Exámen enero 2014, preguntas y respuestas

Paradigmas de Programación (Universidade da Coruña)

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 defición 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)

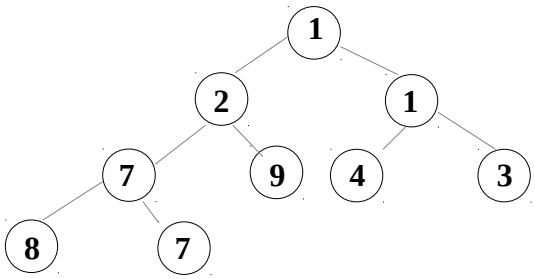
```
let segcr l =
    let rec segcr1 n s = function
        [] -> List.rev s
        | h::[] -> segcr1 [] ((n@[h])::s) []
        | h::t -> if ((h) < (List.hd t)) then (segcr1 (n@[h]) s t) else
        (segcr1 [] ((n@[h])::s) t)
    in segcr1 [] [] l;;

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 (<=).



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

…
```