|>) x f = f x;;



-2 |> abs |> (+) 3 |> function x -> x * x;;
```

2. **Reescribe las siguientes definiciones sin utilizar definiciones ni expresiones if…then…else:**

```
let f x = let (x, y) = x in x
  let f (a, _) = a;;
  (* val f: 'a * 'b -> 'a = <fun> *)

let n x g =
  if g x then true else false;;
  let n x g = (g x) = true;;
  (* val n: 'a -> ('a -> bool) -> bool = <fun> *)
```

3. Indica el tipo de las siguientes funciones:

```
let rec sorted = function
      [] | [_] -> true
     |x::y::z -> x <= y && sorted (y::z);;
```

```
val sorted: 'a list -> bool = <fun>
```

La función trabaja aparentemente con una lista devuelve valores boleanos, el problema viene en el tipo de la lista. Dado que las operaciones que vemos en la última línea se pueden aplicar a varios tipos de datos, tendremos que decantarnos por que sea una 'a list.
Verifica si los elementos de una lista están ordenados, trabajando de forma recursiva no terminal.

```
let rec merge l1 l2 = match (l1, l2) with
      (l, []) | ([], l) -> l
     |(h1::t1, h2::t2)  -> if h1 <= h2 then h1::merge t1 l2
                          Else h2::merge l1 t2;;
```

```
val merge: 'a list -> 'a list -> 'a list = <fun>
```

La función parte de 2 listas y en principio la única operación que encontramos es la igualdad en la sentencia condicional, por lo que de nuevo no podemos proveer un tipo concreto, pero sí que almas almacenan el mismo tipo de dato. La la salida ha de ser el mismo tipo que la entrada.
Mezcla estás ordenadas en una lista ordenada.

¿Tienen recursividad terminal? Si es así, realiza también una nueva definición sin recursividad terminal.

Ninguna de las dos es recursiva-terminal. Definiciones recirsivas-terminales:

```
let sorted l =
     let rec sorted máximo li = match li with
           [] -> true
          |h::t -> if maximo <= h then sorted h t
                   else false
     in sorted (List.hd l) l;;

let merge l1 l2 =
     let rec merge l1 l2 lf = match (l1, l2) with
           (l, []) | ([], l) -> lf @ l
          |(h1::t1, h2::t2) -> if h1 <= h2 then merge t1 l2 (lf@[h1])
                               Else merge l2 t2 (lf@[h2])
     in merge l1 l2 [];;
```

4. Considera la siguiente función en OCaml para el tipo de dato bitree (que sirve para representar árboles binarios en OCaml):

type bitree = Empty | Node of bitree * bitree;;

Decimos que un árbol binario es perfecto si tiene llenos todos sus niveles. Esto quiere decir que un árbol perfecto tendrá 2i no dos en el nivel i (para cada nivel i del árbol). Defina la función es_perfect0: bitre -> bool que indique si un árbol eso no perfecto.

```
let rec es_perfecto = function
      Empty
     |Node (Empty, Empty) -> true
     |Node (Empty, _) | Node(_, Empty) -> false
     |Node (i, d) -> es_perfecto i && es_perfecto d;;
```

EXAMEN PP FEBRERO 2007

```
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 doble x y = x (x y);;
      val doble: ('a -> 'a) -> 'a -> 'a = <fun>


let f = doble (function x -> x * x);;
      val f: int -> int = <fun> *


let x = f 2 in x + 1;;
      -: int = 17


let h f = function x -> let c::_ = f x in c;;
      val h: ('a -> 'b list) -> 'a -> 'b = <fun>


let s = h List.tl in s [1;2;3];;
      -: int = 2


let s l = h List.tl l
      val s: 'a list -> 'a = <fun>


let rec num x = function
       []   -> 0
      |h::t -> (if x = h then 1 else 0) + num x t;;
      val num 'a -> 'a list -> int = <fun> *


num "hola";;
      -: string list -> int = <fun>

let rec pre l s = match (l, s) with
       ([], _)  -> false
      |(_, tue) -> true
      |(h1::t1, h2::t2) -> h1 = h2 && pre t1 t2;;
      val pre: 'a list -> 'a list -> bool = <fun>


let l = ['1'; '2'; '3'] in pre l ['1'; '2'], pre l (List.tl l);;
      -: bool * bool = (true, false)
```

EXAMEN PP FEBRERO 2006

1. Escribe el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de OCaml:

```
let f, x = (+), 0;;
  val f = int -> int -> int = <fun>
  val x = 0;;

f x;;
  -: int -> int = <fun>

let y = x + 1, x - 1;;
  val = int * int = (1, -1)

let a, b = y in if a > b then b else a;;
  -: int = -1

let z = let a, b = y in let z = a - b in z * z;;
  val z: int = 4

(function x -> x) (function s -> s ^ s)
  -: String -> String = <fun>

let rec itera op = function
    []   -> ()
   |h::t -> op h; itera op t;;
  val itera: ('a -> 'b) -> 'a list -> unit *

let rec clist n x = if n < 1 then [] else x::clist (n-1);;
  val clist: int -> 'a -> 'a list = <fun>

clist 3 true;;
  -: bool list = [true; true; true]

let rec power n op x = if n <= 1 then x else power (n/2) op (op x x);;
  val power : int -> ('a -> 'a -> 'a) -> 'a -> 'a = <fun> *

let g = power 5 (+) in g 3
  -: int = 12
```

2. Considera la siguiente defínción en OCaml:

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

Escribe su tipo y una versión recursiva-terminal:

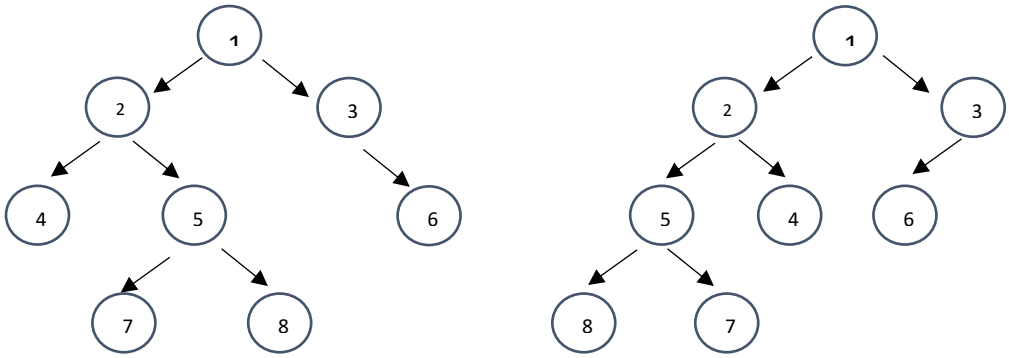
```
val comp: ('a -> 'a) list -> 'a -> 'a = <fun>

let compl l x =
  let rec aux res = function
        []    -> res
       |h::t  -> aux (h res) t
  in aux x (List.rev l);;  *
```

3. La función max_en está definida de forma que el valor max_en x l corresponde a la función de la lista l que alcanza el mayor valor en el punto x. Redefínela para optimizar su eficiencia:

```
let rec max_en x = function
    []    -> raise (Failure "max_es")
   |[f]   -> f
   |f::l  -> if f x > (max_en x l) x then f else max_en x l;;
```

```
let max_en x = function
    []    -> raise (Failure "max_es")
   |[f]   -> f
   |f::l  -> let res aux fmax vmax = function
                []    -> vmax
               |h::t  -> let hx = h x in
                                 if hx > vmax then aux h hx t
                                 else aux fmax vmax t
             in aux f (f x) l;;
```

4. Diremos que un árbol binario es un "giro" de otro árbol binario si el primero puede obtenerse del segundo intercambiando las ramas de cualquiera de sus nodos.
Así, por ejemplo, los d
os árboles siguientes son un "giro" del otro, pues el segundo se puede obtener a partir del primero, intercambiando las ramas de los nodos "2", "3" y "5".



Utilizando el tipo de dato 'a bitree:

```
type 'a bintree = Empty | Node of 'a bitree * 'a bitree;;
```

Implemente en un camión la función giro: 'a bitree -> 'a bitree -> bool que indique si un árbol es giro de otro.

```
let rec giro a1 a2 = match (a1, a2) with ->
      Empty, Empty -> true
     |Node (i1, r1, d1), Node(i2, r2, d2) -> r1 = r2 &&
                                             ((giro i1 i2 && giro d1 d2) ||
                                             (giro i1 d2 && giro d1 i2))
     | _ -> false;;
```

EXAMEN PP SEPTIEMBRE 2003

1. **Escribe el resultado de las siguientes frases, con tipos y valores, como lo indicaría el 'top level' de OCaml:**

```
let x, y = 1, 5;;
  val x: int = 1
  val y: int = 5

let x = let y = x < y in y;;
  val x: bool = true

let z = if x then y + 1 else y - 1;;
  val z: int = 6

let x y = y + 1 in x;;
  -: int -> int = <fun>

x;;
  val x: bool = true

let x y = y + 1;;
  val x: int -> int = <fun>

x y, x z;;
  -: int * int = 6, 7

let f = function f -> snd f, fst f;;
  val f: 'a * 'b -> 'b * 'a = <fun>

let x = f (y, z);;
  -: int * int = 6, 7

f x, x;;
  -: (int * int) * (int * int) = (5, 6), (6, 5)

let f p = f (f p) in f (1, 2);;
  -: int * int = 1, 2

f (1, 2);;
  -: int * int = 2, 1

let dos x y = x (x y y) y;;
  val dos: ('a -> 'a -> 'a) -> 'a -> 'a = <fun>

dos (+) 2, dos (*) 2;;
  -: int * int = 6, 8

function x -> function y ->  function z -> z y x;;
  -: 'a -> 'b -> ('b -> 'a -> 'c) -> c = <fun> *
```

2. Considere las siguientes definiciones en OCaml:

```
let mappar f (x, y) = f x, f y;;

let rec split f = function
    []   -> [], []
   |h::t-> let t1, t2 = split f t in
                        if f h then h::t1, t2
                        else t1, h::t2;;

let split f l = mappar List.rev (split f l);;
```

a. Indique el tipo de cada una de las funciones:

```
val mappar: ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>

val split: ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>

val split: ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
```

b. Escriba una definición terminal para la última definición de "split" sin usar el código anterior:

```
let split f l =
   let rec aux = function
         ([], l1, l2)    -> l1, l2
        |(h::t, l1, l2) -> if f h then aux(t, h:l1, l2) else aux(t, l1, h::l2)
   in aux (l, [], []);;
```

3. Indica el tipo de las funciones definidas en el siguiente código y luego simplifíquelo. Es decir, reescríbelo de la forma más breve posible:

```
let rec mx x y = if x > y = true then x else y;;
    and my x y = if x > y then true else false;;

let rec rollo f n l =
   if l = [] then n else
        let nn = f n (List.hd l)
        and nl = List.tl l in rollo f nn nl;;

val mx: 'a -> 'a -> 'a = <fun>
val my: 'a -> 'a -> bool = <fun>
val rollo: ('a -> 'b list -> 'a) -> 'a -> 'b list -> 'a

let mx = max and my = (>);;
let rollo = List.fold_left;; //
```

EXAMEN PP 30 DE ENERO 2004

1. **Escribe el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el toplevel de OCaml:**

```
let no f x = not (f x);;
  val no: ('a -> bool) -> 'a -> bool = <fun>

let par x = x mod 2 = 0 in no par;;
  -: int -> bool = <fun>

let rec rep n f x = if n > 0 then rep (n-1) f (f x) else x;;
  val rep: 'a -> ('a -> 'a) -> 'a -> 'a = <fun>

rep 3 (function x -> x * x) 2, rep 4 (function x -> 2 * x) 1;;
  -: int * int = (256 * 16)

let par x y = function z -> x z, y z in par ((+) 2) ((/) 2)) 3;;
  -: int * int = (5, 0)
```

2. **Dada la siguiente definición del tipo de dato 'a árbol:**

```
type 'a árbol = Vacío | Nodo of ('a * 'a árbol * 'a arbol);;
```

a. **Define una función 'a -> 'a árbol -> int que devuelva el número de nodos de un árbol etiquetados con un valor determinado:**

```
let rec cont x = function
    Vacío -> 0
  | Nodo (r, Vacío, Vacío)   -> if r = x then 1 else 0
  | Nodo (r, i, Vacío)       -> if r = x then 1 + cont x i else cont x i
  | Nodo (r, Vacío, d)       -> if r = x then 1 + cont x d else cont x d
  | Nodo (r, i, d)           -> if r = x then 1 + cont x i + cont x d
                                else cont x i + cont x d;;
```

b. **Define una función subst: 'a -> 'a -> 'a arbol de forma que subst x y sea la función que al aplicarla a un árbol, devuelve un árbol igual al original salvo los nodos que tienen valor x, que los cambia por y:**

```
let rec subst x y = function
    Vacío ->
  | Nodo (r, Vacío, Vacío)   -> if r = x then Nodo (y, Vacío, Vacío)
                                else Nodo (r, Vacío, Vacío)
  | Nodo (r, i, Vacío)       -> if r = x then Nodo (y, subst i, Vacío)
                                else Nodo (r, subst i, Vacío)
  | Nodo (r, Vacío, d)       -> if r = x then Nodo (y, Vacío, subst d)
                                else Nodo (r, Vacío, subst d)
  | Nodo (r, i, d)           -> if r = x then Nodo (y, subst i, subst d)
                                else Nodo (r, subst i, subst d);;
```

3. **Defina una función l_ordenada: ('a -> 'a -> bool) -> 'a list -> bool, de forma que si f es una relación de orden en el tipo 'a (esto es, una función que dice si dos elementos de tipo 'a están ordenados), l_ordenada f sea la función que diga si una lista está ordenada según f:**

```
(* igual a sorted_by *)

    let l_ordenada r l = match l with
           a::b::t -> if r a b then l_ordenada r (b::t) else false
         | _ -> true;;
```

4. **Defina utilizando recursividad terminal una función l_max: 'a list -> 'a que devuelva de cada lista el mayor de sus elementos:**

```
let l_max = function
    []       -> raise (Failure "l_max")
  | h::t     -> let rec aux x = function
                      [] -> x
                    | h::t -> if x > h then aux x t else aux h t
                in aux h t;;
```

EXAMEN PP SEPTIEMBRE 2004

1. **Escribe el resultados de la compilación y ejecución de las siguientes frases como lo haría el toplevel de OCaml:**

```
let apa x f = f x;;
  val appa: 'a -> ('a -> 'b) -> 'b;;

List.map (apa 2) [(function x -> x * x); succ; (+)1; (-)1];;
  -: int list = [4; 3; 3; 1]

let apa_rep n x f =
  let rec aux x = function     0 -> x
                             | n -> aux (apa x f) (n-1)
  in aux (abs n);;
  val apa_rep: int -> 'a -> ('a -> 'a) -> 'a = <fun>

apa_rep (-2) "x" (function x -> x ^ x);;
  -: string "xxxx"

let fop op f g = function y -> op (f y) (g y);;
  val fop: ('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> 'd -> 'c = <fun>

let suma = fop (+);;
  val suma: ('a -> int) -> ('a -> int) -> 'a -> int

let f = let f1 x = x * x in
            let f2 x = f1 x * x in
                 suma f1 f2
      in f 2;;
  -: int = 12
```

2. **Redefine la función f de modo que solo utilice recursividad terminal:**

```
let rec f orden = function
    []       -> raise (Failure "f")
  | h::t     -> let m = f orden t in
                if orden h m then h else m;;

let f orden = function
    []       -> raise (Failure "f")
  | h::t     -> let rec aux x = function
                    []       -> x
                  | h::t     -> if orden x h then aux x t
                                else aux h t
             in aux h t;;
```

3. **Una relación en un conjunto A puede representarse como una función de (AxA) -> bool, que indica para cada pareja de elementos de A si están relacionados o no. Dada una función f cualquiera: A -> B, puede hablarse de la relación de equivalencia que induce sobre el conjunto A como aquella en la que son equivalentes los elementos de la misma imagen.**

   a. **Define en OCaml una función rel_eq: ('a -> 'b) -> 'a * 'a -> bool que para cualquier función devuelva la relación de equivalencia inducida por ella en el sentido señalado:**

   ```
   let rel_eq f (x, y) = f x = f y;; *
   ```

   b. **Define en OCaml una función**

   clases_eq: ('a * 'a -> bool) -> 'a lisy -> 'a list list

   de modo que, cada una relación de equivalencia r sobre un conjunto (tipo de dato) A y dada una lista de elementos de A, "divida" los elemento de la lista en clases de equivalencia inducidas por la relación r.

   ```
   let rec clases_eq r = function
     let rec añadir x = function
            []              -> [[x]]
          | (h::t)::clases-> if r (x, h) then (x::h::t)::clases
                             else (h::t)::añadir x clases
     in fuction
            []       -> []
          | h::t     -> añadir h (clases_eq r t);;
   ```

EXAMEN PP FEBRERO 2003

1. **Indique la respuesta del compilador 'toplevel' de OCaml a las siguientes sentencias:**

```
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)

let p = let x, y = p in y, x;;
  val p: float * float = (-3, 2)

let x = x > y and y = x
  val x: bool  = false
  val y: float = -2.5

let rec map2 f1 f2 = function
    []      -> []
  | h::t    -> f1 h::map f2 f1 t;;
  val map2: ('a -> 'b)->( 'a -> 'b)-> 'a list -> 'b list = <fun>

let rec f = function x -> x * x and g x = f (x - 1) + x
in map2 f g [1; 2; 3; 4; 5];;
  -: int list = [1; 3; 9; 13; 25]
```

2. **Define una función**

imprime_inverse: int int list -> unit

Que visualice por la salida estándar los elementos de una lista de enteros en orden inverso, a 1 por línea:

```
let rec imprime_inversa = function
    []      ->   ()
  | h::t    ->   imprime inversa t;
                 print_endline (string_of_int h);;
```

3. Observa la siguiente definición de la función fold_right en el módulo List de OCaml Y realice una nueva implementación que sea recursiva terminal:

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

```
let fold_right f l e =
   let rec aux = function
          ([], r)        -> r
        | (h::t, r)      -> aux (t, f h r)
   in aux (List.rev l, e)
```

4. Considera la siguiente definición del tipo de dato 'a tree que podría servir para representar en OCaml cierto tipo de árboles binarios:

```
type 'a tree = Leaf of 'a | Node of ('a tree * 'a * 'a tree);;
```

Llamaremos "caminos de un árbol" a cada uno de los recorridos descendentes desde la raíz a cada una de las hojas. Si tenemos un árbol con valores numéricos asociados a los nodos, diremos que el "peso" de un camino es la suma de los valores de los nodos que lo componen.

a. Define la función "peso_máximo: float tree -> float" Que devuelva el valor del camino (o caminos) de peso máximo de un árbol:

```
let rec peso_maximo = function
     Leaf p         -> p
   | Node (i, c, d)-> c +. max (peso_maximo i) (peso_maximo d);;
```

b. Define una función "caminos: 'a tree -> 'a list -> 'a list list" que devuelva todos los caminos de un árbol de izquierda a derecha:

```
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);;
```

c. Define la función "camnino_maximo: float tree -> char list" Que describa un camino de peso máximo en cada árbol dado:

```
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);;
```

EXAMEN PP FEBRERO 2005

1. **Escribe el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, cómo lo indicaría el 'toplevel' de OCaml:**

```
let x = let x = 0 in x, x+1, x+2;;
      val x: int * int * int = (0,1,2)

let par = let f x = 2 * x in let g x = f x * x in (f, g)
      val par: (int -> int) * (int -> int) = (<fun>, <fun>)

let doble p x = fst p x, snd p x;;
      val doble: ('a -> 'b) * ('a -> 'c) -> 'a -> ('b * 'c)

let rap = doble par in rap 2;;
      -: int * int = (4, 8)

let rec g l x = match l with [] -> x | h::t -> g t (h x);;
      val g: ('a -> 'a) list -> 'a -> 'a

let rec suces gen x0 = function
        0 -> []
      | n -> x0::suces gen (gen x0) (n-1)
      val suces: ('a -> 'a) -> 'a -> 'a list = <fun>

let lista = suces ((+) 1) 0 5;;
      val lista: int list = [0; 1; 2; 3; 4]

g (List.map (+) lista) 10;;
      -: int = 20
```

2. **Realiza una nueva definición para la función sumpro definida a continuación, de forma que solo utilice recursividad terminal**

```
let rec sumpro n =
   if n < 1 then (0, 1)
   else let (s, p) = sumpro (n-1) in (s+n, p*n);;

let sumpro n =
   let rec aux (x, i, j) =
        if (x < 1) then (i, j)
        else aux (x-1, i+x, j*x)
   in aux (n, 0, 1);;
```

3. **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. Ejemplos:**

```
aBase 10 1234 = [4;3;2;1]
aBase 100 1234 = [34; 12]
aBase 2 11 = [1;1;0;1]
aBase 3 11 = [2;0;1]
aBase 16 200 = [8;12]
```

```
deBase 10 [4;3;2;1] = 1234
deBase 100 [34; 12] = 1234
deBase 2 [1;1;0;1]= 11
deBase 3 [2;0;1] = 11
deBase 16 [8;12] = 200
```

```
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;;
```

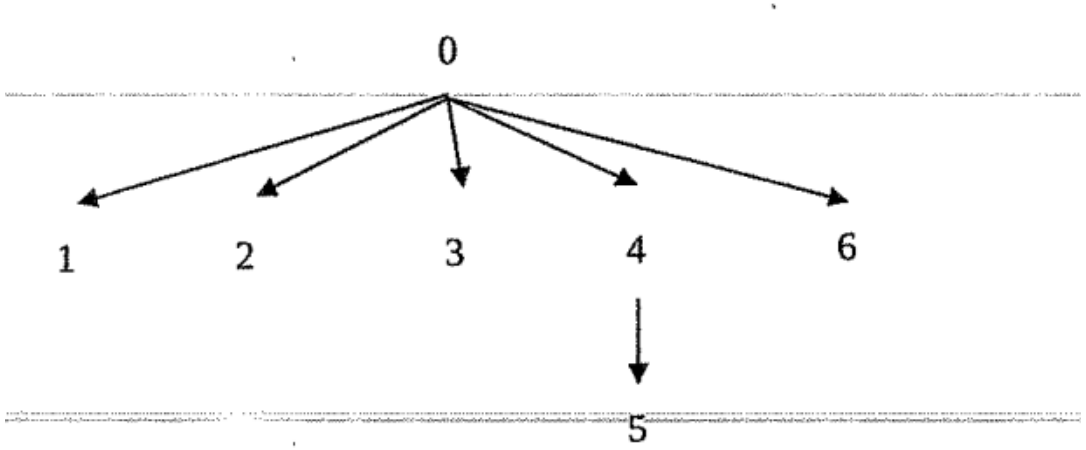
4. Dada la siguiente definición en OCaml para los tipos de dato 'a arbolgen, que sirven para representar árboles con nodos etiquetados con valores de tipo 'a, en los que cada nodo puede tener cualquier número de ramas:

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

De modo que por ejemplo, el valor ag1 definido por:

```
let ag1 = Nodo (0, [Nodo [1, []);
                    Nodo [2, []);
                    Nodo [3, []);
                    Nodo [4, [Nodo(5, [])]);
                    Nodo [6, []))
```

Correspondería al árbol:



Define la función hojas: 'a arbolgen -> 'a list
que devuelva una lista de valores asociados a las hojas del árbol, de izquierda a derecha:

```
let rec hojas = function
        Nodo (r, []) -> [r]
      | Nodo (r, [h::t] -> if t = [] then hojas h
                           else hojas h @ hojas Nodo(r, i);;
```