Paradigmas de Programación
Práctica 7

Salvo indicación en sentido contrario, las definiciones que se piden en esta práctica deben incluirse en un archivo “sort.ml”.

1. Considere la siguiente implementación, en OCaml, del método de ordenación por inserción.

```
let rec insert x = function
     [] -> [x]
   | h::t -> if x <= h then x :: h :: t
             else h :: insert x t

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

Ni la función *insert* ni *isort* están definidas de modo recursivo terminal. Defina un valor ***bigl: int list*** tal que *isort bigl* provoque un error de ejecución “stack overflow” (se supone que se ejecutaría en la máquina virtual de OCaml con la configuración por defecto del stack propio de la versión 4 del compilador).

Defina las funciones ***insert_t: ‘a -> ‘a list -> ‘a list*** e ***isort_t: ‘a list -> ‘a list*** de modo que resulten implementaciones recursivas terminales de la funciones *insert* e *isort*, respectivamente.

Compruebe que ahora no se produce ningún error al aplicar la función *isort_t* al valor *bigl.*

Utilice la función *Random.int* para definir una (“falsa”) función ***rlist: int -> int list*** tal que, para cada $n \geq 0$, *rlist n* resulte ser una lista “aleatoria” con *n* valores de tipo *int*. Hágalo de modo que la lista resulte convenientemente “dispersa” (esto es, que no contenga muchos valores repetidos).

Use la función crono definida a continuación para comprobar cómo crece el tiempo de ejecución de *isort* e *isort_t* con la longitud de la lista. Para ello: defina ***lc1*** y ***lc2*** como listas crecientes de enteros con *10.000* y *20.000* elementos respectivamente; defina ***ld1*** y ***ld2*** como listas decrecientes de enteros con *10.000* y *20.000* elementos respectivamente, y ***lr1*** y ***lr2*** como listas “aleatorias” con *10.000* y *20.000* enteros respectivamente. Compruebe, en primer lugar, que, con cualquiera de esos valores, las funciones *isort* e *isort_t* producen el mismo resultado. Compruebe, entonces, en cada uno de esos 3 casos cómo se comporta el tiempo de ejecución al duplicar el tamaño de la lista. Explique (como comentario dentro del archivo sort.ml) cómo se justifica ese comportamiento. Compare el tiempo de ejecución de

isort lr2 e *isort_t lr2*. Indique (como comentario dentro del archivo sort.ml) si hay una diferencia apreciable y, en ese caso, a qué puede deberse.

```
let crono f x =
    let t = Sys.time () in
    let _ = f x in
    Sys.time () -. t
```

Defina una función ***isort_g: (‘a -> ‘a -> bool) -> ‘a list -> ‘a list*** que proporcione una implementación recursiva terminal del algoritmo de ordenación por inserción y que tome como argumento la relación de orden que se desea emplear para la ordenación.

Considere la siguiente implementación, en OCaml, del método de ordenación por mezcla (o fusión).

```
let rec split l = match l with
    h1::h2::t -> let t1, t2 = split t
                 in h1::t1, h2::t2
  | _ -> l, []

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)

let rec msort l = match l with
    [] | [_] -> l
  | _ -> let l1, l2 = split l
         in merge (msort l1, msort l2)
```

Compruebe que *msort* produce el mismo resultado que *isort* en todas las listas utilizadas previamente *(lc1, lc2, ld1, ld2, lr1* y *lr2).*

Ninguna de estas tres funciones *(split, merge* y *msort)* está definida de modo recursivo terminal. Defina un valor ***bigl2: int list*** tal que al aplicarle la función *msort* se produzca un error “stack overflow”.

Defina ***split_t: ‘a list -> ‘a list * ‘a list*** y ***merge_t: ‘a list * ‘a list -> ‘a list***, versiones recursivas terminales de split y merge, respectivamente. Atención: no es necesario que split_t dé el mismo resultado que split (cuando se aplican a una misma lista); es suficiente con que al reunir los elementos del par de listas que devuelve se obtenga los mismos elementos que hay en la lista a la que se aplica (no necesariamente en el mismo orden, pero sí respetando el número de apariciones de cada valor). La función merge, cuando se aplica

a un par de lista ordenadas, debe devolver una lista ordenada que reúna los elementos de ambas componentes del par (respetando el número de apariciones de cada valor).

Defina una nueva versión de *msort*, ***msort': 'a list -> 'a list***, que utilice *split_t* y *merge_t* (en vez de *split* y *merge*).
Compruebe que *msort'* no provoca "stack overflow" al aplicarla al valor *bigl2*. Intente definir una valor ***bigl3: int list***, tal que, al aplicarle la función *msort',* provoque "stack overflow". Si no es posible, defina *bigl3* como la lista vacía e intente explicar por qué, no siendo recursiva terminal la definición de *msort',* no se produce "stack overflow".

Compruebe los tiempos de ejecución de la aplicación de *msort'* a los valores *lc1, lc2, ld1, ld2, lr1* y *lr2* y explique (como comentario dentro del archivo sort.ml) los resultados obtenidos.

Compare los tiempos de ejecución de *msort* y *msort'* a *lr1* y *lr2.* Explique los resultados obtenidos

Por último implemente una versión ***msort_g: ('a -> 'a -> bool) -> 'a list -> 'a list*** de la ordenación por fusión que tome como argumento la relación de orden a emplear.

El fichero sort.ml debería compilar sin errores, con el archivo sort.mli proporcionado, con la orden

```
ocamlc -c sort.mli sort.ml
```

2. (Ejercicio opcional) Redefina en un fichero “tail.ml” las siguientes funciones de modo que no se utilice, en ningún momento, recursividad no terminal:

```
let rec to0from n =
    if n < 0 then []
    else n :: to0from (n-1)

let rec fromto m n =
    if m > n then []
    else m :: fromto (m+1) n

let rec remove x = function
    [] -> []
  | h::t -> if x = h then t
            else h :: remove x t

let rec compress = function
    h1::h2::t -> if h1 = h2 then compress (h2::t)
                 else h1 :: compress (h2::t)
  | l -> l

let append' = List.append

let map' = List.map

let fold_right' = List.fold_right

let incseg l =
    List.fold_right (fun x t -> x::List.map ((+) x) t) l []
```

No descarte el uso de funciones recursivas terminales del módulo *List*, como *init, fold_left, rev, rev_append, rev_map,* etc.

El archivo tail.ml debería compilar sin errores, con el archivo tail.mli proporcionado, con la orden:

```
ocamlc -c tail.mli tail.ml
```