Paradigmas de Programación

Variables

Estructura:

```
let identificador = expresion
(* identificador es el nombre de la variables expresion puede ser una función, un int, un string... *)
```

Ejemplos de tipos de datos:

```
();;
(* - : unit = () *)

2 + 5 * 3;;
(* - : int 17 *)

1.0;;
(* - : float = 1. *)

5 / 3;;
(* - : int = 1 *)
(* Estamos haciendo una divisíon entera *)

5 mod 3;;
(* - : int = 2 *)
(* Devolverá el resto de hacer 5 entre 2 *)

3.0 *. 2.0 ** 3.0;;
(* - : float = 24 *)
(* 3 * 2^3 = 3 * 8 = 24 *)

3.0 = float_of_int 3;;
(* - : bool = true *)
(* Comprueba si el flotante de 3 es idéntico a 3.0 *)

sqrt 4.;;
(* - : float = 2. *)
(* sqrt necesita un flotante sí o sí *)

int_of_float 2.1 + int_of_float (-2.9);;
(* - : int = 0 *)
(* Convierte el flotante 2.1 y -2.9 a entero y los suma *)

truncate 2.1 + truncate(-2.9);;
(* - : int = 0 *)
(* Redondea para abajo los positivos y para arriba los negativos y opera *)

floor 2.1 +. floor (-2.9);;
(* - : int = -1 *)
```

```
(* Rendondea para abajo y opera *)

ceil 2.1 +. ceil (-2.9);;
(* - : float = 1. *)
(* ceil redondea siempre para arriba *)

2.0 ** 3.0 ** 2.0;;
(* - : float = 512*)
(* Opera las potencias, 2^3^2, torre de pontencias *)

Char.code 'B';;
(* Mostrará el código del caracter 'B' *)
(* - : int = 66 *)

Char.chr 67;;
(* Mostrará el caracter del código 67 *)
(*- : char = 'C'*)

"this is a string";;
(* - : string = "this is a string" *)

String.length "longitud";;
(* - : int = 8*)
(* Devuelve cuantas letras hay en la cadena *)

"1999" ^ "1";;
(* - : string = "19991" *)
(* El operador ^ sirve para concatenar strings *)

string_of_int 010;;
(* - : string = "10" *)
(* Pasa el número a string, los ceros iniciales los descarta *)

not true;;
(* - : bool = false *)

true && false;;
(* - : bool = false *)

true || false;;
(* - : bool = true *)

(1 < 2) = false;;
(* - : bool = false *)
(* Comprueba si 1 < 2 es falso *)

"1" < "2";;
(* Compureba si el código de "1" es menor que el de "2" *)
(* - : bool = true *)

2 < 12;;
(* - : bool = true *)
(* Comprueba si 2 es menor que 12 *)

if 3 = 4 then "0" else "4";;
(* - : string = "4" *)
```

```
(* Vuelve a comprobar, pero devuelve un string *)

(if 3 < 5 then 8 else 10) + 4;;
(* - : int = 12 *)
(* Ejecuta primero el condicional, y después le suma el número 4 *)

function x -> 2 * x;;
(* - : int -> int = <fun> *)

(function x -> 2 * x) (2 + 1);;
(* - : int = 6 *)
(* Ejecuta la función tomando como parámetro la suma 2 + 1 *)

int_of_float;;
(* - : float -> int = <fun>*)
(* Es una función que pide un flotante para sacar su entero *)

abs;;
(* - : int -> int = <fun> *)
(* Convierte un entero en el valor absoluto *)

sqrt;;
(* - : float -> float = <fun> *)

truncate;;
(* - : float -> int = <fun> *)

ceil;;
(* - : float -> float = <fun> *)

floor;;
(* - : float -> float = <fun> *)

Char.code;;
(* - : char -> int = <fun> *)

Char.chr;;
(* - : int -> char = <fun> *)

int_of_string;;
(* - : string -> int = <fun> *)

string_of_int;;
(* - : int -> string = <fun> *)

String.length;;
(* - : String -> int = <fun> *)

let f = function x -> 2 * x;;
(* val f : int -> int = <fun> *)
(* Crea un función f que retorna el doble de un número *)

f (2+1);;
(* - : int = 6 *)

f 2 + 1;;
```

```
(* - : int = 5 *)
(* Hace la función sobre el número 2 y luego le suma 1 *)

let n = 1;;
(* val n : int = 1 *)
(* Crea la variable n *)

let g x = x + n;;
(* val g : int -> int = <fun> *)
(* Crea la función g que suma 1 al parámetro x *)

g 3;;
(* - : int = 4 *)
(* Se usa la funcion g con el parámetro 3 *)
```

Condicionales

Estructura:

```
if expresionBooleana (* Si es true, se ejecuta expresión1, si no lo es se ejecuta expresión2 *)
then expresion1
else expresion2
```

Una estructura condicional solo puede tener dos expresiones que se ejecutarán en función del resultado del booleano, sin embargo, se puede poner en expresión1 y 2 otro condicional con su respectivo then-else. Es muy importante recordad que **ambas expresiones deben retornar el mismo tipo de dato**, una no puede retornar un entero y otra un string.

```
if 3 = 3
then if 3 = 4
     then "a"
     else "b"
else "c"
(* - : string = "b" *)
```

Funciones

Estructura:

```
let identificador parametro = expresion
let identificador = function parametro -> expresion
let identificador parametro1 parametro2 ... = expresion
let identificador = function entrada1 -> salida1 | entrada2 -> salida2 | ...
```

Ejemplos:

```
(* Calcula el área de un triángulo estructurando el código de dos formas distintas *)
let areaTriangulo = function base -> function altura -> base *. altura /. 2
let areaTriangulo base altura = base *. altura /. 2

(* Si se le introduce una letra retorna true si es una vocal, y false en cualquier otro caso *)
let esVocal = function
    'a' | 'e' | 'i' | 'o' | 'u' -> true
    | _ -> false (* El _ significa: Puede ser cualquier cosa, pero da igual lo que sea porque no se usará *)
```

También se pueden hacer definiciones locales, donde se usa la palabra reservada **in** que sirve para *inicializar* una función con un valor predeterminado.

```
let sucesor x = x + 1 (* Sucesor retorna el siguiente número al introducido *)
in sucesor 3 * sucesor 4 (* Aquí lo que se hará será multiplicar el sucesor de cada uno pero solo localmente *)
(* - : int = 20 *)
```

Otras cosa que se puede hacer es meter una función en una expresión.

```
let y = 1+2 in ((function x -> x+x) y)
(* - : int = 6 *)
(* Internamente lo que está haciendo es: y = 3, entonces ese 3 se le pasa la función que en la entrada suma ese número a sí mismo, por lo que 3 + 3 = 6 *)
```

Funciones recursivas

Estructura:

```
let rec identificador parametros = expresion
(* Son las mismas estructuras que para una función normal solo que añadiendo la palabra
reservada rec al antes del identificador *)
```

Ejemplos:

```
let rec fib = function n -> match n with (* match sirve para buscar patrones *)
    0 -> 1
    | 1 -> 1
    | n -> fib(n-1) + fib(n-2)

(* Versión simplificada sin usar pattern matching*)
let rec fib = function
    0 -> 1 (* Caso base *)
    | 1 -> 1 (* Caso base *)
    | n -> fib(n-1) + fib(n-2) (* Llamadas recursivas *)
```

Listas

Estructura:

```
primerElemento::cola
primerElemento::segundoElemento::cola

(* Para construir una cola se hace: *)
let lista = [elemento1; elemento2; elemento3; ...; elementoN]
```

Una lista puede ser de cualquier tipo de dato, pero todos los elementos de esa lista deben ser del mismo tipo. Por lo que una lista de booleanos solo tendrá booleanos, una strings solo strings, una de enteros solo enteros...

Ejemplos:

```
let animales = ["perro"; "gato"; "vaca"; "cerdo"; "oveja"]

let hd = function list -> match list with (* - : string = "perro" *)
    [] -> failwith "Lista vacía"
    | head::_ -> head

let tl = function (* - : string list = ["gato"; "vaca"; "cerdo"; "oveja"] *)
    [] -> failwith "Lista vacía"
    | _::tail -> tail

let secHd = function (* - : string = "gato" *)
    [] -> raise(Failure "Lista vacía")
    | _::[] -> raise(Failure "Lista de un solo elemento")
    | _::head2::_ -> head2
```

Funciones importantes del módulo List

```
let animales = ["perro"; "gato"; "vaca"; "cerdo"; "oveja"]

length l : Número de elementos de una lista, List.length animales = 5
hd l: Primer elemento de una lista
tl l: Una lista entera sin su primer elemento
nth l N: N-ésimo elemento de una lista, siendo el primero el 0, List.nth animales 2 = "vaca"
rev l: Da la vuelta a la lista
init N f: Crea una lista de N elementos usando f para generar cada elemento, List.init 5 (fun x -> x) = [0; 1; 2; 3; 4]
append l1 l2: Une dos listas, hace lo mismo que hacer l1 @ l2, no es terminal
rev_append l1 l2: Une dos listas dándo primero la vuelta a l1, es terminal
map f l: Aplica la función f a cada elemento de l, no es terminal
rev_map f l: Aplica map y depués le da la vuelta a la lista, es terminal
fold_left f init l: Hace composición de funciones de izquierda a derecha, haciendo f (... (f (f init b1) b2) ...) bn, es terminal
fold_right f l init: Hace composición de derecha a izquierda, haciendo f a1 (f a2 (... (f an init) ...)), no es terminal
for_all f l: Comprueba que todos los elementos de l satisfacen la función booleana f
exists f l: Comprueba que al menos un elementos satisface a la función booleana f
mem a l: Retorna true si al menos un elementos de l es igual a a
find f l: Retorna el primer elemento que satisface al predicado f
filter f l: Devuelve una lista con solo los elementos que cumplen el predicado f
split a: Divide una lista de pares en dos listas, no es terminal
combine l1 l2: Transforma dos listas en una lista de pares, no es terminal
assoc a l: Busca la clave a en la lista l, y retorna su valor en una lista de pares cartesianos
```

Tipos de datos

Estructura:

```
type identificador = Tipo1 | Tipo2 | Tipo3 | TipoN;;

(* Con la palabra reservada of, podemos decir a ese tipo, de que subtipo es *)
type identificador = Tipo1 of Subtipo | Tipo2 of Subtipo;;
```

Ejemplo, con este ejemplo podemos ver que es posible crear una tipo de dato carta que almacena el número y su palo en forma de tupla. Es importante saber que los nuevos tipos que creemos, deben comenzar por mayúscula.

```
type palo = Bastos | Espadas | Oros | Copas;;
type carta = Numero of (int * palo);;

Copas;;
- : palo = Copas

(7, Oros);;
- : int * palo = (7, Oros)

Numero (12, Espadas);;
- : carta = Numero (12, Espadas)
```

También es posible crear tipos de datos recursivos, útil para hacer estructuras como por ejemplo, árboles binarios.

```
type 'a tree = (* 'a indica que puede almacenar cualquier tipo de dato *)
    Empty (* Para indicar una hoja *)
    | Node of 'a * 'a tree * 'a tree;; (* Con Node se crearán árboles: Dato, arbol izquierdo, arbol derecho *)
```

Al escribir 'a estamos dejando que el primer elemento que se añada al un árbol hecho con este tipo sea el que decida que tipo de dato almacenará el árbol, porque todos los nodos del árbol deben tener el mismo tipo de dato, o Empty. Si queremos forzar que un tipo de árbol almacene un tipo de dato en específico, podemos hacerlo declarándolo en lo que es el Nodo.

```
type treeC =
    Empty
    | NodeC of char * treeC * treeC * treeC;;
```

En este caso acabamos de crear un tipo treeC, que es un árbol ternario (Cada nodo tiene 3 ramas), que solo permite almacenar caracteres. Usando los tipo de datos de antes, quedaría así:

```
Node(1,
    Node(2,
        Node(4, Empty, Empty),
        Empty),
    Node(3, Empty, Empty));;

NodeC('a',
    NodeC('1', Empty, Empty, Empty),
    NodeC('b',
        NodeC('c', Empty, Empty, Empty),
        Empty,
        NodeC('2', Empty, Empty, Empty)),
    NodeC('3', Empty, NodeC('d', Empty, Empty, Empty), NodeC('f', Empty, Empty, Empty)));;

    1             a
  /   \         / | \
 2     3       1  b  3
  \             /\  |\
   4           c  2 d f
```

Arrays

Estructura:

```
let identificador = [| elemento1; elemento2; elemento3; elementoN |]
```

Nos permite asociar datos e identificarlos a un solo índice, también como pasa con las listas o los tipos de datos, en un array no se pueden guardar tipos de datos distintos en un mismo array:

```
let array = [| 1; 2; 3 ;4 ;5 ;6 ;7 ;8 ;9 |]

array. (0);; (* Es obligatorio poner el punto detrás del nombre del array *)
- : int = 1

array. (8) <- 10;; (* Así asignamos el 10 a la posición 8, que es la del número 9 *)
array;; (* Después de ejecutar la otra línea, quedará así *)
- : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 10|]
```

Programación Orientada a Objetos

La programación orientada a objetos en OCaml funciona de una manera muy similar a la de Java. Hay clases que generan objetos con los atributos y métodos, pero hay una distinciones importantes.

- Existen **dos tipos de objetos**
    - **De clase**: Son objetos normales que se crean a partir de una clase como en Java, tienen su atributos y métodos própios.
    - **Inmediatos**: Son objetos sin clase, también tienen sus propios atributos y métodos. Se pueden crear con valores por defecto o con los valores que queramos a partir de funciones factoría.
- Los atributos están **obligados a tener un valor por defecto**.
- No tienen un método constructor, es **la propia clase la que crea los objetos**.

Estructura de una clase.

```
class <identificador> [<parametros>] =
object(<alias>)
    (* Atributos y métodos *)
end;;
```

- **Identficador**: Será el nombre de la clase.
- **Parámetros**: Son los parámetros que recibe la clase para construir el objetos.
- **Alias**: Es el nombre que vamos a dar para referirnos a la propia clase. Esto en Java serías el `this`, pero en OCaml somos nosotros como programadores los que debemos escoger que palabra debemos escoger nosotros para referirnos a la propia clase.

Siguiendo los apuntes de clase, vamos hacer una clase que genere puntos en un plano de dos dimensiones:

```
class point2D (xInit, yInit) (* coords. del punto *) =
object (self)
    (* Variables, coordenadas XX' e YY' *)
    val mutable x = xInit
    val mutable y = yInit

    (* Metodos*)
    (* getters acceso a coord. *)
    method get_x = x
    method get_y = y

    (* setters asignacion coord. *)
    method set_x x' = x<-x'
    method set_y y' = y<-y'

    (* reasignar coordenadas de forma absoluta o relativa *)
    method moveto (x',y') = x<-x'; y<-y'
    method rmoveto (dx,dy) = self#moveto(x + dx, y + dy)

    (* toString() *)
    method to_string () = "("^(string_of_int x)^", "^(string_of_int y)^")"

end;;

class point2D :
    int * int ->
    object
        val mutable x : int
        val mutable y : int
        method get_x : int
        method get_y : int
        method moveto : int * int -> unit
        method rmoveto : int * int -> unit
     method set_x : int -> unit
    method set_y : int -> unit
    method to_string : unit -> string
  end
```

Y ahora vamos a analizar el código:

- **Variables**: Se declaran con la palabra reservada `val`. **Por defecto** son **inmutables**, esto quiere decir que el valor que tienen por defecto no se puede cambiar, como si fuera un atributo final de Java. **Para** hacer que sí se pueda **modificar cada atributo** debemos añadir en su declaración `mutable`.
- **Métodos**: Se declaran con la palabra reservada `method`. Como en otros lenguajes orientados a objetos, también tenemos getters, setters y métodos normales.
    - **Getters**: No tienen ningún parámetro de entrada porque no hace falta, simplemente retornan un valor como `method get_x = x`.
    - **Setters**: Necesitan un parámetro para actualizar atributos, esta actualización se realiza con el operador de asignación visto en los arrays, `method set_y nuevaY = y<-nuevaY`.
    - **Métodos**: Son métodos normales como en Java, una diferencia es que cuando queremos referirnos a la propia clase, a la hora de utilizar el alias, se utiliza poniendo un **#**. `method`

`moveto (x',y') = x<-x'; y<-y'` y `method rmoveto (dx,dy) = self#moveto(x + dx, y + dy)`.

Herencia

Siguiendo con paradigma de la programación orientada a objetos y el ejemplo de clase, uno base de la OOP, es la herencia de clases, donde podemos crear nuevos objetos que compartan atributos y métodos. Cabe destacar que en OCaml a diferencia de Java, existe la herencia múltiple, permitiéndonos hacer que una clase herede de más de una a la vez.

Para el ejemplo vamos hacer una clase point2Deq, que tendrá a mayores un método equals para comparar dos puntos:

```
class point2Deq (x_init, y_init) =
object (self: 'self)
    inherit point2D (x_init, y_init)

    method equals (o: 'self) = (o#get_x = self#get_x) && (o#get_y = self#get_y)
end;;
```

Ahora mismo, la clase point2Deq hereda todos los métodos hechos antes en point2D, esta herencia se hace con la palabra reservada `inherit`, el tipo de herencia de OCaml es distinto al de Java, es como si todo lo que escribiéramos en una clase, lo copiáramos y pegásemos en las que heredan de ella. Este es el motivo por el cual debemos escribir un segundo self, uno para referimos a la clase actual, y otro para el padre.

Objetos inmediatos

Son objetos sin clase, un ejemplo para un objeto de una coordenada unidimensional, sería:

```
let o1d =
      object
            val mutable x = 0
            method get_x = x
            method rmoveto d' = x <- x + d'
end;;
```

Como podemos ver, este objeto o1d, se crea con un valor por defecto en la variable x, y tal como está hecho, no podemos inicializar el objeto con otro valor en la x. Para hacer esto, existen las **funciones factoría**, que permiten crear objetos sin clase y llamarlos a partir de variables.

```
let factoria_pinmediato1D (xinit:int) =
    object
        val mutable x = xinit
        method get_x = x
        method rmoveto d' = x <- x + d'
end;;
- val factoria_pinmediato1D : int -> < get_x : int; rmoveto : int -> unit > = <fun>

let i5 = factoria_pinmediato1D 5
- val i5 : < get_x : int; rmoveto : int -> unit > = <obj>
```

Agregación

Se trata de crear un nuevo objetos que se crea a partir de otros objetos. Siguiendo con las coordenadas, podemos hacer una clase arista que sea el resultado de unir dos puntos del plano.

```
class edge2D (a: point2D) (b: point2D) = (* Poniendo los :, podemos obligar que a y b sean objetos poi*)
      object
        val vertexes = (a,b)
        method get_vertexes = vertexes
end;;

class edge2D :
    point2D ->
    point2D ->
    object
        val vertexes : point2D * point2D
        method get_vertexes : point2D * point2D
end
```

Prácticas

Factorial

```
let rec fact = function (* Calula el factorial de un número de forma recursiva *)
    0 -> 1
    | n -> n * fact (n - 1)

let numArgumentos = Array.length Sys.argv (* Contamos cuantos argumentos hay *)

let main = (* Si no son dos, retorna un error *)
    if numArgumentos <> 2 then print_endline "Número de parámetros incorrecto"
    else print_endline(string_of_int( fact( int_of_string( Array.get Sys.argv(1) ) ) ) )
    (* Para imprimir pasamos el string a entero, calculamos el factorial de ese número,
        y lo pasamos a string de nuevo para imprimirlo por pantalla *)

(* Otra opción *)
let rec fact = function (* Calcula el factorial *)
    0 -> 1
  | n -> n * fact (n - 1);;

try (* Se intenta ejecutar este código que puede dar error *)
    print_endline (string_of_int (fact (int_of_string Sys.argv.(1))))
with (* Si no puede busca el error que es y ejecuta el siguiente código *)
  | Stack_overflow
  | Invalid_argument _
  | Failure _ -> print_endline "argumento invalido"
```

Fibonacci

```
let rec fib n =
    if n <= 1 then n
    else fib (n-1) + fib (n-2)

let rec calculoSiguienteNumero n =
    if n = 0
    then "0"
    else calculoSiguienteNumero(n-1) ^ "\n" ^ string_of_int(fib(n));;

let rec mensaje =
    if (Array.length Sys.argv) = 2
    then (calculoSiguienteNumero (int_of_string(Sys.argv.(1))))
    else ("Número de argumentos incorrecto") in
    print_endline mensaje;;
```

Ejercicio 41, suma de cifras, numero de cifras, exp10, reverso de un número y palíndromos

```
let rec sum_cifras n = (* sum_cifras 1234 -> 10 *)
    if n = 0
    then 0
    else n mod 10 + sum_cifras (n / 10)

let rec num_cifras n = (* num_cifras 1234 -> 4 *)
    if n = 0
    then 0
    else 1 + num_cifras (n / 10)

let rec exp10 n = (* exp10 3 -> 1000 *)
    if n = 0
    then 1
    else 10 * exp10 (n - 1)

let rec reverse n = (* reverse 1234 -> 4321 *)
    if n = 0
    then 0
    else (n mod 10) * exp10 (num_cifras n - 1) + reverse (n / 10)

let rec palindromo s = (* palindromo vaca -> false ;; palindromo abba -> true *)
    let rec counter i =
        if i >= (String.length s - i)
        then true
        else if s.[i] <> s.[String.length s - i - 1]
             then false
             else counter (i + 1)
    in counter 0

val sum_cifras : int -> int
val num_cifras : int -> int
val exp10 : int -> int
val reverse : int -> int
val palindromo : string -> bool
```

Potencias

```
let rec power x y =
    if y = 0 then 1
    else x * power x (y - 1)

let rec power' x y =
    if y = 0 then 1
    else if (y mod 2 = 0) then power' (x * x) (y / 2)
    else x * power' (x * x) (y / 2)
(* La función power' es más eficiente que la primera, porque por cada
    iteriación que da el valor del exponente y se reduce a la mitad
    y no solo en 1 como en power *)

let rec powerf x y =
    if y = 0 then 1.
    else if (y mod 2 = 0) then powerf (x*.x) (y / 2)
    else x *. powerf (x *. x) (y / 2)
(* Solo hay que poner que las x sean valores flotantes *)

val power : int -> int -> int
val power' : int -> int -> int
val powerf : float -> int -> float
```

Ejercicio 62, curry, uncurry

```
(* curry : (('a * 'b) -> 'c) -> ('a -> ('b -> 'c)) *)
let curry =  function c -> function a -> function b -> c (a,b)
let curry c a b = c (a,b)

(* uncurry : (('a -> ('b -> 'c) -> ('a * 'b) -> 'c)) *)
let uncurry = function c -> function (a,b) -> c a b
let uncurry c(a,b) = c a b

(* ----- *)

(* uncurry (+); *)
(* Devolverá lo que hace uncurry *)

let sum = (uncurry (+))
(* Se almacena en sum, lo que hace uncurry *)

(* sum 1;; *)
(* Retornará un error porque necesita dos parámetros *)

(* sum (2,1); *)
(* Devolverá 3, porque en la línea de let sum, indicamos que queremos
   hacer una suma *)

let g = curry (function p -> 2 * fst p + 3 * snd p)
(* Almacena en g, lo que queremos hacer con curry *)
(* fst retorna el primer elemento de una pareja *)
(* snd retornar el segundo elemento de una pareja *)

(* g (2,5); *)
(* Debería dar un error por no recibir el tipo de dato correcto *)

```

```
let h = g 2
(* En h, metemos la operación de g 2 *)

(* h 1, h 2, h 3;; *)
(*  h 1 = g 2 1 = 2 * 2 + 3 * 1 = 7
    h 2 = g 2 2 = 2 * 2 + 3 * 2 = 10
    h 3 = g 2 3 = 2 * 2 + 3 * 3 = 13  *)

(*  h 1, h 2, h 3 = (7, 10, 13)  *)

(* ----- *)

(* comp : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b) *)
let comp = function f -> function g -> function c -> f (g c)

let f = let square x = x * x in comp square ((+) 1)

let i = function a -> a;;
let j = function (a, b) -> a;;
let k = function (a, b) -> b;;
let l = function a -> [a];;

val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
val sum : int * int -> int
val g : int -> int -> int
val h : int -> int

val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
val f : int -> int

val i : 'a -> 'a
val j : 'a * 'b -> 'a
val k : 'a * 'b -> 'b
val l : 'a -> 'a list
```

Potencia modular

```
let rec powmod m b e =
    let restoMB = b mod m
    in if e > 0
    then if (e mod 2) = 0
        then powmod m (restoMB * restoMB) (e / 2) mod m
        else restoMB * powmod m (restoMB * restoMB) ((e - 1) / 2) mod m
    else 1

val powmod : int -> int -> int -> int
```

Tipos de datos de las funciones del módulo List:

```
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val length : 'a list -> int
val compare_lengths : 'a list -> 'b list -> int
```

```
val nth : 'a list -> int -> 'a
val append : 'a list -> 'a list -> 'a list
val init : int -> (int -> 'a) -> 'a list
val rev : 'a list -> 'a list
val rev_append : 'a list -> 'a list -> 'a list
val concat : 'a list list -> 'a list
val flatten : 'a list list -> 'a list
val map : ('a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val find : ('a -> bool) -> 'a list -> 'a
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val mem : 'a -> 'a list -> bool
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
```

Ejercicio 91, to0from, fromto, incseg, remove, compress

```
let to0from n = (* to0from 4 -> [4; 3; 2; 1; 0] *)
    List.rev (List.init (n + 1) (function x -> x))

let fromto m n = (* fromto 2 5 -> [2; 3; 4; 5] *)
    let rec aux l i =
        if i < m
        then l
        else aux (i::l) (i - 1)
    in aux [] n

let incseg l = (* incseg [1; 2; 3; 4] -> [1; 3; 6; 10] *)
    let rec aux l acc l2 = match l with
         [] -> []
        | [head] -> List.rev ((head + acc)::l2)
        | head::tail -> aux tail (head + acc) ((head + acc)::l2)
     in aux l 0 []

let remove x l = (* remove 2 [1; 2; 3; 2; 4] -> [1; 3; 2; 4] *)
    let rec aux acc = function
        [] -> l |
        head::tail -> if x = head
                      then List.rev_append acc tail
                       else aux (head::acc) tail
    in aux [] l

let compress l = (* compress [1; 1; 2; 2; 2; 3; 3; 3; 2] -> [1; 2; 3; 2]*)
  let rec aux acc l = match l with
        | head1::head2::tail -> if head1=head2
                                then aux acc (head2::tail)
                                  else aux (head1::acc) (head2::tail)
        | [head] -> aux (head::acc) []
```

```
        | []-> List.rev acc
  in (aux [] l);;

val toOfrom : int -> int list
val fromto : int -> int -> int list
val incseg : int list -> int list
val remove : 'a -> 'a list -> 'a list
val compress : 'a list -> 'a list
```

Merge sort y qsort

```
let merge' ord (l1, l2) =
    let rec aux (a1, a2) mer = match a1, a2 with
        [], l | l, [] -> List.rev_append mer l
        | head1::tail1, head2::tail2 -> if ord head1 head2
                                        then aux (tail1, head2::tail2) (head1::mer)
                                        else aux (head1::tail1, tail2) (head2::mer)
    in aux (l1, l2) [];;

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)

val merge' : ('a -> 'a -> bool) -> 'a list * 'a list -> 'a list
val qsort2 : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Árboles binarios

```
type 'a bin_tree =
    Empty
    | Node of 'a * 'a bin_tree * 'a bin_tree;;

let map_tree f tree =
    let rec aux = function
        | Empty -> Empty
        | Node (x, l, r) -> Node (f x, aux l, aux r)
    in aux tree

let rec fold_tree f a = function
    Empty -> a
    | Node (x, l, r) -> f x (fold_tree f a l) (fold_tree f a r)

let rec sum t =
    fold_tree(fun a b c-> a + b + c) 0 t

let rec prod t =
    fold_tree(fun a b c-> a *. b*. c) 1. t

let rec size t =
    fold_tree(fun a b c-> 1 + b + c) 0 t

```

```
let height tree =
    let rec aux = function
        | Empty -> 0
        | Node (_, l, r) -> 1 + max (aux l) (aux r)
    in aux tree

let rec inorder t =
    fold_tree(fun a b c -> b @ [a] @ c) [] t

let rec mirror t =
    fold_tree(fun a b c -> Node(a, c, b)) Empty t
(*-----------------------*)
type 'a bin_tree =
    Empty
  | Node of 'a * 'a bin_tree * 'a bin_tree

val map_tree : ('a -> 'b) -> 'a bin_tree -> 'b bin_tree
(* devuelve el bin_tree resultante de aplicar una función a cada unos de sus nodos *)

val fold_tree : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a bin_tree -> 'b
(* generaliza operaciones de reducción sobre valores de tipo bin_tree *)

val sum : int bin_tree -> int
(* devuelve la suma de los nodos de un int bin_tree *)

val prod : float bin_tree -> float
(* devuelve el producto de los nodos de un float bin_tree *)

val size : 'a bin_tree -> int
(* devuelve el número de nodos de un bin_tree *)

val height : 'a bin_tree -> int
(* devuelve la altura de un bin_tree *)

val inorder : 'a bin_tree -> 'a list
(* devuelve la lista de nodos de un bin_tree en "orden" *)

val mirror : 'a bin_tree -> 'a bin_tree
(* devuelve la imagen especular de un bin_tree *)
```

GTree, un árbol que tiene como hijo una lista de arboles GTree

```
type 'a g_tree =
  Gt of 'a * 'a g_tree list;;

let rec size = function
    Gt (_, []) -> 1
    | Gt (r, h::t) -> size h + size (Gt (r, t))

let size tree =
  let rec aux = function
    | Gt (_, hijo) ->
        1 + List.fold_left (fun acc t -> acc + aux t) 0 hijo
  in aux tree

```

```
let height tree =
  let rec aux = function
    | Gt (_, hijo) ->
        if hijo = []
        then 1
        else 1 + List.fold_left max 0 (List.map aux hijo)

  in aux tree

let leaves tree =
    let rec aux acc = function
        | Gt (x, hijo) ->
            if hijo = []
            then x :: acc
            else List.fold_left aux acc hijo

    in aux [] tree

let mirror tree =
    let rec aux = function
        | Gt (x, hijo) ->
            Gt (x, List.rev (List.map aux hijo))

    in aux tree

let preorden tree =
    let rec aux acc = function
        | Gt (x, hijo) ->
            let acc' = x :: acc
            in List.fold_left aux acc' hijo

    in aux [] tree

let preorder tree = List.rev (preorden tree)

let postorden tree =
    let rec aux acc = function
        | Gt (x, hijo) ->
            let acc' = List.fold_left aux acc hijo
            in x :: acc'

    in aux [] tree

let postorder tree = List.rev (postorden tree)
(*------------------------*)
type 'a g_tree = Gt of 'a * 'a g_tree list

val size : 'a g_tree -> int
(* devuelve el número de nodos de un g_tree *)

val height : 'a g_tree -> int
(* devuelve la "altura", como número de niveles, de un g_tree *)

val leaves : 'a g_tree -> 'a list
(* devuelve las hojas de un g_tree, "de izquierda a derecha" *)

```

```
val mirror : 'a g_tree -> 'a g_tree
(* devuelve la imagen especular de un g_tree *)

val preorder : 'a g_tree -> 'a list
(* devuelve la lista de nodos de un g_tree en "preorden" *)

val postorder : 'a g_tree -> 'a list
(* devuelve la lista de nodos de un g_tree en "postorden" *)
```

Recorrido en anchura de árboles GTree, breadth_first

```
open G_tree;;

let rec breadth_first = function
    Gt (x, []) -> [x]
    | Gt (x, (Gt (y, t2))::t1) -> x :: breadth_first (Gt (y, t1@t2))

let breadth_first_t arbol =
    let rec aux acc = function
        Gt (x, []) -> List.rev (x::acc)
        | Gt (x, Gt(raiz, ramas)::lista) ->
            aux (x::acc) (Gt(raiz, List.rev_append (List.rev lista) ramas))
    in aux [] arbol

let leaf v = Gt(v,[])

let id x = x

let init_tree n = Gt(n, List.rev_map leaf (List.init n id))
(*---------------------*)
val breadth_first : 'a G_tree.g_tree -> 'a list
val breadth_first_t : 'a G_tree.g_tree -> 'a list
val t2 : int G_tree.g_tree
```

Árbol binario de búsqueda

```
open Bin_tree;;

let insert_tree ord x t =
    let rec insert = function
        | Empty -> Node (x, Empty, Empty)
        | Node (y, left, right) ->
            if ord x y
            then Node (y, insert left, right)
            else Node (y, left, insert right)
    in insert t;;

let tsort ord l =
    inorder (List.fold_left (fun a x -> insert_tree ord x a) Empty l)
(*-----------------------*)
val insert_tree :  ('a -> 'a -> bool) -> 'a -> 'a Bin_tree.bin_tree -> 'a
Bin_tree.bin_tree
val tsort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Otro tipo de ejercicios

```
let f = List.fold_left (fun x y -> 2 * x + y ) 0;;
(*-----------------results----------------------*)
val f: int list -> int = <fun>

f [1; 0; 1; 1], f [1; 1; 1; 1];;
(*-----------------results----------------------*)
- : int * int = (11, 15)

let rec base b n =
    let q = n / b
    in if q = 0
        then [n]
        else n mod b::base b q;;
(*-----------------results----------------------*)
val base: int -> int -> int list = <fun>

[base 2 1; base 2 16; base 10 2021];
(*-----------------results----------------------*)
- : int list list = [[1]; [0; 0; 0; 0; 1]; [1; 2; 0; 2]]

let x, y = 2 + 1, 0;;
(*-----------------results----------------------*)
val x: int = 3
val y: int = 0

(function x -> function y -> 2 * y ) y x;;
(*-----------------results----------------------*)
- : int = 6

let f = fun y -> (+) x y;;
(*-----------------results----------------------*)
val f: int -> int = <fun>

let g f x = f (f x ) in g f 5;;
(*-----------------results----------------------*)
- : int = 11

let h = fun x y -> y:: x;;
(*-----------------results----------------------*)
val h: 'a list -> 'a -> 'a list = <fun>

h ['h'];;
(*-----------------results----------------------*)
- : char -> char list = <fun>

h [] [0];;
(*-----------------results----------------------*)
- : int list list = [[0]]

let x, y = y, x;;
(*-----------------results----------------------*)
val x : int = 0
val y : int = 3
```

```
54
55  let v = ref x;;
56  (*-----------------results----------------------*)
57  val v : int ref = {contents = 0 }
58
59  v + 1;;
60  (*-----------------results----------------------*)
61  Line 1 , characters 0 - 1 :
62  1 | v + 1;;;;
63      ^
64  Error: This expression has type int ref
65  but an expression was expected of type int
66
67  let w = v;;
68  (*-----------------results----------------------*)
69  val w : int ref = {contents = 0 }
70
71  w:=! w + 1;! v,! w;;
72  (*-----------------results----------------------*)
73  - : int * int = (1, 1)
74
75  let rec trivide = function
76      [] -> [], [], []
77      | h::t ->
78          let t1, t2, t3 = trivide t in h::t3,t1,t2;;
79  (*-----------------results----------------------*)
80  val trivide: 'a list -> 'a list * 'a list * 'a list = <fun>
81
82  let f3 f x = (x, f x, f(f x));;
83  (*-----------------results----------------------*)
84  val f3 : ('a -> 'a) -> 'a -> 'a * 'a * 'a = <fun>
85
86  let x, y, z = let g x = x * x in f3 g 2;;
87  (*-----------------results----------------------*)
88  val x : int = 2
89  val y : int = 4
90  val z : int = 16
91
92  (function _ :: _ :: t-> t) [1;2;3];;
93  (*-----------------results----------------------*)
94  Warning 8: this pattern-matching is not exhaustive.
95  Here is an example of a case that is not matched:
96  (_::[]|[])
97  - : int list = [3]
98
99  List.map (function x -> 2 * x + 1);;
100 (*-----------------results----------------------*)
101 - : int list -> int list = <fun>
102
103 let rec f = function [] -> 0 | h::[] -> h
104     | h1::h2::t -> h1 + h2 - f t;;
105 (*-----------------results----------------------*)
106 val f : int list -> int = <fun>
107
108 f [1000; 100; 10], f [1000; 100; 10; 1];;
109 (*-----------------results----------------------*)
```

```
- : int * int = (1090, 1089)

List.fold_right (-) [4; 3; 2] 1;;
(*-----------------results----------------------*)
- : int = 2

let rec comb f = function
    h1::h2::t -> f h1 h2 :: comb f t
    | l -> l;;
(*-----------------results----------------------*)
val comb : ('a -> 'a -> 'a) -> 'a list -> 'a list = <fun>

comb (+);;
(*-----------------results----------------------*)
- : int list -> int list = <fun>

comb (+) [1; 2; 3; 4; 5];;
(*-----------------results----------------------*)
- : int list = [3; 7; 5]

let combT f l =
    let rec loop f lAux = function
        [] -> List.rev lAux
        | h::[] -> List.rev (h::lAux)
        | h1::h2::t -> loop f (f h1 h2::lAux) t
    in loop f [] l;;
(*-----------------results----------------------*)

type 'a tree = T of 'a * 'a tree list;;
let s x = T (x, []);;
(*-----------------results----------------------*)
val s : 'a -> 'a tree = <fun>

let t = T (1, [s 2; s 3; s 4]);;
(*-----------------results----------------------*)
val t : int tree = T (1, [T (2, []); T (3, []); T (4, [])])

let rec sum = function
    T (x, []) -> x
    | T (r, T (rl , l)::t) -> r + sum(T (rl, l @ t));;
(*-----------------results----------------------*)
val sum : int tree -> int = <fun>

sum t;;
(*-----------------results----------------------*)
- : int = 10

let sumT t =
    let rec loop aux = function
        T (x, []) -> aux + x
        | T (x, T (x1, l)::t) -> loop (aux + x) (T (x1, List.rev_append l t))
    in loop 0 t;;
```