Paradigmas de Programación
Práctica 6

En Ocaml, cualquier valor de tipo α *list* puede "construirse", a partir de los valores de tipo α, utilizando únicamente la lista vacía ([ ]), y el constructor "cons" (::)
Si *h* : α y *t* : α *list,* entonces (*h :: t*) : α *list* y representa la lista que tiene cabeza *h* y cola *t* (toda lista no vacía tiene cabeza y cola, y dos listas son iguales si ambas son la lista vacía o si tienen la misma cabeza y la misma cola).

En realidad, cuando escribimos, por ejemplo, *[1; 2; 3]*, estamos usando una manera abreviada (o un "pretty-print") para escribir *1 :: 2 :: 3 :: [ ]*
Nótese que, sintácticamente, :: es asociativo por la derecha. Esto es, *1 :: 2 :: 3 :: [ ]* es lo mismo que *1 :: (2 :: (3 :: [ ]))*

Decimos que [ ] y :: son los "constructores" de listas y, al igual que el constructor de pares, la coma (,), pueden utilizarse en el *pattern-matching*.

Así, por ejemplo, podemos escribir la siguiente definición:

```
let hd = function h :: _ -> h
```

O, abreviadamente,

```
let hd (h :: _ ) = h
```

(Nótese que esta definición de hd provoca una advertencia del compilador al no ser exhaustiva; pero es que esta función no está definida en la lista vacía)

De modo similar, podríamos escribir la siguiente definición:

```
let rec length = function
    [] -> 0
  | _::t -> 1 + length t
```

Todas las funciones del módulo List, incluida la función append, que es lo mismo que el operador (@) del módulo Stdlib, pueden definirse utilizando sólo los constructores de listas y pattern-matching.

En esta práctica trataremos de implementar, como ejercicio, muchas de las funciones incluidas en el módulo List. **No está permitido, por lo tanto, utilizar ese módulo ni el operador (@)**.

Para cada una de las funciones enumeradas a continuación intente, en un primer momento, realizar una definición lo más sencilla posible. Después averigüe en el manual de OCaml[1] si la implementación de esa función en el módulo List es o no recursiva terminal (en el manual se indica explícitamente siempre que una implementación no lo es). Si la implementación en el módulo List es recursiva terminal y la realizada por usted no lo es, intente redefinir la función de modo recursivo terminal (en ese caso conserve ambas definiciones con el mismo nombre; la recursiva terminal deberá aparecer, en el código a entregar, después de la no terminal.

Es importante que las funciones definidas por usted se comporten exactamente como las originales del módulo List. Es decir, deben devolver exactamente el mismo valor que devuelven las correspondientes del módulo List (cuando estas devuelven un valor) y deben provocar un error de ejecución también cuando estas lo provocan (si bien, por el momento, no es necesarios que "el mensaje que se muestra acompañando al error de ejecución" sea el mismo)

Así, por ejemplo, la definición aportada en este enunciado para la función *hd* es perfectamente válida para el ejercicio y la definición aportada para la función *length* sería válida como primera definición, pero habría que redefinirla luego de modo recursivo terminal.

Preste también atención a la complejidad computacional de sus implementaciones, procurando que sea la menor posible.

Todas las definiciones deberán escribirse en un archivo de nombre "mylist.ml". Este archivo debe compilar sin errores con la interfaz "mylist.mli", suministrada con este enunciado, con la orden:

ocamlc -c mylist.mli mylist.ml

La lista de funciones que debe implementar en este ejercicio puede verse en el archivo *mylist.mli.* Utilice el manual de OCaml y el compilador interactivo para averiguar cómo debe comportarse exactamente cada función.

En algunas de las definiciones que se piden puede ser cómodo utilizar expresiones match...with…
Estas frases en OCaml tienen la siguiente estructura

`match <e> with <p1> -> <e1> | <p2>  -> <e2> |...| <pn> -> <en>`

[1] Para este propósito utilice el manual de una versión del compilador anterior a la 5.01, ya que desde esta versión pasa a ser recursiva terminal la implementación de algunas funciones que antes no lo era.
Para que quede más claro: las funciones que deben implementarse de modo recursivo terminal en este ejercicio son: length, compare_lengths, compare_length_with, init, nth, rev_append, rev, rev_map, for_all, exists, mem, find, find_all, filter, partition, fold_left, assoc y mem_assoc.

donde <e> es cualquier expresión válida en OCaml, las $<e_i>$ son cualesquiera expresiones válidas en Ocaml pero todas del mismo tipo, y los $<p_i>$ son patrones patrones cuya forma ha de ser compatible con el tipo de <e>. Esta expresión se evaluaría exactamente igual que

(function $<p_1>$ -> $<e_1>$ | $<p_2>$ -> $<e_2>$ |...| $<p_n>$ -> $<e_n>$) <e>

Así, por ejemplo, podríamos escribir la siguiente definición:

```
let rec compare_lengths l1 l2 = match (l1, l2) with
    ([], []) -> 0
  | ([], _) -> -1
  | (_, []) -> 1
  | (_::t1, _::t2) -> compare_lengths t1 t2
```