Listas y Excepciones

Cuando se produce un error (*exception*) durante la ejecución de un programa OCaml, el programa asocia con el error un valor de tipo ***exn***, que sirve para clasificarlo y, eventualmente, interceptarlo o procesarlo.

Division_by_zero, Failure "int_of_string", Match_failure ("//toplevel//", 1, 7) … son ejemplos de valores de tipo ***exn*** en OCaml. *Division_by_zero, Failure, Match_failure, Invalid_argment,* etc., son ***constructores*** de valores de tipos ***exn***. Algunos de estos constructores requieren un argumento de un tipo determinado.

La función ***raise : exn -> 'a***, cuando se aplica a un valor ***e*** de tipo ***exn,*** provoca una excepción asociada al valor ***e***.

Ejercicio 1: Redefina la función ***mcm'*** del ejercicio 1 de la práctica 6, de modo que, en caso de error, en vez de devolver el valor (-1), provoque una excepción asociada al valor ***Invalid_argument "mcm': argumento inválido o mcm excede max_int"*** (Guarde esta definición en un archivo con nombre ***mcm_plus.ml***)

El manejo de listas es otro de los (muchos) puntos fuertes de OCaml. Las listas sirven para representar secuencias de valores. En OCaml las listas son homogéneas (i. e., todos los valores de la secuencia han de ser del mismo tipo). Si los valores que componen la lista son de tipo ***τ*** , la lista es de tipo ***τ list***. Si ***τ*** es un tipo en OCaml, ***τ list*** también lo es (no sería correcto decir que ***list*** es un tipo: es un ***constructor de tipos***). En Ocaml las listas representan secuencias finitas[1] (i. e, tienen un número finito de elementos).

Como el resto de los valores que se utilizan en programación funcional, las listas son inmutables (i. e., no se puede *añadir*, *quitar* ni *modificar* un elemento a o de una lista)

Todos los valores de tipo ***τ list*** se construyen utilizando la lista vacía ***([ ] : 'a list)***, el constructor **::** (“cons”), y los valores de tipo ***τ.***

[1] Esto realmente habría que matizarlo; pero no es ahora el momento…

<head> :: <tail> representa la lista con cabeza <*head*> y cola <*tail*> ; si ***<head>*** es de tipo ***τ***, <*tail*> debe ser necesariamente de tipo ***τ list***

El constructor de listas (::) es sintácticamente asociativo por la derecha.
La expresión `['a'; 'e'; 'i'; 'o'; 'u']`
no es más que un *pretty-print* para `'a'::'e'::'i'::'o'::'u'::[]`

El módulo ***List*** de la librería *Standard* de OCaml contiene una gran variedad de funciones útiles para el manejo de listas; pero todas ellas podrían ser definidas directamente utilizando sólo los constructores de listas (el *cons* y la *lista vacía*).

En el siguiente ejercicio le pediremos que defina usted mismo muchas de estas funciones. Se trata de respetar los nombres, los tipos y el significado de las versiones originales definidas en el módulo *List*, pero (lógicamente, para que tenga sentido este ejercicio) no pueden usarse las definiciones del módulo *List* (ni el operador de concatenación de listas, *(@)*, incluído en el módulo *Stdlib*).

El ejercicio tiene 2 objetivos fundamentales. Por un lado, el de conocer algunas de las funciones más útiles del módulo List; y, por otro, realizar una práctica de implementación de definiciones recursivas, muy interesante por sí misma.

Cuando las funciones del módulo *List* puedan provoquen excepciones al aplicarse, las funciones definidas por usted durante el ejercicio deben también provocar las mismas excepciones.

Se sugiere encarecidamente que consulte la sección del Manual de OCaml correspondiente al módulo *List*[2], para comprender exactamente el comportamiento de las funciones que se soliciten. Use también el compilador interactivo ***ocaml*** para aclarar las dudas que puedan quedar.

Ejercicio 2: Incluya, en un archivo con nombre **myList.ml**, las definiciones necesarias para satisfacer la interfaz ***myList.mli*** proporcionada junto a este enunciado.

En algunos casos se solicitan 2 versiones para la misma función. La primera, identificada con el nombre exacto de la función correspondiente del módulo ***List***, debe ser realizada con la definición más sencilla y directa posible; la segunda, diferenciada con un apóstrofe al final

[2] Por ejemplo, si tiene la version 4.14 del compilador, https://ocaml.org/manual/4.14/api/List.html

del nombre, debe implementarse con recursividad terminal (de modo que nunca provoque un error *Stack Overflow*), aunque, aparte de eso, se mantenga la sencillez de la definición en la medida de lo posible.

En todo caso, las implementaciones deben tener una complejidad computacional (en términos de tiempo de ejecución y espacio necesario) similar a la de las versiones del módulo *List*.

Para distinguir casos, intente utilizar *pattern-matching*.

Aunque está permitido el uso de las funciones que vaya definiendo en las definiciones posteriores, intente no abusar de ello (en muchos casos es casi más sencillo - y a veces menos costoso - no hacerlo).

Para escribir algunas de las definiciones puede ser más cómodo utilizar expresiones *"match .. with .."* como, por ejemplo, en la definición de la función ***append***:

```
let rec append l1 l2 = (* not tail recursive *)
    match l1 with
      [] -> l2
    | h::t -> h :: append t l2
```

Para evaluar una expresión como `match` *<e> with <p₁> -> <e₁>* | … | *<pₙ> -> <eₙ>* , se comienza evaluando la expresión <e> y luego se busca la primera de las reglas que siguen a la palabra reservada ***with*** en la que el patrón a la izquierda de la flecha coincida con el valor de <e>; entonces, se evalúa la expresión a la derecha de la flecha en esa misma regla y su valor es el de la expresión *"match .. with .."*. Si ningún patrón encaja con el valor de <e> se produce una excepción por "fallo de match".

Tenga en cuenta que *<e>* puede ser de cualquier tipo; pero los patrones *<pᵢ>* deben tener todos una forma compatible con el tipo de *<e>*. Las *<ei>* pueden ser de cualquier tipo (no necesariamente el mismo que <e>); pero todas ellas del mismo.

Una vez completado el archivo ***myList.ml*** debería poder compilarlo con la orden

```
ocamlc -c myList.mli myList.ml
```