Paradigmas de Programación

Práctica 9

Ejercicios:

1. Redefina en un fichero `ej91.ml` las siguientes funciones de modo que no se utilice 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 incseg l =
  List.fold_right (fun x t -> x::List.map ((+) x) t) l [];;

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;;
```

   El fichero `ej91.ml` debe compilar sin errores con la orden `ocamlc -c ej91.mli ej91.ml`.

2. Considere la siguiente implementación del algoritmo *quicksort*:

```
let rec qsort1 ord = function
    [] -> []
  | h::t -> let after, before = List.partition (ord h) t in
            qsort1 ord before @ h :: qsort1 ord after;;
```

   ¿En qué casos no será bueno el rendimiento de esta implementación? Para evitar problemas con la no terminalidad de `(@)` podemos hacer el siguiente cambio:

```
let rec qsort2 ord =
  let append' l1 l2 = List.rev_append (List.rev l1) l2 in
  function
      [] -> []
    | h::t -> let after, before = List.partition (ord h) t in
              append' (qsort2 ord before) (h :: qsort2 ord after);;
```

   ¿Tiene `qsort2` alguna ventaja sobre `qsort1`? ¿Permite `qsort2` ordenar listas que no podrían ordenarse con `qsort1`? En caso afirmativo, defina un valor `l1 : int list` que sea ejemplo de ello. En caso negativo, defina `l1` como la lista vacía.

   ¿Tiene `qsort2` alguna desventaja sobre `qsort1`? Compruebe si `qsort2` es más lento que `qsort1`. Si es así, explique por qué y estime la penalización, en porcentaje de tiempo usado, de `qsort2` respecto a `qsort1`.

Realice las implementaciones de este ejercicio en un fichero `qsort.ml`. Como siempre, las respuestas "de palabra" que se piden en algunos de los apartados deben ser incluidas como comentarios en este mismo fichero. El fichero debe compilar sin errores con la orden `ocamlc -c qsort.mli qsort.ml`.

3. Considere la siguiente implementación de la ordenación por fusión:

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

let rec merge = function
    [], l | l, [] -> l
  | h1::t1, h2::t2 -> if h1 <= h2 then h1 :: merge (t1, h2::t2)
                      else h2 :: merge (h1::t1, t2);;

let rec msort1 l = match l with
    [] | _::[] -> l
  | _ -> let l1, l2 = divide l in
         merge (msort1 l1, msort1 l2);;
```

¿Puede provocar algún problema la no terminalidad de `divide` o `merge`? En caso afirmativo, defina un valor `l2 : int list` que sea un ejemplo de ello. En caso negativo, defina `l2` como la lista vacía.

Defina de modo recursivo terminal una función `divide'` que cumpla el mismo cometido que `divide`.

Defina de modo recursivo terminal una función `merge'` que cumpla el mismo cometido que `merge`, y que además tenga tipo `('a -> 'a -> bool) -> 'a list * 'a list -> 'a list`, de modo que pueda ser utilizada con cualquier orden (y no solo con `(<=)`).

Defina una función `msort2` con tipo `('a -> 'a -> bool) -> 'a list -> 'a list`, que realice la ordenación por fusión mediante el uso de `divide'` y `merge'`.

Compare el rendimiento en tiempo de ejecución de `msort2` con el de `msort1` y con el de `qsort2`.

Realice las implementaciones de este ejercicio en un fichero `msort.ml`. Como siempre, las respuestas "de palabra" que se piden en algunos de los apartados deben ser incluidas como comentarios en este mismo fichero. El fichero debe compilar sin errores con la orden `ocamlc -c msort.mli msort.ml`.

4. (Ejercicio opcional) Observe los siguientes ejemplos de ejecución del programa `fact` del ejercicio 3 de la práctica 2:

```
$ ./fact 10
3628800

$ ./fact
fact: número de argumentos inválido

$ ./fact -1
Fatal error: exception Stack_overflow
```

```
$ ./fact a
Fatal error: exception Failure("int_of_string")
```

Reescriba el fichero `fact.ml` para que la función `fact` no acepte argumentos negativos, y para que se intercepten con frases `try-with` las excepciones asociadas a los errores de ejecución que se puedan producir en las dos últimas situaciones, de forma que el programa no aborte descontroladamente, y el programador tenga la oportunidad de finalizar la ejecución de manera elegante. El nuevo comportamiento en esos dos casos debe ser el siguiente:

```
$ ./fact -1
fact: argumento inválido
```

```
$ ./fact a
fact: argumento inválido
```