

PARADIGMAS DE LA PROGRAMACIÓN

3 DE FEBRERO DE 2008

NOMBRE: _____

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;;
```

```
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) = <fun>
```

```
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 = <fun>
```

```
let rec suces gen x0 = function
```

```
  0 -> []
```

```
  | n -> x0 ::: suces gen (gen x0) (n-1);;
```

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

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

```
- : int list = [0;1;2;3;4]
```

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

```
- : int = 20
```

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);;
```

```
let sumpro n =
  let rec aux (a,b) m =
    if m < 1 then (a,b)
    else aux (a+m,b*m) (m-1)
  in aux (0,1) 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*:

<i>aBase 10 1234 = [4;3;2;1]</i>	<i>deBase 10 [4;3;2;1] = 1234</i>
<i>aBase 100 1234 = [34; 12]</i>	<i>deBase 100 [34; 12] = 1234</i>
<i>aBase 2 11 = [1;1;0;1]</i>	<i>deBase 2 [1;1;0;1] = 11</i>
<i>aBase 3 11 = [2;0;1]</i>	<i>deBase 3 [2;0;1] = 11</i>
<i>aBase 16 200 = [8;12]</i>	<i>deBase 16 [8;12] = 200</i>

y *deBase b (aBase b n) = n*, siempre que *b* y *n* cumplan los requisitos arriba mencionados.

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

let rec deBase b = function
  [] -> 0
  | h::t -> h + b * (deBase b t);;
```

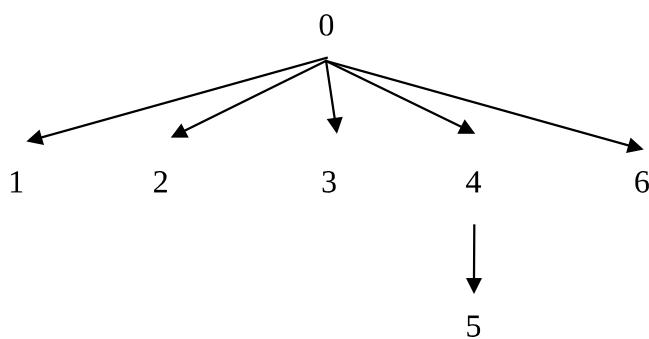
4. (2 puntos) Dada la siguiente definición en *ocaml* para los tipos de datos '*a arbolgen* (que sirven para representar árboles con nodos etiquetados con valores de tipo '*a*, en los que de cada nodo puede colgar cualquier número de ramas)

```
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]

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

PARADIGMAS DE LA PROGRAMACIÓN
22 de diciembre 2009

NOMBRE: _____

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

```
let x, y = "hola", "adios";;
```

```
val x: string = "hola"  
val y: string = "adios"
```

```
let x y = x ^ y;;  
val x: string → string = <fun>
```

```
let x = x "pepe" in x;;  
:: string "holapepe"
```

```
let x::y::z = [1] @ [2] @ [3];;  
val x: int = 1  
val y: int = 2  
val z: int list = [3]
```

```
let x::y::z = [1]::[2]::[3]::[];;  
val x: int list = [1]  
val y: int list = [2]  
val z: int list list = [[3]]
```

```
let rec m2 l1 l2= match (l1,l2) with  
([],[]) -> [] | (a::b, c::d) -> a c :: m2 b d;;  
val rec: ('a → 'b) list → 'a list → 'b list = <fun>
```

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

```
(function x-> x (x 2)) (function x -> 2 * x * x);;  
:: int = 128
```

2. Diremos que una lista es sublista 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 sublista, esta tendrá 2^n sublistas. Así, por ejemplo, la lista [3;5;3] tendría las siguientes sublistas: [], [3], [5], [5; 3], [3], [3; 3], [3; 5] y [3; 5; 3]. (Nótese que alguna sublista 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 sublistas de una lista dada. (Si lo desea puede optar por no incluir repeticiones en la lista de sublistas; pero en ese caso debe indicarlo explícitamente).

```
let insertar x l =
  let rec aux r = function
    []    -> r@l
  | h::t -> aux ((x::h)::r) t
  in aux [] l;;

let rec sublistas = function
  []    -> []
| [h]  -> [[h]];[]
| h::t -> insertar h (sublistas t);;
```

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, sublista de l1. (Se tendrá en cuenta la eficiencia de la definición).

```
let rec pertenece x = function
  []    -> false
| h::t -> if x = h then true
           else pertenece x t;; 

let rec quitar x = function
  []    -> []
| h::t -> if x = h then t
           else h::(quitar x t);;

let rec sublista_de l = function
  []    -> true
| h::t -> if pertenece h l then sublista_de (quitar h l) t
           else false;;
```

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.

```
let apariciones x l =
  let rec aux n = function
    [] -> n
  | h::t -> if x = h then aux (n+1) t
             else aux n t
  in aux 0 l;;
```

PARADIGMAS DE LA PROGRAMACIÓN
6 de febrero de 2009

NOMBRE: _____ DNI: _____

Programación funcional

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 x = 0;;
```

```
val x: int = 0
```

```
# x+1, x-1;;
```

```
-: (int * int) = (1,-1)
```

```
# let x = let x = 2 in x * x;;
```

```
val x : int = 4
```

```
# let f x = 0;;
```

```
val f: 'a -> int = <fun>
```

```
# let f , y = (function x -> x + x), f 2;;
```

```
val f: int -> int = <fun>
val y: int = 0
```

```
# let rec map f = function
  h::t -> f h :: map f t
  | []    -> [];;
```

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

```
# map abs;;
```

```
-: int list -> int list = <fun>
```

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

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

```
# let ds = doble ((+) 2) in map ds [1; 2; 3];;
```

```
-: int list = [5;6;7]
```

2. (1.5 ptos.) Indique el tipo de la función definida e continuación y redefínala usando sólo recursividad terminal. (Si lo desea puede utilizar la función List.rev)

```
# let rec pares = function
  x::y::t -> y :: pares t
  | _ -> [];;
```

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

```
let pares l =
  let rec aux r = function
    x::y::t -> aux (y::r) t
    | _           -> r
  in List.rev (aux [] l);;
```

3. (1.5 ptos.) Diremos que un árbol binario es **completo** si de cada nodo que no sea una hoja *cuelgan dos* ramas (no vacias. Defina la función **escompleto**: 'a arbb -> bool.

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

```
let rec completo = function
  Vacio          -> true
  | Nodo(_,i,Vacio) -> false
  | Nodo(_,Vacio,d) -> false
  | Nodo(_,i,d)      -> (completo i) && (completo d);;
```

PARADIGMAS DE LA PROGRAMACIÓN

6 DE SEPTIEMBRE DE 2009

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, reescríbalo 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

10 DE DECEMBRO DE 2010

APELIDOS: _____ NOME: _____

E.I.

E.T.I.X.

1. (5 puntos) Escriba o resultado da compilación e execución das seguintes frases, con tipos e valores, como faría o compilador interactivo de *ocaml*:

`let five _ = 5;;`

`let id x = x and apply x y = x y;;`

`five 0, id 0, (id five) 0, id (five 0), apply five true;;`

`let mx3 x y z = max x (max y z);;`

`mx3 1, mx3 1 2, mx3 1 2 3;;`

`let rec fold x = function [] -> x | (op,y)::t -> fold (op x y) t;;`

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

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

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

```
let x = let x = 1 and y = 2 in x + y, x - y;;
```

2. (1 punto) Reescriba as seguintes definicións sen utilizar definicións locais nin expresións if...then...else...

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

```
let n x g = if g x then true else false;;
```

3. (3 puntos) Indique o tipo das seguintes funcións:

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

```
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;;
```

¿É terminal a recursividade empregada nestas definicións?

Se nalgún caso non é así, realice unha nova definición recursiva terminal para a función correspondente.

4. (1 punto) Considere a seguinte definición en ocaml para o tipo de dato **bitree** (que serve para representar árbores binarias en ocaml):

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

Dicimos que unha árbore binaria é “perfecta” se ten cheos todos os seus niveis. Isto é, unha árbore binaria perfecta terá 2^i nodos no nivel i (para cada nivel i da árbore). Defina a función **es_perfecta**: **bitree -> bool** que indique se unha árbore é ou non é perfecta.

PARADIGMAS DE LA PROGRAMACIÓN

16 DE DICIEMBRE DE 2011

NOMBRE: _____

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;;
  val id: 'a → 'a = <fun>
```

```
let cte k = function _ -> k;;
  val cte: 'a → 'b → 'a = <fun>
```

```
(cte 0) "a";;
  -: int = 0
```

```
let rec genlist f = function
  0 -> []
  | n -> (genlist f (n-1)) @ [f n];;
  val genlist: (int → 'a) → int → 'a list = <fun>
```

```
let l1,l2 = let dela n = genlist id n
            in dela 4, dela 5;;
  val l1: int list = [1;2;3;4]
  val l2: int list = [1;2;3;4;5]
```

```
let l3 = let rep x n = genlist (cte x) n in rep 5 2;;
  val l3: int list [5;5]
```

```
let rec reduce f e = function [] -> e | h::t -> f h (reduce f e t);;
  val reduce: ('a → 'b → 'b) → 'b → 'a list → 'b = <fun>
```

```
let sigma = reduce ( + ) 0;;
  val sigma: int list → int = <fun>
```

```
let pi = reduce ( * ) 1;;
  val pi: int list → int = <fun>
```

```
let l4 = (List.map sigma [l1; l2; l3]) in (sigma l4, pi l4);;
  -: int * int = (35,1500)
```

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.

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

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");
let f = function
    0 -> 0
  | n -> let rec aux m r1 r2 =
              if m=n then r2
              else aux (m+1) r2 (2*r2 - 3*r1)
            in if n<0 then raise (Failure "f")
                else aux 1 0 1;;
```

4. (2 puntos) Defina una función criba : ('a -> bool) list -> 'a list list, de forma que criba [p1; p2;...; pn] l devuelva una lista de listas cuyo primer elemento sea la lista de elementos de l en los que se cumple el predicado p1 ; el segundo, la lista de elementos de l que no cumplen p1 , pero sí p2 ;... ; el penúltimo, la lista de elementos de l que cumplen el predicado pn , 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]]]
```

```
let criba flist list =
  let rec aux ep eq fl ra rf = match (fl,eq) with
    ([],[])
      -> rf
    | ([],l)
      -> rf@[l]
    | (f::ft,[])
      -> aux [] ep ft [] (rf@[ra])
    | (f::ft,h::t)
      -> if f h then aux ep t (f::ft) (ra@[h]) rf
          else aux (ep@[h]) t (f::ft) ra rf
  in aux [] list flist [] [];;
```

PARADIGMAS DE LA PROGRAMACIÓN

11 de febrero de 2011

NOMBRE: _____ DNI: _____

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) = (2.0,-3.0)
```

```
# let p = let x,y = p in y,x;;
```

```
val p: (float * float) = (-3.0,2.0)
```

```
# 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 -> print_endline(string_of_int h); imprime_inversa t;;
```

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

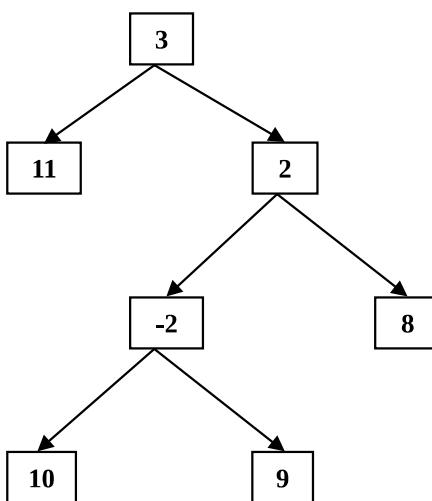
4. (3 ptos.) Considere la siguiente definición del tipo de dato ‘**a tree**’ que podría servir para representar en ocaml cierto tipo de árboles binarios:

```
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 14.



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,p,d) -> p +. 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;2;-2;10];[3;2;-2;9];[3;2;8]]**.

```
let rec caminos = function
  Leaf p -> [[p]]
  | Node(i,p,d) -> let f l = p :: 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.

PARADIGMAS DE LA PROGRAMACIÓN
11 de diciembre de 2012

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;;
```

```
val x: int → (int * int) = <fun>
```

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

```
val y: (int * int) = (1,-1)
```

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

```
ERROR
```

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

```
val fold2: ('a → 'b → 'b) → ('a → 'b → 'b) → 'b → 'a list → 'b = <fun>
```

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

```
-: (int * int) = (7,-3)
```

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

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

```
val op: (('a → 'b) * 'a) → 'b = <fun>
```

```
let op f x = f x;;
```

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

```
# let div (x,y) = (/) x y;;
```

```
val div: (int * int) -> int = <fun>
```

```
let div x y = x / y;;
```

```
val div: int -> int -> int = <fun>
```

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

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

```
let max p ord = if ord p then snd p else fst p;;
```

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

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 longitud (x1,y1) (x2,y2) =
  sqrt( ((x2 -. x1) ** 2.0) +. ((y2 -. y1) ** 2.0) );;

let distancia l =
  let rec aux r = function
    p1::p2::t -> aux (r +. longitud p1 p2) (p2::t)
    | _ -> r
  in aux 0. (l @ [List.hd l]);;
```

PARADIGMAS DE LA PROGRAMACIÓN

13 DE SEPTIEMBRE DE 2012

NOMBRE: _____

1. (6 puntos) Escriba el resultado de la compilacion y ejecucion de las siguientes frases, con tipos y valores, como lo indicaría el toplevel 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 → 'b) → 'b = <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
```

```
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 r = function
    [] -> r
    | h::t -> aux (h + 2*r) t
  in aux 0 (List.rev l);;
```

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, [])]);;
```

corresponderia arbol

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

```
graph TD; 0 --- 1; 0 --- 2; 0 --- 3; 0 --- 4; 4 --- 5; 3 --- 6;
```

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

PARADIGMAS DE LA PROGRAMACIÓN

14 DE FEBRERO DE 2012

NOMBRE: _____

1. (5.5 puntos) Escriba el resultado de la compilacion y ejecucion 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);
  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 r = function
    [] -> r
  | h::t -> aux (h r) t
  in aux x (List.rev l);;
```

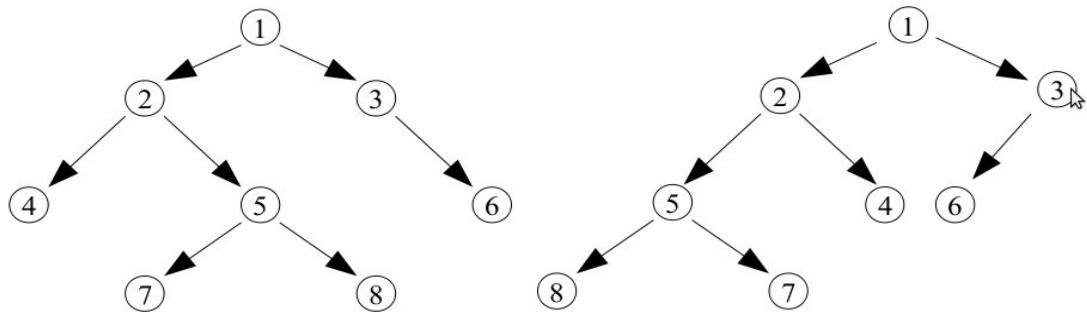
3.(1.5 puntos) La función max_en está definida de forma que el valor de max_en x l corresponde a la función de la lista l que alcanza el mayor valor en el punto x. Redefínala para optimizar su eficiencia.

```
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")
  | h::t -> 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 h (h x) t;;
```

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;;
```

1.-(1 Pto)Indique el valor y el tipo de:

```
# let x = 2.0 in function x -> 2*x;;
# let swap(x,y) = y,x in let x,y = 2,3 in swap (y,x)
# function x -> function f -> function f x
# let f s i = s ^ string_of_int i ^ "," in List.fold_left f ""
```

2.-(1,5 Ptos)Dada la siguiente funcion:

```
# let rec f n= if n<=1 then n else 2*f(n-1) - f(n-2)
```

a.-(0,75 Ptos) Recursividad terminal de f:

```
let rec f1 n =
  let rec aux x menos1 menos2 =
    if x=n then
      2*menos1-menos2
    else
      aux (x+1) (2*menos1-menos2) menos1
  in
    if n>1 then
      aux 2 1 0
    else
      n;;
```

b.-(0,75 Ptos) Implementación Imperativa de f:

```
let f n = n;;
```

3.-(0,75 Ptos)

a.-(0,5 Ptos) Define list2string: ('a -> string) -> string -> 'a list -> string , de xeito que list2string f s transforme unha lista nun string, representando os elementos da lista cos seus valores pola función f, separados co string s. Así, por exemplo list2string string_of_int "," [2;3;7] debe ser o string "2,3,7"

```
let rec list2string f s = function
  []->""
  |h::[] -> f h
  |h::t -> f h ^ s ^ list2string f s t;;
```

b.-(0,25 Ptos) Utilizando list2string define printintlist: int list -> unit , de xeito que printintlist l envie á salida estandar a mesma representación que emprega nas suas mensaxes o compilador interactivo Ocaml para as listas de enteros

```
let printintlist l = print_endline ("["^(list2string string_of_int "; " l)^"]");;
```

4.-(1 Pto) Defina prefix: 'a list -> 'a list -> bool, de xeito que o valor de prefix l1 l2 indique se a lista l1 é un segmento inicial da lista l2

(Os segmentos iniciais de [1;2;3] son [],[1],[1,2],[1;2;3])

```
let rec prefix l1 l2 = match l1 with
  [] -> true
  |h::t -> if h=(List.hd l2) then
    prefix t (List.tl l2)
  else false;;
```

5.-(1,75 Ptos) Considere:

```
# type 'a tree= Empty | Node of 'a tree * 'a * 'a tree;;
# type vtree = VEmpty | VNode of vtree * vtree;;
```

Que serven para representar árbores binarias con nodos etiquetados con valores de tipo 'a, e estructuras de árbores binarias.

a.-(0,25 Ptos) Defina preorder: 'a tree -> 'a list

b.-(0,5 Ptos) Defina forget: 'a tree -> vtree, de xeito que forget t sexa a árbore que resulta ao "esquecer" os valores dos nodos da árbore t, respectando a súa estrutura.

c.-(1 Pto) Defina fillwith: 'a list -> vtree -> 'a tree, tal que a árbore fillwith l t teña a mesma estrutura que t e o seu percorrido en preorde sexa un segmento inicial de l (é dicir, forget(fillwith l t) = t e prefix(preorder(fillwith l t)) l deben ser true). Se, para unha lista l e para unha estrutura de árbore t, non se pode construír tal árbore, a función fillwith debe activar a excepción Invalid_argument "fillwith"

(**5**)
type 'a tree= Empty | Node of 'a tree * 'a * 'a tree;;
type vtree = VEmpty | VNode of vtree * vtree;;

(*a*)
let rec preorder a = match a with
 Empty -> raise (Failure "árbol vacio")
 |Node (Empty,r,Empty) -> [r]
 |Node (i,r,Empty) -> r::preorder i
 |Node (Empty,r,d) -> r::preorder d
 |Node (i,r,d) -> [r] @ (preorder i) @(preorder d);;

(*b*)
let rec forget a = match a with
 Empty -> raise (Failure "árbol vacio")
 |Node (Empty,r,Empty) -> VNode(VEmpty,VEmpty)
 |Node (i,r,Empty) -> VNode(forget i,VEmpty)
 |Node (Empty,r,d) -> VNode(VEmpty,forget d)
 |Node (i,r,d) -> VNode(forget i,forget d);;

6.-(2 Ptos) Implementa a clase PILA empregando unha LISTA(list) que contará coas operacións:

```
# PUSH -> mete un elemento na cima
# TOP -> recupera o 1º elemento (cima) sen borralo
# POP -> elimina o elemento da cima
# IS_EMPTY -> comproba se está baleira
```

a.-(1,25 Ptos) Implementa a clase PILA coa axuda de List.hd e List.tl

```
exception PilaVacia;;

class ['a] pila =
object

  val mutable lista = ([] : 'a list)

  method push x =
    lista <- x::lista

  method top =
    try
      List.hd lista;
    with _ -> raise(PilaVacia)

  method pop =
    try
      lista <- List.tl lista
    with _ -> raise(PilaVacia)

  method is_empty =
    (lista = [])
end;;
```

b.-(0,25 Ptos) Indica de que tipo son os métodos resultantes.

```
method is_empty : bool
method pop : unit
method push : 'a -> unit
method top : 'a
```

c.-(0,25 Ptos) Explica porque da erro a seguinte secuencia:

```
let p1 = new pila;;
p1#push 1;;
p1#pop ();
p1#push true;;
```

La lista de la pila se crea vacía y almacena elementos de tipo 'a. Cuando se introduce el primer elemento de tipo integer, el tipo de la lista se cambia a integer. El fallo se produce cuando intentas meter un elemento de tipo bool.

d.-(0,25 Ptos) Extenda a clase PILA a unha subclase pila_enteros que traballe únicamente sobre enteros

```
class pila_enteros =
object
  inherit [int] pila
end;;
```

JULIO DE 2013

1) escriba el resultado de la compilación tal y como lo haría el Ocaml:

```
let rec p = function 0 > true | n > i(n#)
```

```
and i = function 0 > false | n > p(n#);;
```

```
val p : int > bool = <fun>
```

```
val i : 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;;
```

Downloaded by paco perez (takok91961@wmail2.net)

```
(* val y: int = 3 *)
```

```
f x + f y;;
```

```
(* + int = 9 *)
```

```
let p = let x,y = x+y,xy in y,x;;
```

```
(* val p: int*int = (1,5) *)
```

```
let p = x+y,yx in let x,y = p in y,x;;
```

```
(* + int * int = (1,5) *)
```

2) Pasar la función anterior a una sin referencias, de programación imperativa a declarativa.

```
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(b, a mod b)
  in aux(max (abs x) (abs y),min (abs x) (abs y));;

```

3) Dada la siguiente definicion, hacer el recorrido en anchura del arbol
 $'a \text{ arb} \rightarrow 'a \text{ list}$.

```

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];;

```

PARADIGMAS DE LA PROGRAMACIÓN
16 de diciembre de 2013

NOMBRE: _____ DNI: _____

- 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;;
```

```
val f: int → (int * int) = <fun>
```

```
# let x = f 0;;
```

```
val x: (int * int) = (1,1)
```

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

```
val y: int = 1
```

```
val x: int = 1
```

```
val z: (int * int) = (1,1)
```

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

```
* int list = [1]
```

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

```
val mas: ('a → int) → ('b → int) → ('a * 'b) → int = <fun>
```

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

```
val x: int list = [2;3]
```

- 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(x1) -2 * f(x3)
               else x;;
```

```
let f x =
  let rec aux n r1 r2 r3 =
    if n<=x then aux (n+1) r2 r3 (3*r3 -2*r1)
    else r3
  in if x>=4 then aux 4 1 2 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).

```
let lista2arbol l =
  let rec aux r = function
    []    > r
  | h::[] > insert h r
  | h::t  > aux (insert h r) t
  in aux arbol_vacio (List.rev l);;
```

Examen Paradigmas de la Programación Junio 2014

```
# 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
    | (",[") -> 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)
```

2. Definir una función suma que sume los elementos de dos listas respectivamente. Si una de las listas es mas corta que la otra, se rellenará con ceros. Es decir [2;5;4;9] y [2;4;3] debería de dar la lista... [4;9;7;9]. Si se define con recursividad terminal será un punto mas.

Sin recursividad terminal:

```
let rec suma l1 l2 = match (l1,l2) with
  ([],[]) -> []
| (l,[]) -> l
| (,[],l) -> l
| (h1::t1,h2::t2) -> (h1+h2):: suma t1 t2;;
```

Con recursividad terminal

```
let suma l1 l2 =
  let rec aux l1 l2 r = match (l1,l2) with
    ([],[]) -> r
  | (l,[]) -> r@l
  | (,[],l) -> r@l
  | (h1::t1,h2::t2) -> aux t1 t2 (r@[h1+h2])
  in aux l1 l2 [];;
```

3. Para los tipos de datos siguientes:

```
type 'a a2 = AO of 'a
| AIz of 'a * 'a a2
| ADC of 'a * 'a a2
| A2 of 'a * 'a a2 * 'a a2;;
type 'a abin = V | N of 'a * 'a abin * 'a abin;;
```

Definir la función a2_of_abin, que a partir de un abin generará un a2 y la función abin_of_a2 que hará lo contrario.

```
let rec abin_of_a2 =
  AO(a) -> N(a, V, V)
| AIz(a,i) -> N(a, abin_of_a2 i, V)
| ADC(a,d) -> N(a, V, abin_of_a2 d)
| A2(a,i,d) -> N(a, abin_of_a2 i, abin_of_a2 d);;
```

```
let rec a2_of_abin = function
  V -> raise (Failure "a2_of_abin")
| N(a,V,V) -> AO(a)
| N(a,V,d) -> ADC(a, a2_of_abin d)
| N(a,i,V) -> AIz(a, a2_of_abin i)
| N(a,i,d) -> A2(a, a2_of_abin i, a2_of_abin d);;
```

PARADIGMAS DE LA PROGRAMACIÓN

30 DE ENERO DE 2014
PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

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);;
```

```
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: (int → ('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. (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.

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

b. defina una función ***subst: 'a -> 'a -> 'a arbol -> 'a arbol***, de forma que ***subst x y*** sea una función que, al aplicarla a un árbol, devuelve un arbol igual al original salvo los nodos que tuviesen valor ***x***, que tendrán valor ***y***.

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

3. (1 punto) 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 este tipo están ordenados), ***l_ordenada f*** sea la función que dice si una lista está ordenada según el orden **f**.

```
let rec l_ordenada f = function
  []
    -> true
  | h::[]
    -> true
  | h1::h2::t -> if f h1 h2 then l_ordenada f (h2::t)
                  else false;;
```

4. (1 punto) Defina utilizando exclusivamente 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 "max")
)
| h::t -> let rec aux m = function
  []
    -> m
  | h::t -> if h>m then aux h t
              else aux m t
in aux h t;;
```

Ejercicio 1.**(2 puntos)**

Escriba el resultado de las siguientes frases, con tipos y valores, como lo indicaría el “toplevel” de ocaml:

```
let g f x = x, f x, f (f x);;
val g : ('a -> 'a) -> 'a -> 'a * 'a * 'a = <fun>
```

```
let x, y, z = g ((+) 2) 0;;
val x : int = 0
val y : int = 2
val z : int = 4
```

```
let x = x + y + z in let y = x + y + z in x, y;;
- : int * int = (6, 12)
```

```
x + y + z;;
- : int = 6
```

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

```
let doble = pow 2 and sq x = x * x in doble sq 3;;
- : int = 81
```

```
let rec zip f = function ( [], [] ) -> []
| ( h1 :: t1, h2 :: t2 ) -> f h1 h2 :: zip f ( t1, t2 ) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: ----- > (_ :: _, [])
val zip : ('a -> 'b -> 'c) -> 'a list * 'b list -> 'c list = <fun>
```

```
zip (+) ( [1; 2; 3], [4; 5] );;
Exception: Match_failure ("", 2, -27).
```

Ejercicio 2.

Considere la siguiente función en ocaml:

```
let rec stcr = function
  [] → true
  | [h] → true
  | h1 :: h2 :: t → (h1 < h2) && (stcr ( h2 :: t ));;
```

a) Indique el tipo y valor de la función *stcr*. (0'25 puntos)

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

b) Realice una nueva definición para *stcr* de forma que sea recursiva terminal. (0'50 puntos)

```
let stcr lista =
  let rec aux accum = function
    [] → accum
    | [h] → accum
    | h1 :: h2 :: t → ( let accum2 = ( h1 < h2 ) in
      aux accum2 t )
  in aux true lista;;
```

c) Defina una función *segcr*: '*a list* → '*a list list* de modo que:

List.concat (segcr l) = *l* y *List.for_all stcr (segcr l)* = *true* para todo *l*: '*a list*. Y si hay otra lista *l'*: '*a list list* que cumpla *List.concat l' = l* y *List.for_all stcr l'* = *true*, entonces *List.length l'* >= *List.length l*.

(Nota: podría decirse que *segcr* descompone una lista en el menor número posible de segmentos estrictamente crecientes).

(Por ejemplo: *segcr []* = *[]*; y *segcr [4;2;5;7;7;8;1;1;0]* = *[[4] ; [2; 5; 7] ; [7; 8] ; [1] ; [1] ; [0]]*). (1 punto)

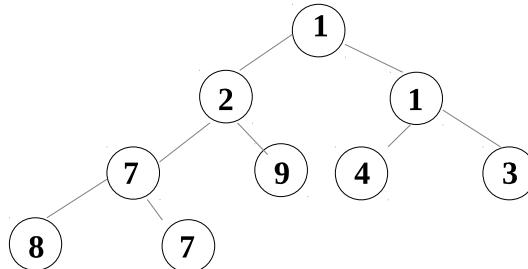
```
let segcr l =
  let rec segcr1 n s = function
    [] -> List.rev s
    | h::[] -> segcr1 [] ((n@[h]):s) []
    | h::t -> if ((h) < (List.hd t)) then (segcr1 (n@[h]) s t) else
      (segcr1 [] ((n@[h]):s) t)
  in segcr1 [] [] l;;
```

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

Ejercicio 3.**(1 punto)**

Diremos que un árbol es un “montículo de mínimos” respecto a una relación de orden, si el valor asociado a cada nodo del árbol es anterior (según esa relación) a los valores de todos los nodos que descienden de él.

Por ejemplo, el siguiente árbol será un “montículo de mínimos” respecto a la relación de orden habitual en los enteros (\leq).



Dado el tipo 'a tree definido a continuación, define una función minmt : ('a -> 'a -> bool) -> 'a arbol -> bool, que indique si un árbol es, o no, un montículo de mínimos, respecto a una relación de orden dada.

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

(Función auxiliar que devuelve la raíz del árbol)

```
let root = function
    Leaf r → r
    | Node (r, _, _) → r;;
```

(Función propia, minmt)

```
let rec minmt p = function
    Leaf _ → true
    | Node (r, Leaf i, Leaf d) → (p r i) && (p r d)
    | Node (r, Node i, Leaf d) → (p r (root (Node i))) && (p r d) &&
                                    (minmt p (Node i))
    | Node (r, Leaf i, Node d) → (p r i) && (p r (root (Node d))) &&
                                    (minmt p (Node d))
    | Node (r, Node i, Node d) → (p r (root (Node i))) && (p r (root (Node d))) &&
                                    (minmt p (Node i)) && (minmt p (Node d));;
```

Ejercicio 4.

Considere la siguiente definición en ocaml de la función lst:

```
let rec lst = function
  [] → []
  | [h] → [h]
  | h::t → lst t;;
```

a) Indique el tipo y valor de la función *stcr*. (0'25 puntos)

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

b) Redefina la función *length* del módulo List, “en estilo imperativo”, sin utilizar recursividad. Las únicas funciones predefinidas que puede utilizar son List.hd y List.tl
El tipo debe ser : val length : 'a list -> int = <fun> (0'50 puntos)

```
let length lista = match lista with
  [] → 0
  | _ → let f = ref 1
    and l = ref lista in
      while ( List.tl !l <> [] ) do
        f := !f + 1;
        l := List.tl !l;
      done;
    !f;;
```

c) Redefina la función *lst* “en estilo imperativo”, sin utilizar recursividad. Las únicas funciones predefinidas que puede utilizar, son List.hd, List.tl y la función *length* definida en el apartado anterior. (0'50 puntos)

```
let lst lista = match lista with
  [] → []
  | _ → let f = ref []
    and l = ref lista in
      for i = 0 to ((length lista) -1) do
        f := [List.hd !l];
        l := List.tl !l;
      done;
    !f;;
```

Ejercicio 5.

Ejercicio de orientación a objetos.

a) Crear una clase, con sus respectivos métodos getters, etc...

b) Crea una subclase de manera que la superclase sólo esté definida para utilizar con floats:

```
class float_wrapper (arbol : tipoarbol) (context: float tipo_contexto) =  
    object (this)  
        inherit NombreSuperClase arbol context as super  
  
    ...
```

PARADIGMAS DE PROGRAMACIÓN

15 / 01 / 2015

APELLIDOS _____ NOMBRE _____
APELIDOS NOME

1. (3 puntos) Indique el efecto de la compilación y ejecución de las siguientes frases en ocaml (con tipos y valores, y distinguiendo definiciones de expresiones), como lo indicaría el compilador interactivo:

Indique o efecto da compilación e execución das seguintes frases en ocaml (con tipos e valores, e distinguindo definícōes de expresōes), como o indicaría o compilador interactivo:

```
let x = [(1,2);(3,4)];;
```

```
let x::y = x in x::x::y;;
```

```
let (x,y)::_ = List.tl x;;
```

```
x + (let x = x + y in x + y);;
```

```
(function x -> x,x) "hola";;
```

```
let rec prod x = function
  [] -> []
  | h::t -> (x,h):: prod x t;;
```

```
prod 1 [2;3;4];;
```

```
let rec lprod l1 l2 = match l1 with
  [] -> []
  | h::t -> prod h l2 @ lprod t l2;;
```

```
lprod [1;2] ['a';'b';'c'];;
```

2. (0,5 puntos) Redefina la función *prod* do ejercicio anterior con la función *List.map*, sen utilizar directamente recursividat.

Redefina a función *prod* do exercicio anterior coa función *List.map*, sen utilizar directamente recursividat.

```
let prod
```

3. a) (0,5 puntos) Indique el tipo de la función *f* definida a continuación:

Indique o tipo da función *f* definida a continuación:

```
let rec f = function x::y::t -> f (y::t) && x <= y
  | _ -> true;;
```

```
f:
```

b) (0,5 puntos) ¿Es esa definición recursiva terminal? Si no lo fuera, redefina la función *f* de modo que só se utilice recursividat terminal.

É esa definición recursiva terminal? Se non o fose, redefina a función *f* de modo que só se utilice recursividat terminal.

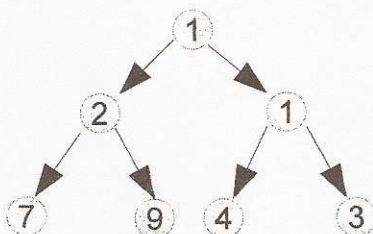
4. Considere la siguiente definición de tipos de datos que sirven para representar árboles estrictamente binarios, con nodos etiquetados con valores de tipo '*a*'.

Considere a seguinte definición de tipos de datos que serven para representar árbores estrictamente binarias, con nodos etiquetados con valores de tipo '*a*'.

```
type 'a tree = L of 'a | T of 'a * 'a tree * 'a tree
```

a) (0,5 puntos) Represente, con un valor de tipo *int tree*, el arbol, *t*, dibuxado a continuación.

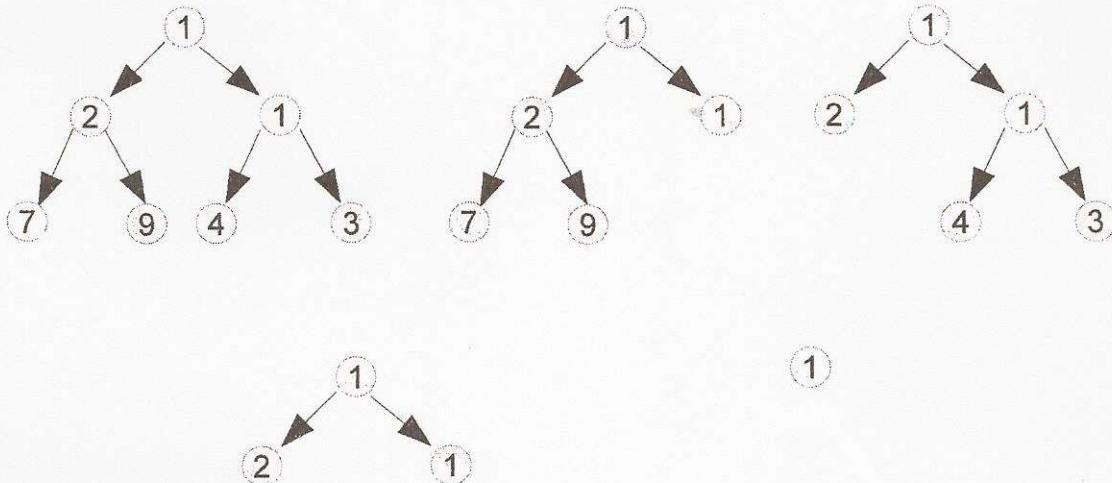
Represente, cun valor de tipo *int tree*, el arbol, *t*, debuxado a continuación.



```
let t =
```

Llamaremos "segmentos raíz" de un árbol, a todos aquellos árboles que puedan obtenerse a partir de este "podándolo", o no, por uno o varios nodos. Siendo más precisos, los segmentos raíz del árbol *t*, serían exactamente los 5 siguientes:

Chamaremos "segmentos raíz" dunha árbore, a todas aquellas árbores que poidan obterse a partir desta "podánda", quizais, por un ou varios nodos. Sendo más precisos, os segmentos raíz da árbore *t*, serían exactamente os 5 seguintes:



b) (1 punto) Defina una función `es_segrz`: '`a tree -> 'a tree -> bool`', de forma que `es_segrz t1 t2` indique si `t1` es (o no es) un segmento raíz de `t2`.

Defina unha función `es_segrz`: '`a tree -> 'a tree -> bool`', de xeito que `es_segrz t1 t2` indique se `t1` é (ou non é) un segmento raíz de `t2`.

PARADIGMAS DE PROGRAMACIÓN

XANEIRO 2015: Parte de obxetos / ENERO 2015: Parte de objetos

APELIDOS/APELLIDOS: _____ NOME/NOMBRE: _____

[0.6 puntos] EXERCICIO / EJERCICIO 1:

Indique cales das seguintes definicións son ou non correctas. Se son correctas, indique o resultado que devolvería o compilador interactivo (tipos e valores, clases inclusive); se non o son, indique brevemente por que.

Indique cuáles de las siguientes definiciones son o no correctas. Si son correctas, indique el resultado que devolvería el compilador interactivo (tipos y valores, clases inclusive); si no lo son, indique brevemente por qué.

class claseB pc1 = object val atributo1 = pc1 method metodo1 () = String.length(atributo1) method metodo2 pm1 = atributo1^pm1 end;;	
class subclaseB1 pc1 pc2 = object inherit claseB pc1 val atributo2 = pc2 method metodo2 pm1 = atributo1^atributo2^pm1 end;;	
class subclaseB2 pc1 pc2 = object inherit claseB pc1 val atributo2 = pc2 method metodo2 pm1 pm2 = atributo1^atributo2^pm1^pm2 end;;	

[0.6 puntos] EXERCICIO / EJERCICIO 2:

```
class claseJ = object  
    val atributo1 = {10; 20; 30}  
    method metodo1 () = atributo1  
    method metodo2 () = List.hd atributo1  
    method metodo3 () = List.tl atributo1  
end;;  
  
class subclaseJ2 = object  
    inherit claseJ  
    val atributo1 = [0]  
end;;  
  
class subclaseJ5 = object  
    inherit claseJ as super  
    method metodo4 () = super#metodo1()  
end;;
```

```
class claseM n =  
object  
    val atributo1 = n::[]  
    method metodo3 () = atributo1  
    method metodo2 () = 0  
    method metodo1 () = [n; (n+1)]  
end;;  
  
let j = new claseJ;;  
let sj2 = new subclaseJ2;;  
let sj5 = new subclaseJ5;;  
let m = new claseM 15;;
```

Dadas as definicións anteriores, indique cales das seguintes listas son ou non válidas. No caso de serlo, indique o seu tipo resultante; no caso de non serlo, indique brevemente por que.

Dadas las definiciones anteriores, indique cuáles de las siguientes listas son o no válidas. En el caso de serlo, indique su tipo resultante, en el caso de no serlo, indique brevemente por qué.

[j; m];;

PARADIGMAS DE PROGRAMACIÓN

2015.07.06

APELLIDOS
APELIDOS

NOMBRE
NOME

1. (3 puntos) Indique o efecto da compilación e execución das seguintes frases en ocaml (con tipos e valores), como o indicaría o compilador interactivo. Distinga claramente expresións de definicións.

Indique el efecto de la compilación y ejecución de las siguientes frases en ocaml (con tipos y valores), como lo indicaría el compilador interactivo. Distinga claramente expresiones de definiciones.

```
let sufix s1 s2 = s2 ^ s1;;
```

```
let past = sufix "ed";;
```

```
past "show";;
```

```
let rec first_match f1 f2 = function
  [] -> raise (Failure "first_match: none")
  | h::t -> if f1 h = f2 h then h else first_match f1 f2 t;;
```

```
let rec dist = function
  h1::h2::t -> let t1,t2 = dist t in h1::t1, h2::t2
  | l -> l, [];;
```

```
dist ['a';'e';'i';'o';'u'];;
```

2. (0,5 puntos) Segundo o manual de OCaml (no módulo List):

Según el manual de OCaml (en el módulo List):

```
val find : ('a -> bool) -> 'a list -> 'a
```

find p l returns the first element of the list l that satisfies the predicate p. Raise Not_found if there is no value that satisfies p in the list l.

Redefina a función *first_match*, do exercicio anterior, utilizando a función *List.find*. (Coa nova definición, a función debe comportarse exactamente igual que a orixinal). É recursiva terminal algunha das dúas versións?

Redefina la función first_match, del ejercicio anterior, utilizando la función List.find. (Con la nueva definición, la función debe comportarse exactamente igual que la original). ¿Es recursiva terminal alguna de las dos versiones?

3. (0,5 puntos) Simplifique os seguintes fragmentos de código OCaml. (Supонse que os valores de *f*, *x* e *h* están previamente definidos e teñen os tipos adecuados; doutra banda, a aplicación de *f* non ten efectos colaterais).

Simplifique los siguientes fragmentos de código OCaml. (Se supone que los valores de f, x y h están previamente definidos y tienen los tipos adecuados; por otro lado, la aplicación de f no tiene efectos colaterales).

if f (x) = true then true else false

if f (x) < h then f (x) else h

4. (1 punto) Realice unha implementación recursiva terminal da función *dist* do exercicio anterior.

Realice una implementación recursiva terminal de la función dist del ejercicio anterior.

5. (0,5 puntos) As dúas funcións definidas no seguinte fragmento de código producen, ao aplicállas, o mesmo efecto. Presenta algunha delas vantaxes ou inconvenientes sobre a outra? É algunha destas dúas definicións recursiva terminal? Indique tamén o tipo de cada unha delas.

Las dos funciones definidas en el siguiente fragmento de código producen, al aplicarlas, el mismo efecto. ¿Presenta alguna de ellas ventajas o inconvenientes sobre la otra? ¿Es alguna de estas dos definiciones recursiva terminal? Indique también el tipo de cada una de ellas.

```
let iter1 f l = for i = 0 to List.length l - 1 do
    f (List.nth l i) done;;
let rec iter2 f = function [] -> () |
    h::t -> f h; iter2 f t;;
```

6. (0,5 puntos) O seguinte fragmento de código foi compilado correctamente co compilador de OCaml.
El siguiente fragmento de código ha sido compilado correctamente con el compilador de OCaml.

```
type treeSK = EmptySK | NodeSK of treeSK * treeSK;;
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree;;
let rec fillwith l sk = match (sk,l) with
    (EmptySK, _) -> Empty, l
  | (NodeSK (ri,rd), h::t) -> let new_ri, resto = fillwith t ri in
    let new_rd, resto = fillwith resto rd in
      Node (h, new_ri, new_rd), resto
  | _ -> raise (Invalid_argument "fillwith: list is too short");;
```

Indique o tipo da función *fillwith*.

Indique el tipo de la función fillwith.

Explique de forma concisa que poderían representar os tipos *treeSK* e *'a tree* e o que faría a función *fillwith*.
Explique de forma concisa qué podrían representar los tipos treeSK y 'a tree y qué haría la función fillwith.

Que sucedería se un programa trata de avaliar a expresión *fillwith [] (NodeSK (EmptySK, EmptySK))*?
¿Qué sucedería si un programa trata de evaluar la expresión fillwith [] (NodeSK (EmptySK, EmptySK))?

PARADIGMAS DE PROGRAMACIÓN

XULLO 2015: Parte de obxetos / JULIO 2015: Parte de objetos

APELIDOS/APELLIDOS: _____ NOME/NOMBRE: _____

[0.4 puntos] EXERCICIO / EJERCICIO 1:

```
class claseF pc1 pc2 =  
object (self: 'self)  
  val atr1 = pc1  
  val atr2 = pc2  
  method private metodo1 () = List.length(atr1)  
  method metodo2 () = Array.length(atr2)  
  method metodo3 () = self#metodo1()  
    + List.hd atr1 + atr2.(0)  
end;;
```

```
class claseI1 (pc1, pc2) =  
object  
  val atr1 = pc1^pc2  
  method metodo1 () = String.length(atr1)  
end;;
```

Dadas as definicións anteriores, indique a continuación, nos espacios axeitados para elo, o resultado (valor e tipo) que devolvería o compilador interactivo para a seguinte secuencia de instruccións:

Dadas las definiciones anteriores, indique a continuación, en los espacios habilitados para ello, el resultado obtenido (valor y tipo) que devolvería el compilador interactivo para la siguiente secuencia de instrucciones:

```
let f1 = new claseF [1;2] [|3;4|];; (* _____ *)
```

```
let valor = f1#metodo1 ();; (* _____ *)
```

```
let f1_2 (pf1, pf2) = new claseI1 (pf1, pf2);; (* _____ *)
```

[0.4 puntos] EXERCICIO / EJERCICIO 2:

Indique cales das seguintes definicións son ou non correctas. Se son correctas, indique o resultado que devolvería o compilador interactivo (tipos e valores, clases inclusive); se non o son, indique brevemente por qué.

NOTA: `Array.make n x` devolve un array de lonxitude `n` con todas as posicións inicializadas a `x`.
`val make : int -> 'a -> 'a array`

Indique cuáles de las siguientes definiciones son o no correctas. Si son correctas, indique el resultado que devolvería el compilador interactivo (tipos y valores, clases inclusive); si no lo son, indique brevemente por qué.

NOTA: `Array.make n x` devuelve un array de longitud `n` con todas las posiciones inicializadas a `x`.
`val make : int -> 'a -> 'a array`

```
class clase01 (n:int) (c: char) =  
object  
  val atr1 = n  
  val atr2 = c  
  val atr3 = [| atr2; atr2; atr2 |]  
  method met1 () = atr1  
  method met2 pm1 (pm2:char)=Array.make pm1 pm2  
  method met3 () = [| atr2; atr2; atr2 |]  
end;;
```

```
class clase03 (n:int) (c: char) =  
object  
  val atr1 = n  
  val atr2 = c  
  val mutable atr3 = [| |]  
  initializer atr3 <- Array.make atr1 atr2  
  method met1 () = atr1  
  method met2 pm1 (pm2:char)=Array.make pm1 pm2  
  method met3 () = [| atr2; atr2; atr2 |]  
end;;
```

PARADIGMAS DE PROGRAMACIÓN

2016.01.20

NOME

APELIDOS

1. (1 punto) Indique o tipo (ou erro de tipo) para cada unha das seguintes expresións en OCaml.

not (if true then 2 = 3 else 'a' <> 'c')

int
 bool

float
 string

char
 type error

if not false then float_of_int 2 else sqrt (sqrt 16.)

int
 bool

float
 string

char
 type error

if 'a' = String.get "abcd" (if 0. = 1. then 2 else 3) then Char.code 'A' else 65

int
 bool

float
 string

char
 type error

1 + (if "int" <> "float" then 2. else 3.) +. 4.

int
 bool

float
 string

char
 type error

2. (1 punto) Indique o efecto da compilación e execución das seguintes frases en OCaml (con tipos e valores), como o indicaría o compilador interactivo. Distinga claramente expresións de definicións.

let clasif p x (pos,neg) = if p x then (x::pos,neg) else (pos,x::neg);;

let divide p l = List.fold_right (clasif p) l ([] , []);;

divide (fun x -> x mod 2 = 0) [1;2;3;4;5];;

3. (1 punto) Defina de xeito recursivo terminal unha función `last_element: 'a list -> 'a` que devolva o último elemento da lista á que se aplique. Se a lista está baleira debe provocar a excepción `Invalid_argument "last_element"`.

4. (1 punto) Defina unha función (procedimento) `output_multiples: int -> int -> int -> unit` de xeito que `output_multiples x a b` "imprima" na saída estándar todos os múltiplos de `x` no intervalo $[a, b]$, un por línea.

5. O tipo de dato '`a clist`', definido a continuación, serve para representar listas de valores de tipo '`a`'. Basta con considerar que `Empty` representa a lista baleira, que `Single x` representa a lista cun único elemento con valor `x`, e que `App (l1,l2)` representa a concatenación das listas `l1` e `l2` (se `l1` e `l2` representan listas do mesmo tipo).

```
type 'a clist = Empty | Single of 'a | App of 'a clist * 'a clist
```

Con esta representación, existirían distintos valores de tipo '`a clist`' que representarían a mesma lista de valores de tipo '`a`'. Así, por exemplo, `App (App (Single 1, Single 2), Single 3)` representaría a mesma lista que `App (Single 1, App (Single 2, Single 3))`; como tamén o farián `Empty` e `App (Empty, Empty)`.

(0,5 puntos) Defina unha función `is_empty: 'a clist -> bool` que sirva para determinar se un valor deste tipo representa a lista baleira.

(0,5 puntos) Con esta representación pódese implementar a concatenación de listas de xeito que se avalie en tempo constante. Píagao cunha función *append* : '*a clist* -> '*a clist* -> '*a clist*

(0,5 puntos) Defína tamén as funcións *hd*: '*a clist* -> '*a* e *tl*: '*a clist* -> '*a clist*, que sirvan para obter, respectivamente a cabeza e a cola dunha lista. Estas funcións deben provocar excepcións *Invalid_argument* se se aplican a un argumento inadecuado.

(0,5 puntos) Defína unha función *eq_clist* : '*a clist* -> '*a clist* -> *bool* que sirva para determinar se dous valores de tipo '*a clist* representan a mesma lista.

PARADIGMAS DE PROGRAMACIÓN

XANEIRO 2016: Parte de obxectos / ENERO 2016: Parte de objetos

NOME/ NOMBRE: _____

APELIDOS/APELLIDOS: _____

[0.6 puntos] EXERCICIO / EJERCICIO 1:

Indique cales das seguintes definicións son correctas ou non. Se son correctas, indique o resultado que devolvería o compilador interactivo (tipos e valores, clases inclusive). Se non o son, indique brevemente por qué.

Indique cuáles de las siguientes definiciones son correctas o no. Si son correctas, indique el resultado que devolvería el compilador interactivo (tipos y valores, clases inclusive). Si no lo son, indique brevemente por qué.

```
class claseA =  
object (self)  
    val atributo1 = 10  
    method metodo1 () = 1 + atributo1  
    method metodo2 () = self#metodo1() + 1  
end;;
```

```
class claseA =  
object  
    val atributo1 = 10  
    method metodo1 () = 1 + atributo1  
    method metodo2 () = metodo1() + 1  
end;;
```

```
class claseA =  
object  
    val atributo1 = 10  
    method metodo1 () = 1 + atributo1  
    method metodo2 () = this#metodo1() + 1  
end;;
```

[0.6 puntos] EXERCICIO / EJERCICIO 2:

```
class claseJ =  
object  
    val atributo1 = [10; 20; 30]  
    method metodo1 () = atributo1  
    method metodo2 () = List.hd atributo1  
    method metodo3 () = List.tl atributo1  
end;;
```

```
class claseK =  
object  
    val atributo1 = [10; 20; 30]  
    method metodo1 () = atributo1  
    method metodo2 () = List.hd atributo1  
    method metodo3 () = List.tl atributo1  
end;;
```

```
class subclaseJ6 (m,n) =  
object  
    inherit claseJ as super  
    val atributo1 = [m; n]  
end;;  
  
let j = new claseJ;;  
let k = new claseK;;  
let sj62 = new subclaseJ6;;
```

Dadas as definicións anteriores, indique cales das seguintes listas son ou non válidas. No caso de serlo, indique o seu tipo resultante; no caso de non serlo, indique brevemente por qué.

Dadas las definiciones anteriores, indique cuáles de las siguientes listas son o no válidas. En el caso de serlo, indique su tipo resultante, en el caso de no serlo, indique brevemente por qué.

[j; sj62 (0,0)];;

[k; sj62 (0,0)];;

k::List.tl [sj62 (0,0)];;

[0.4 puntos] EXERCICIO / EJERCICIO 3:

```
class claseR =
object (this)
  val mutable atr1 = [1; 2; 3; 4; 5]
  method metodo1 () = atr1
  method metodo2 () = List.hd (this#metodo1())
  method metodo3 () = List.tl (this#metodo1())
end;;
```

```
class subclaseR1 pcl =
object (this)
  inherit claseR as super
  val mutable atr1 = pcl
  method metodo4 pmi = atr1 <- pmi::atr1
end;;
```

Dadas as definicións anteriores, indique cales das seguintes funcións aceptarían como entrada unha instancia da clase `claseR`. No caso daquelas que a aceptarian, indique cal sería o valor obtido á saída e o seu tipo. No caso de que as funcións non as aceptasen, explique brevemente por que.

Dadas las definiciones anteriores, indique cuáles de las siguientes funciones aceptarían como entrada una instancia de la clase `claseR`. En el caso de aquéllas que la aceptarían, indique cuál sería el valor obtenido a la salida y su tipo. En el caso de que las funciones no las aceptasen, explique brevemente por qué.

```
let f3 (x: <metodo1: unit->int list; metodo2: unit->int; metodo3: unit->int list>) = x#metodo2();;
```

```
let f4 (x: <metodo1: unit->int list; metodo2: unit->int; metodo3: unit->int list; ..>) = x#metodo2();;
```

[0.4 puntos] EXERCICIO / EJERCICIO 4:

```
class ('a, 'b) claseS (pcl: 'a) =
object (this)
  val mutable arreglo = [|pcl; pcl|]
  val mutable listilla = []
  method get_arreglo = arreglo
  method get_lista: 'b list = listilla
  method set_arreglo pmi = arreglo.(0) <- pmi; arreglo.(1) <- pmi
  method set_lista pmi = listilla <- pmi
  method add_lista pmi = listilla <- pmi::listilla
  method resetear_lista () = listilla <- []
end;;
```

Dada a definición anterior, e sen redefinir métodos nin atributos, extenda `claseS` nunha subclase `subclaseS1` de tal xeito que o atributo `listilla` sexa sempre de tipo lista de pares de enteros. Indique o resultado devolto polo compilador interactivo (constructor, tipos de atributos, métodos, etc.) para tal definición.

Dada la definición anterior, y sin redefinir métodos ni atributos, extienda `claseS` en una subclase `subclaseS1` de tal forma que el atributo `listilla` sea siempre de tipo lista de pares de enteros. Indique el resultado devuelto por el compilador interactivo (constructor, tipos de atributos, métodos, etc.) para tal definición.