

PROGRAMACIÓN DECLARATIVA

6 DE SEPTIEMBRE DE 2003

1. (5 ptos.) Escriba el resultado de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" 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;;
```

```
- : 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);;
```

```
val x : int * int = 6, 5
```

```
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. (2'5 ptos.) 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 definidas con ese código.

```
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. (2'5 ptos.) Indique el tipo de cada una de las funciones definidas en el siguiente fragmento de código ocaml y, luego, simplifíquelo. Es decir, reescribalo de la forma más breve posible, de modo que todas las definiciones resulten totalmente equivalentes a las dadas. (Puede utilizar cualquier valor predefinido por el compilador de ocaml).

```
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 -> 'a) -> 'a -> 'b list -> 'a = <fun>  
  
let mx = max and my = (>);;
```

```
let rollo = List.fold_left;;
```

PROGRAMACIÓN DECLARATIVA
11 de febrero de 2003

NOMBRE: _____ DNI: _____

I.I. I.T.I.G.

1. (4 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el "toplevel" de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

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 :: map2 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. (1 pto.) Defina una función “`imprime_inversa: int list -> unit`” que “visualice” por la salida estándar los elementos de una lista de enteros en orden inverso y uno por línea.
Así, por ejemplo, al evaluar la expresión “`imprime_inversa [1;2;3]`” debería aparecer por la salida estándar:

3
2
1

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

3. (2 ptos.) Observe la siguiente definición de la función `fold_right` del módulo `List` de caml y realice una nueva definición que sea recursiva terminal.

```
let rec fold_right f l e = match l with
  [] -> e
  | h::t -> f h (fold_right f t 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. (3 ptos.) Considere 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 todos los nodos que lo componen.

Así, por ejemplo, en el siguiente árbol el **peso máximo** de todos sus caminos es 15.

a) Defina una función “**peso_maximo: float tree -> float**” que devuelva el valor del camino (o los 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) Defina una función “**caminos: 'a tree -> 'a list list**” que devuelva todos los caminos de un árbol de izquierda a derecha, de forma que, por ejemplo, para el árbol del dibujo dé la lista [[3;11]; [3;4;-2;10]; [3;4;-2;9]; [3;4;8]].

```

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) Un camino dentro de un árbol, puede indicarse enumerando las ramas que hay que escoger en cada nodo para seguirlo desde la raíz hasta la hoja. Si elegimos la letra 'I' para referirnos a las ramas izquierdas y 'D' para las derechas, las listas ['D'; 'I'; 'I'] y ['D'; 'D'] describen ambas caminos de peso máximo en el árbol del dibujo.

Defina una función “`camino_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);;

```

NOTA: Las definiciones de los ejercicios 2, 3 y 4 deben realizarse de la forma más sencilla posible.

PROGRAMACIÓN DECLARATIVA

30 DE ENERO DE 2004

PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

I.I.

I.T.I.G

1. (2,5 puntos) Escriba 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);;

let par x = x mod 2 = 0 in no par;;

let rec rep n f x = if n > 0 then rep (n-1) f (f x) else x;;

rep 3 (function x -> x * x) 2, rep 4 (function x -> 2 * x) 1;;

(let par x y = function z -> x z, y z in par ((+) 2) ((/) 2)) 3;;

2. (2 puntos) Dada la siguiente definición del tipo de dato '*a arbol*', que sirve para representar cierto tipo de árboles binarios

type 'a arbol = Vacio | Nodo of ('a * 'a arbol * 'a arbol);;

- a. defina una función *cont:'a -> 'a arbol -> int*, que devuelva el número de nodos de un árbol que están etiquetados con un valor determinado.

b. defina una función $\text{subst}: 'a \rightarrow 'a \rightarrow 'a \text{ arbol} \rightarrow 'a \text{ arbol}$, de forma que $\text{subst } x \ y$ sea una función que, al aplicarla a un árbol, devuelva un arbol igual al original salvo los nodos que tuviesen valor x , que tendrán valor y .

3. (1 punto) Defina una función $\text{l_ordenada}: ('a \rightarrow 'a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow \text{bool}$, de forma que si f es una relación de orden en el tipo $'a$ (esto es, una función que dice si dos elementos de este tipo están ordenados), $\text{l_ordenada } f$ sea la función que dice si una lista está ordenada según el orden f .

4. (1 punto) Defina utilizando exclusivamente recursividad terminal una función $\text{l_max}: 'a \text{ list} \rightarrow 'a$ que devuelva de cada lista el mayor de sus elementos.

PROGRAMACIÓN DECLARATIVA
3 DE FEBRERO DE 2005

NOMBRE: _____

I.I. I.T.I.G

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *toplevel* de *ocaml*:

let x = let x = 0 in x, x+1, x+2;;

let par = let f x = 2 * x in let g x = f x * x in (f,g);;

let doble p x = fst p x, snd p x;;

let rap = doble par in rap 2;;

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

let rec suces gen x0=
 function 0 -> [] | n -> x0 :: suces gen (gen x0) (n-1);;

let lista = suces ((+) 1) 0 5;;

g (List.map (+) lista) 10;;

2. (2 puntos) Realice una nueva definición para la función *sumpro* definida a continuación, de forma que sólo se 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);;
```

3. (2 puntos) Defina 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* (de forma que la cabeza de la lista corresponda al dígito menos significativo, y una función *deBase*: *int -> int list -> int* tal que *deBase b l* devuelva el número entero cuya representación en base *b* corresponde a la lista *l* (es decir, de forma que *deBase b* sea la función inversa a *aBase b*). El parámetro *b* que indica la base será siempre un número entero estrictamente mayor que 1, el número *n* será siempre un entero positivo y los elementos de la lista *l* serán siempre enteros no negativos estrictamente menores que *b*).

Por tanto, estas definiciones deberán comportarse de forma que las siguientes expresiones de tipo *bool* tengan todas valor *true* en *ocaml*:

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

y $\text{deBase } b \ (\text{aBase } b \ n) = n$, siempre que b y n cumplan los requisitos arriba mencionados.

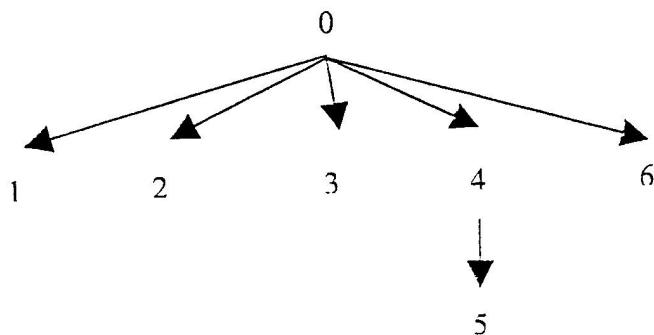
4. (2 puntos) Dada la siguiente definición en *ocaml* para los tipos de datos '*a arbolgen*' (que sirven para representar árboles con nodos etiquetados con valores de tipo '*a*', en los que de cada nodo puede colgar cualquier número de ramas)

```
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



defina una función *hojas* : '*a arbolgen* -> '*a list*
que devuelva la lista de valores asociados a las hojas del árbol, de izquierda a
derecha, de forma que *hojas ag1* = [1; 2; 3; 5; 6]

PROGRAMACIÓN DECLARATIVA
16 DE DICIEMBRE DE 2005

NOMBRE: _____

I.I.O

I.T.I.G.O

1. (5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de *ocaml*:

```
let id = function x -> x;;
```

```
let cte k = function _ -> k;;
```

```
(cte 0) "a";;
```

```
let rec genlist f = function
  0 -> []
  | n -> (genlist f (n-1)) @ [f n];;
```

```
let l1,l2 = let dela n = genlist id n
            in dela 4, dela 5;;
```

```
let l3 = let rep x n = genlist (cte x) n in rep 5 2;;
```

```
let rec reduce f e = function [] -> e | h::t -> f h (reduce f e t);;
```

```
let sigma = reduce (+) 0;;
```

```
let pi = reduce (*) 1;;
```

```
let l4 = (List.map sigma [l1; l2; l3]) in (sigma l4, pi l4);;
```

2. (1 punto) Considere la siguientes definiciones escritas en ocaml

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

let rec g = function
    Empty -> 0
  | Node (r,i,d) -> if g d > 2 * g i then r + g d / 2
                        else r + g i / 3;;
```

La definición de la función *g* contiene un error de diseño que no afecta al resultado de la función, pero sí gravemente a la eficiencia del cálculo.

Redefina la función *g*, cambiando su definición lo menos posible, de forma que se corrija ese problema de eficiencia.

3. (2 puntos) Escriba una definición alternativa para la función *f*, de modo que sólo se utilice recursividad terminal

```
let rec f = function
    0 -> 0
  | 1 -> 1
  | n -> if n > 0 then 2*f(n-1) - 3*f(n-2)
            else raise (Failure "f");;
```



4. (2 puntos) Defina una función **criba** : ('a -> bool) list -> 'a list list, de forma que **criba** [$p_1; p_2; \dots; p_n$] l devuelva una lista de listas cuyo primer elemento sea la lista de elementos de l en los que se cumple el predicado p_1 ; el segundo, la lista de elementos de l que no cumplen p_1 , pero sí p_2 ; ...; el penúltimo, la lista de elementos de l que cumplen el predicado p_n , pero ninguno de los anteriores; y el último, la lista de elementos de l que no cumplen ninguno de los predicados. En cada una de estas listas los elementos deben conservar entre sí el mismo orden relativo que tenían dentro de l .

Así, por ejemplo, ha de verificarse que

```
criba [(function x -> x mod 2 = 0); (function x -> x mod 3 = 0); (<)
0]
[-10;-9;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9;10] =
[[[-10; -8; -6; -4; -2; 0; 2; 4; 6; 8; 10];
[-9; -3; 3; 9];
[1; 5; 7];
[-7; -5; -1]]]
```


PROGRAMACIÓN DECLARATIVA
14 DE FEBRERO DE 2006

NOMBRE: _____

I.I.

I.T.I.G

1. (5.5 puntos) Escriba 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 : int = 0
f x;;
- : int -> int = <fun>
```

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

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

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

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

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



2. (1.5 puntos) Considere la siguiente definición en ocaml:

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

¿Cuál es el tipo de la función *comp*?

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

Realice una nueva definición para *comp* de forma que sólo se utilice recursividad terminal.

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

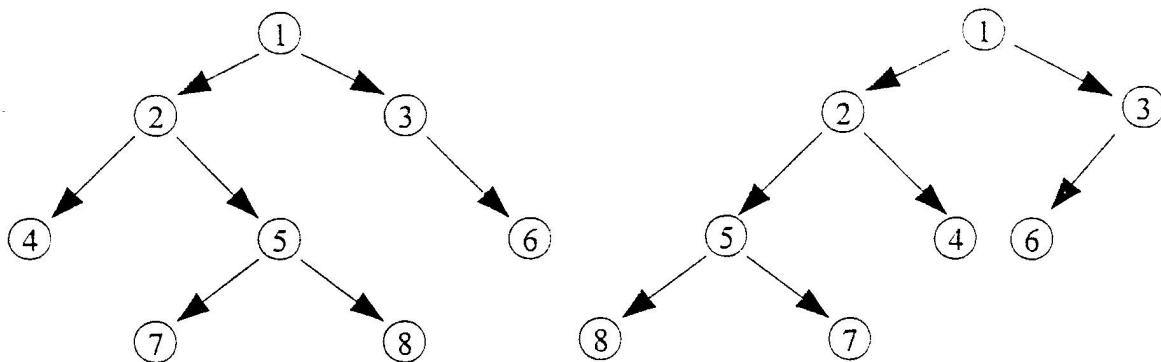
3. (1.5 puntos) La función *max_en* está definida de forma que el valor de *max_en x l* corresponde a la función de la lista *l* que alcanza el mayor valor en el punto *x*. Redefinala para optimizar su eficiencia.

```
let rec max_en x = function
  [] -> raise (Failure "max_en")
  | [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_en")
  | f::l -> let rec aux fmax vmax = function
    [] -> fmax
    | 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. (1.5 puntos) Diremos que un árbol binario es un "giro" de otro árbol binario si el primero puede obtenerse del segundo intercambiando las ramas de cualesquiera de sus nodos.

Así, por ejemplo, los dos árboles siguientes son el uno "giro" del otro, pues el segundo se puede obtener a partir del primero intercambiando las ramas de los nodos "2", "3", y "5".



Utilizando, para representar los árboles binarios, el tipo de dato '*a bintree*' definido a continuación, implemente en ocaml una función *giro*: '*a bintree* -> '*a bintree* -> *bool* que indique si dos árboles son el uno giro del otro.

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

```
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;;
```

PROGRAMACIÓN DECLARATIVA

13 DE SEPTIEMBRE DE 2006

NOMBRE: _____

I.I.

I.T.I.G

1. (6 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *top level* de *ocaml*:

let x, y = 2, 5;;

- val x : int = 2
- val y : int = 5

let f y = x + y;;

- val f : int → int = <fun>

f (let f = function x -> x * x in f x);;

- : int = 6

let v x f = f x;;

- val v : 'a → ('a → 'a) → 'a = <fun>

let cero x = v 0 x and dos x = v 2 x;;

- val cero : (int → 'a) → 'a = <fun>
- val dos : (int → 'a) → 'a = <fun>

```
let g = cero (-) in g 1;;
```

```
- : int = -1
```

```
let h = dos (+);;
```

```
val h : int → int = <fun>
```

```
dos h;;
```

```
- : int = 4
```

```
g 0, h 0;;
```

Error. q no está definida

Unbound value q

```
let doble = let vacia = "" in function
  vacia → vacia
  | s → s ^ s;;
```

```
val doble : string → string = <fun>
```

```
doble "hola";;
```

```
- : string = "hola"
```

```
let rec ap = function
```

```
[] → 0 | h :: t → h (ap t);;
```

```
val ap : (int → int) list → int = <fun>
```

2. (2 puntos) Realice una nueva definición para la función f , de forma que sólo se utilice recursividad terminal.

```
let rec f = function
  [] -> 0
  | h::t -> h + 2 * f t;;
```

```
let f l =
  let rec aux = function
    ([] , x) -> x
    | (h::t , x) -> aux (t , h+2 * x)
  in aux (list . rev l , 0 );;
```

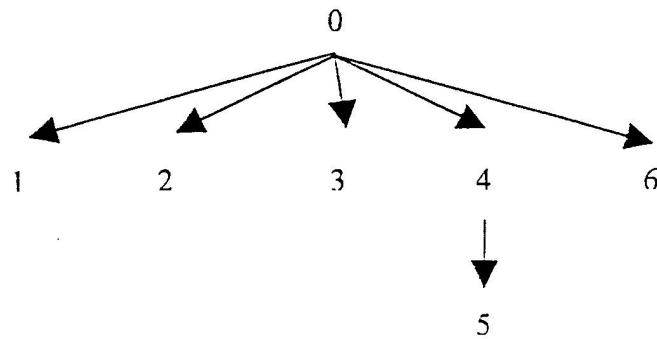
3. (2 puntos) Considere la siguiente definición en *ocaml* para el tipo de dato ‘*a arbolgen*’ (que sirve para representar árboles con nodos etiquetados con valores de tipo ‘*a*’, en los que de cada nodo puede colgar cualquier número de ramas)

```
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



Defina una función *nnodos* : '*a arbolgen* -> *int*
que devuelva el número de nodos de un árbol, de forma que, por ejemplo,
nnodos ag1 = 7

PROGRAMACIÓN DECLARATIVA

22 de diciembre 2006

NOMBRE: _____
I.I. I.T.I.G

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *toplevel* de *ocaml*:

let x, y = "hola", "adiós";;

let x y = x ^ y;;

let x = x "pepe" in x;;

let x::y::z = [1] @ [2] @ [3];;

let x::y::z = [1] :: [2] :: [3] :: [];;

let rec m2 l1 l2 = match (l1, l2) with
([], []) -> [] | (a::b, c::d) -> a c :: m2 b d;;

```
m2 [abs] [1; -2];;
```

```
(function x -> x (x 2)) (function x -> 2 * x * x);;
```

2. Diremos que una lista es *sub-lista* de otra si puede obtenerse eliminando algunos elementos de esta. Puesto que cada elemento de una lista de n elementos puede ser o no eliminado para obtener una sub-lista, esta tendrá 2^n sub-listas. Así, por ejemplo, la lista [3;5;3] tendría las siguientes sub-listas: [], [3], [5], [5; 3], [3], [3; 3], [3; 5] y [3; 5; 3]. (Nótese que alguna sub-lista aparece más de una vez debido a que puede ser obtenida eliminando distintos elementos de la lista original).

a. (2 puntos) Defina una función *sublistas* : 'a list -> 'a list list que dé todas las sub-listas de una lista dada. (Si lo desea puede optar por no incluir repeticiones en la lista de sub-listas; pero en ese caso debe indicarlo explícitamente).

b. (2 puntos) Defina una función `sublista_de` : '`a list -> 'a list -> bool`, tal que `sublista_de l1 l2` indique si `l2` es, o no, *sub-lista de l1*. (Se tendrá en cuenta la eficiencia de la definición).

3. (2 puntos) Defina, utilizando sólo **recursividad terminal**, una función `apariciones`: '`a -> 'a list -> int` que devuelva el número de veces que aparece un valor en una lista.

PROGRAMACIÓN DECLARATIVA

5 de septiembre de 2002

NOMBRE: _____ DNI: _____

I.I. I.T.I.G

Programación funcional

1. (2 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocamli. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

let x = 0;;

val x : int = 0

(let x = x + 1 in 2 * x), 2 * x;;

x;;

let f x = x, x in f x;;

let doble f = function x -> f (f x);;

2. Dada la siguiente definición:

```
# let rec ffac = function
  0 -> 1 | 1 -> 1
  | n -> n * ffac (n-2);;
```

- a) (0.5 ptos.) Indique su tipo:

val facc :

- b) (0.5 ptos.) Indique el tipo y el valor de la siguiente expresión:

ffac 6;;

- :

c) (1 pto.) Redefina la función **ffac** utilizando sólo recursividad terminal:

3. La siguiente definición sirve para representar árboles binarios:

```
type arbb = Vacio | Nodo of arbb * arbb;;
```

- a) (1 pto.) Defina una función **altura** : **arbb -> int** que, para cada árbol binario, devuelva su altura. Llamamos "altura" de un árbol a la distancia de la raíz a la hoja más lejana (contada como el número de nodos que hay que recorrer para llegar hasta ella). (La altura del árbol vacío es 0 y la de un árbol que sólo tenga raíz es 1).

- b) Defina una función **tamanno** : **arbb -> int** que, para cada árbol binario, devuelva su tamaño. Llamamos "tamaño" de un árbol al número de nodos que contiene. (El tamaño del árbol vacío es 0 y el de un árbol que sólo tenga raíz es 1).

PROGRAMACIÓN DECLARATIVA

8 DE SEPTIEMBRE DE 2004

PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

I.I.

I.T.I.G

1. (3,5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

```
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 apa_rep n x f =
  let rec aux x = function 0 -> x
    | n -> aux (apa x f) (n-1)
  in aux x (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 = <fun>
```

```
let f = let f1 x = x * x in
  let f2 x = f1 x * x in
  suma f1 f2
in f 2;;
```

```
- : int = 12
```

2. (1 punto) Redefina la función f de modo que sólo se utilice recursividad terminal (pero debe dar siempre el mismo resultado que la original).

```
let rec f orden =
  function [] -> raise (Failure "f") | [x] -> x
    | 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. (2 puntos) Una relación (de equivalencia, de orden, etc...) en un conjunto A puede representarse como una función de $(A \times A) \rightarrow \text{bool}$, que indica para cada pareja de elementos de A si están o no relacionados. Dada una función cualquiera $f : A \rightarrow B$, puede hablarse de la relación de equivalencia que induce sobre el conjunto A como aquella en la que son equivalentes los elementos que tienen la misma imagen.

a. Defina en ocaml una función `rel_eq` : $('a \rightarrow 'b) \rightarrow 'a * 'a \rightarrow \text{bool}$, 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. Defina en ocaml una función

`clases_eq : ('a * 'a -> bool) -> 'a list -> 'a list list,`

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

(El orden no sería relevante, aunque sí el número de apariciones de cada elemento; así, por ejemplo,

`clases_eq (function (x,y) -> x mod 2 = y mod 2)
[5;7;9;10;30;0;4;1;5;10]`

podría ser la lista

`[[10; 30; 0; 4; 10]; [5; 7; 9; 1; 5]]).`

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

PROGRAMACIÓN DECLARATIVA
19 de diciembre de 2000

APELLIDOS: _____

NOMBRE: _____ DNI: _____

Programación Funcional

1. Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

let x = [1;2;3] in 4::x;;

List.hd x;;

let x = let x = [1;2;3] in List.tl x;;

let y::x = x;;

let (y,x) = (x,y::[]) in y@x;;

let x y = y::[] in x y;;

(function f -> f (f 2.0)) (function f -> f ** f);;

let rec f n = if n > 1 then n * f (n-2) else n;;

f 4 + f 5;;

```

let rec f = function n -> n * f (n-2) | 1 -> 1 | 0 -> 0;;
f 4 + f 5;; 

let f = function 0 -> 0 | 1 -> 1 | n -> n * f (n-2);;
f 4 + f 5;; 

let rec ap op = function [] -> []
  | cab::col -> op cab::ap op col;;
let pri (a,_)= a;;
ap pri;;
ap pri [[[],[],[]]];;

```

2. Defina las funciones **ldoble**, **lcuadrado** y **lcubo** de forma que aplicadas a una lista de flotantes den, respectivamente, la lista de sus dobles, de sus cuadrados y de sus cubos. Y defina las funciones **lsuma** y **lproducto** de forma que aplicadas a una lista de flotantes den, respectivamente, la suma y el producto de todos sus elementos.

3. a. Indique el tipo de las funciones `filtro` y `no_pertenenece_a`, definidas a continuación.
b. Utilizando las funciones `filtro` y `no_pertenenece_a`, escriba una definición lo más breve posible de la función `diferencia`: '`a list->'a list-> 'a list`', de forma que `diferencia l1 l2` sea la lista de los elementos de `l1` que no están en `l2`.

```
let rec filtro f =
  function [] -> []
    | h :: t -> if f h then h :: filtro f t
                  else filtro f t;;
```

```
let rec no_pertenenece_a =
  function [] -> (function _ -> true)
    | h :: t -> (function x -> x <> h & no_pertenenece_a t x);;
```

```
let diferencia
```

PROGRAMACIÓN DECLARATIVA

11 de diciembre de 2001

NOMBRE: _____ DNI: _____

Programación funcional

1. (2.5 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
# let x y = y + 1, y - 1;;
```

```
# let y = x 0;;
```

```
# let x,y = y and y = 0;;
```

```
# let rec fold2 op1 op2 e = function
  h :: t -> op1 h (fold2 op2 op1 e t)
  | []    -> e;;
```

```
# fold2 (+) (* ) 0 [1;2;3;4], fold2 (+) (-) 1 [1;2;3;4];;
```

2. (1.5 ptos.) Indique el tipo de las funciones definidas e continuación y redefinalas en modo "curry" de la forma más breve que pueda.

```
# let op = function p -> fst p (snd p);;
```

```
val op :
```

```
# let op =
```

```
val op :
```

```

# let div (x,y) = (/) x y;;
val div :

# let div =
  val div : float * float -> float

# let max (p,ord) = if ord p then snd p else fst p;;
val max :

# let max =
  val max : float * float -> float

```

3. (1.5 ptos.) Un valor de tipo `(float * float) list` puede representar un camino poligonal en el plano (sería la lista ordenada de las coordenadas de los vértices del camino). Defina una función `distancia`: `(float * float) list -> float` que calcule la longitud de cada uno de estos caminos. Recuerde que la distancia entre dos puntos, (x_1, y_1) y (x_2, y_2) , viene dada por la fórmula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

`let distancia`

PROGRAMACIÓN DECLARATIVA
16 de diciembre de 2002

NOMBRE: _____ DNI: _____

Ingeniería Técnica en Informática de Gestión Ingeniería Informática

1. (3 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

let f x = x + 1, x - 1;;

let x = f 0;;

let y, x = x and z = x;;

let x y = y::[] in x y;;

let mas f g = function (x,y) -> f x + g y;;

let x = let x = [1;2;3] in List.tl x;;

2. (2 ptos.) Escriba una definición recursiva terminal de la siguiente función `f`: `int -> int`

```
let rec f x = if x >= 4 then 3 * f(x-1) - 2 * f(x-3)
               else x;;
```

3. (2 ptos.) Escriba una definición recursiva terminal de la función `lista2arbol`: `'a list -> 'a arbol`:

```
let rec lista2arbol = function
    [] -> arbol_vacio
    | h::t -> insert h (lista2arbol t);;
```

donde `arbol_vacio`: `'a arbol` y `insert`: `'a -> 'a arbol -> 'a arbol`.
(Puede suponerse que la función `insert` está definida de modo terminal).