Práctica 9

En el ejercicio *Más Hanoi*, propuesto como práctica competitiva, se utiliza una terna de listas de enteros para representar el estado, en cualquier momento, del puzzle “Torres de Hanoi”. En el mismo enunciado se describe lo que se entiende por un “estado estable”.

1. Defina en Ocaml una función ***is_stable : int list * int list * int list -> bool*** que sirva para determinar si una de estas ternas representa (o no) correctamente un estado estable del juego con n discos. Guarde esta definición en un archivo con nombre ***hanoi_plus.ml***

```
# is_stable ([1;2;3],[4;5],[6]);;
- : bool = true

# is_stable ([1;3;2],[4;5],[6]);;
- : bool = false

# is_stable ([1;2],[4;5],[6]);;
- : bool = false

# is_stable ([1;2;3],[2],[]);;
- : bool = false
```

Para poder realizar pruebas intensivas con las propuestas presentadas por participantes en la prueba, se pretende comprobar su funcionamiento en todos los casos posibles para un determinado número de discos.

2. Defina una función ***all_states: int -> (int list * int list * int list) list*** , de modo que ***all_states n*** dé la lista de todos los estados estables posibles para el juego con ***n*** discos. Añada esta definición al archivo ***hanoi_plus.ml.***

```
# all_states 0;;
- : (int list * int list * int list) list = [([], [], [])]

# all_states 1;;
- : (int list * int list * int list) list =
[([1], [], []); ([], [1], []); ([], [], [1])]

# all_states 2;;
- : (int list * int list * int list) list =
[([1; 2], [], []); ([2], [1], []); ([2], [], [1]); ([1], [2], []);
 ([], [1; 2], []); ([], [2], [1]); ([1], [], [2]); ([], [1], [2]);
 ([], [], [1; 2])]
```

No es necesario que la función definida por usted devuelva los estados en el mismo orden que aparecen en el ejemplo. Fíjese en que la lista devuelta por ***all_states n*** debería tener exactamente 3^n elementos; así que no espere poder calcularla para valores muy grandes de ***n*** (pero sí aún para, por ejemplo, *14* discos, ya que la lista tendría entonces *sólo 4.782.969* estados distintos)

En el mismo enunciado se define un tipo de dato para representar los distintos movimientos que pueden realizarse durante el desarrollo del juego

```
type move = LtoC | LtoR | CtoL | CtoR | RtoL | RtoC
```

3. Incluya esta definición en el archivo ***hanoi_plus.ml*** y añada la definición para una función ***move : int list * int list * int list -> move -> int list * int list * int list*** de modo que al aplicarla a un estado estable y uno de los movimientos posibles, devuelva el nuevo estado en el que quedaría el juego si se pudiese aplicar (de modo legal) ese movimiento en el estado dado. La función move debe provocar la excepción ***Invalid_argument "move"*** si el estado inicial no es estable o si el movimiento resultase ilegal.

```
# move ([1;2;3],[4;5],[6]) LtoC;;
- : int list * int list * int list = ([2; 3], [1; 4; 5], [6])

# move ([1;2;3],[4;5],[6]) CtoL;;
Exception: Invalid_argument "move".
```

pone Exception: Invalid_argument move

```
# move ([1;2;3],[4;5],[6]) CtoR;;
- : int list * int list * int list = ([1; 2; 3], [5], [4; 6])

# move ([1;2;3],[4;5],[6]) RtoL;;
Exception: Invalid_argument "move".
```

4. Defina, también en el archivo ***hanoi_plus.ml***, una función
move_sequence : int list * int list * int list -> move list -> int list * int list * int list de modo que ***move_sequence ini moves*** devuelva el estado al que se llegaría, partiendo del estado ***ini***, después de aplicar la secuencia de movimientos ***moves***. Si el estado inicial no es estable o si alguno de los movimientos de la secuencia no puede aplicarse legalmente, la función debe provocar la excepción ***Invalid_argument "move_sequence"***.

```
# move_sequence ([2; 3], [1; 4; 5], [6]) [CtoR; LtoC; RtoL];;
- : int list * int list * int list = ([1; 3], [2; 4; 5], [6])
```

5 (Opcional). En un examen en el que se pedía una implementación recursiva terminal de la función concat del módulo List, algunos alumnos hicieron la siguiente definición[1]:

```
let concat l =
    let rec aux acc = function
        [] -> acc
      | h::t -> aux (List.append acc h) t
    in aux [] l
```

Si bien su lógica es impecable, la aplicación de la función definida de esta forma resulta muy ineficiente si la lista a la que se aplica es larga (y las listas que la componen no son vacías). Compruebe que los valores que devuelva al aplicarla son los mismos que devuelve la función List.concat, pero que, si las listas son largas, puede tardar mucho más tiempo. Puede probar a aplicarla, por ejemplo, a la lista definida a continuación (que tiene 50.000 elementos)

```
let l = List.init 50_000 (fun i -> [i])
```

Averigüe por qué sucede esto, explíquelo (como comentario) en un archivo con nombre ***tail.ml*** e incluya en el mismo archivo una nueva definición recursiva terminal para la función ***concat*** que no presente este problema.

Redefina también las siguientes funciones de modo **recursivo terminal**[2]. Las nuevas definiciones deben incluirse también en el archivo ***tail.ml.*** Cuando sea posible, puede utilizar, si lo desea, la transformación *"Tail Modulo Constructor"* descrita en el manual de OCaml

```
let rec front = function
    [] -> raise (Failure "front")
  | h::[] -> []
  | h::t -> h::front t

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

let rec ofo = function
     [] -> []
   | h::t -> h :: List.filter ((<>) h) (ofo t);;
```

[1] Tenga en cuenta que, a partir de la versión 5.1 de Ocaml, la implementación de List.append es recursiva terminal.
[2] Para este ejercicio puede utilizar List.append y List.map como si fuesen recursivas terminales (como de hecho lo son desde la versión 5.1).

```
let fold_right' = List.fold_right

let rec sublists = function
    [] -> [[]]
  | h::t -> let st = sublists t in
            st @ List.map (fun l -> h::l) st;;
```