Exámenes 2011, preguntas y respuestas

Paradigmas de Programación (Universidade da Coruña)

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 de1a n = genlist id n
            in de1a 4, de1a 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, cambiándo 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 -> '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: flota = 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);;
```

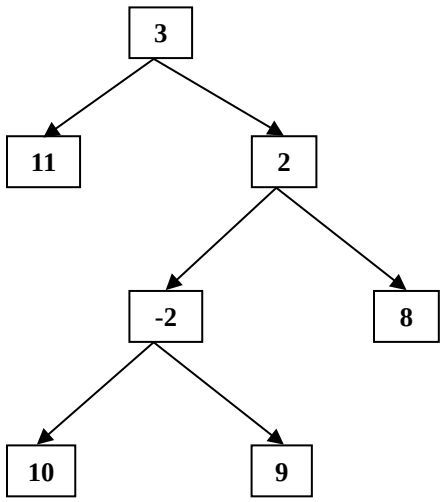
4. (3 ptos.) Considere la siguiente definición del tipo de dato **'a tree** que podría servir para representar en ocaml cierto tipo de árboles binarios:

```
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 ejercicions 2, 3 y 4 deben realizarse de la forma más sencilla posible.