

Paradigmas De Programación

José María Molinelli

Desp 4.0.9.

Viernes → 9:30 - 10:30

11:30 - 12:30

16:30 - 17:30

Lunes → 10:30 - 11:30

12:30 - 13:30

17:30 - 18:30

www.dc.fi.udc.es/~paradigmas

Cmml.inria.fr

Developing applications with Objective Cmml.

Todo el material y manuales en la página web de la asignatura, escrito anteriormente.

Ocaml es un lenguaje declarativo fu

El compilador de Ocaml → ocamle. test
La terminación ".ml" es equivalente a
El "test" es el nombre de "a.out".

El comando (línea de código) que he
anterioridad, genera código para una máqui
pensada para un programa ocamle.

En cambio, "ocamlopt" genera código
directamente sobre la máquina física. Si
ese procesador, para esa plataforma concret

La máquina virtual se llama "ocamlr

Código máquina virtual → bytecode

Código máquina física → native cod

Existe un tercer compilador, el interactiu

Cuidado con la compilación y la ejecución
aprender a diferenciarlas.

Para terminar las frases en el compilador
se pone ";". No forman parte del

2 + 3 ;;] La # es del compilador interactivo.
- : int = 5] no forma parte del lenguaje. El "- :" indica el tipo de una expresión.

'\064';;

↳ Devuelve el char correspondiente en la tabla ASCII.

Para esto también existen funciones como:

int-of-char 'z';;

char-of-int 121;;

De char a int, son funciones tipo <fun>.

Expresiones.

Definiciones: let.

Comienza índice de valores en el manual (Index of values).

El módulo "Pervasives" es el modo principal, el básico.

Excepción → error de ejecución.

Comentarios, entre (* *)

PA.ml

Un valor float es un real.

2 + 5 * 3;;

(* -: int = 17*)

1.0;;

(* -: float = 1*)

ELENA
DELAMANO
FREIJE

Los nombres de los valores siempre empiezan en minúscula.

Char. code ;;

↳ minúscula porque es el valor.
↳ Mayúscula porque es el módulo.

Los del módulo "Pervasives" se pueden poner directamente los valores, mientras que en el resto, tenemos la necesidad de poner delante del nombre del valor el nombre del módulo.

();;

(* -! unit = ()*) es el tipo de dato unit más sencillo. Sólo tiene ese valor de tipo unit. Similar al "void" en C.

Las funciones son valores, y tienen su propio tipo de

dates

Unbound → que no está ligado a nada.

Distinción entre Mayúsculas y minúsculas.

code = int_of_char.

(2,5);;

(* -! int x int = (2,5)*) \Rightarrow Escrito sin paréntesis es
↓
Producto cartesiano. igual. "2 x 'A'" \Rightarrow int x char.

Tipos → Unit, functions, int, floats, doubles, productos cartesianos, definición (dar nombre a un valor) ...

<ids> → Nombre que se le pone a un valor.

Let <ids> := <exp>

<exp> → cualquier 'exp' válida en Ocaml.

Para salir del compilador interactivo hay varias posibilidades, pero no pertenecen al lenguaje:

ctrl + Z ;;

ctrl + D ;; si no hay entrada, se acaba, pero lo mejor es utilizar el comando:

#quit ;;

Si ponemos "edit ocaml" o "led ocaml", abriremos el editor de ocaml, que funciona exactamente igual que el compilador interactivo, sólo que tenemos la posibilidad de acceder a comandos o frases que hemos puesto con anterioridad pulsando ↑, como en el terminal en si.

Sentencia IF - THEN - ELSE.

(if then <e1>, else <e2>) → tipo α.

 ⇒ expresión correcta en boolean.

<e1> ⇒ expresión correcta tipo α. } Tienen que ser

<e2> ⇒ expresión correcta tipo α. } ambas del mismo tipo.

Function <id> → <e> // Las funciones son valores.

(Function x → 2*x) (2+1) // Esta es de tipo int.

Function de α en β

Argumento válido para la función.
Tiene que ser de tipo α.

evaluación eager and lazy // rluwrap.

Se comparan del mismo tipo. $n > 0$. Tiene que ser int, no puede ser float (0.0);

La palabra "let" sirve para definir.

La definición de funciones se puede abreviar:

let $\langle f \rangle = \text{function} \langle x \rangle \rightarrow \langle e \rangle$

let $\langle f \rangle \langle x \rangle = \langle e \rangle$

Por ejemplo, para la función de abs sería:

let abs = function $n \rightarrow$

if $n > 0$ then n else $-n$;

let abs $n = \text{if } n > 0 \text{ then } n \text{ else } -n$;

Pero también podemos definir funciones recursivas.

Sólo es necesario introducir un "rec" antes del nombre de la función, de la siguiente manera:

let rec fact = function $n \rightarrow$ if $n = 0$ then 1
else $n * \text{fact}(n-1)$;

En los int's negativos, cuando alcanza el tope, se pasa al tope positivo.

Ejercicio propuesto para TGR's.

Se deben construir los valores del módulo char:

- Uppercase. } Definir Upper y lower;
- Lowercase. } funciones que hacen lo mismo.

No se pueden usar esos valores propios.

TGR's (profesores y grupos)

J. M. Molinelli

- 3.2 (Ma 17:30)
- 3.3 (Mi 17:30)
- 3.1 (Ju 17:30)

Jorge Grana

- 4.2 (Ma 15:30)
- 1.3 (Mi 10:30)
- 2.1 (Ju 8:30)
- 1.1 (Ju 10:30)
- 4.1 (Ju 15:30)

jorge.grana @ udc.es.

$\langle b_1 \rangle \parallel \langle b_2 \rangle : \text{bool} \Rightarrow \text{if } \langle b_1 \rangle \text{ then TRUE else } \langle b_2 \rangle$.

El \parallel \Rightarrow OR lógico.

$b_1 \vee b_2 \Rightarrow$ tipo bool

$\langle b_1 \rangle \& \langle b_2 \rangle : \text{bool} \Rightarrow \text{if } \langle b_1 \rangle \text{ then } \langle b_2 \rangle \text{ else FALSE}$.

El $\&$ \Rightarrow AND lógico

$b_1 \wedge b_2 \Rightarrow$ tipo bool.

Lazy \Rightarrow NO evalúa la segunda ; con atajo ; cortocircuito.

`# [1; 2; 3];;` } El mismo tipo.
`# [1; 3; 5; 7];;` } Son secuencias, listas de enteros.
`-: Unit List = [1;2;3].`

Para dos listas ser iguales, tienen que ser del mismo tipo, tener los mismos números y en el mismo orden.

Las hay de diferentes datos, no sólo de int.
 Si existe el dato "x", existirá "x list". Son listas homogéneas: todos los elementos son del mismo tipo.

Se pueden solicitar dos cosas de una lista:

- La cabecera de la lista (head).
- La cola, que es todo menos la cabecera (tail).

En una 'int list', la cabecera es un int y la cola es una 'int list'.

`List.hd lista;; → head.`
`List.tl lista;; → tail.`
} La lista es un valor, no se puede añadir, porque sería otra lista. Puede concatenarse, etc.

La concatenación de dos listas:

`[1; 2; 3] @ [5; 7];;`
`↳ [1; 2; 3; 5; 7]`

Existe en Ocaml una función predefinida para saber el número de elementos de una lista:

`List.length lista;;`

Para referirse a "tal cabecera y tal cola".

$\langle h \rangle :! d$ } $\langle h \rangle :: \langle t \rangle$ } la lista que tiene
 $\langle t \rangle : a \text{ list}$ } como cabecera $\langle h \rangle$ y como cola $\langle t \rangle$.

$\lambda :: [3; 5];;$
- : int.list = $[1; 3; 5]$.

$l_2 = [5; 9]$

$l_3 = \text{list.tl } l_2 = [9]$.

$l_4 \Rightarrow \text{list.te } l_3 = \text{listaVacia}$
↳ int.list = []

list.hd $l_4 \Rightarrow$ No se debe hacer. No hay. Error!

Funciones.

primeros : int \longrightarrow int.list
 $\emptyset \longrightarrow$ Lista.Vacia []
1 \longrightarrow [1]
2 \longrightarrow [1; 2]
3 \longrightarrow [1; 2; 3].

:

n \longrightarrow [1; 2; ...; n]

ELENA
DELAMANO
FREIRE

```
Let rec primeros = function n →  
    if n > 0 then primeros (n-1) @ [n]  
    else [];;
```

↑
No sirve. Para
el \emptyset no funciona,
hay que poner el if.

[] lista vacía para todos tipos.

Las funciones "Uppercase" y "Lowercase"
están resueltas en el apartado de seminarios.
Son dos funciones de $\text{char} \rightarrow \text{char}$ <fun>.

[];;

- :  list.

; Es un tipo polimorfo, es decir, que
puede funcionar con cualquiera de los
tipos.

Function x → x;;

Esta es la función identidad. Es válido para
cualquier tipo: polimorfa.

λ x. x
λ x. x
x x

```
# Function (x: int) → x ;;
```

También es la función identidad, pero
esta sólo es válida para los int.

```
# fst → También es polimórfica.
```

```
# Let v = function f → f Ø ii } int → 'a.  
                                f aplica a Ø }
```

```
val v = (int → 'a) → 'a  
        f           f Ø
```

v se aplica a una función, como "abs".

Funciones de (int → 'a), como "char_of_int".

; int_of_char no sería, porque (char → int).

```
# Let s = function x → (función y → x + y) ii
```

```
val s : int → (int → int)
```

```
s (+) ^ = s
```

Podemos representar / declarar esa función de forma
más abreviada:

```
# let s x = function (y) → x + y ii
```

o

```
# let s x y = x + y ii
```

No existen funciones de varios argumentos, sólo de uno.

```
# let suma = function p → fst (p) + snd (p);;
```

↑
tipo (int × int) → int.

Hay dos formas:

- * Forma clásica: función que devuelve función.
- * Forma Haskell Curry, donde se tiene que, por ejemplo, s: int → int → int.

```
# (+) 3 4;;
```

(+) : función a la curry. Todos los operadores binarios igual.

(^) : concatenación de strings.

```
# let succ = (+) 1;; → Función sucesor.
```

```
f @) : 'a list → 'a list → 'a list
```

Otra posibilidad de implementar la suma es:

```
# let suma (a,b) = a + b;;
```

o:

```
# let prime (x,y) = x;;
```

- Patrones: estructuras que tienen forma de algún tipo de valor y que se pueden usar nombres para representar parte de ese valor.

$(\text{function } a, b \rightarrow a + b)(2+1, 5-1)$
 $\quad \quad \quad (3, 4)$



↳ Que valores tienen (a, b) para que el patrón sea $(3, 4)$.

pattern → watching ! comparación de patrones.

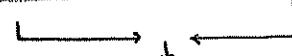
fst (x, y)

En vez de poner un nombre, ponemos un comodín, que se representa con " $-$ ". Así no nos confundimos con un nombre que no se va a usar.

let fst $(x, -) = x$.

Una sola def asigna a varios valores.

let <pattern> = <exp>



Tienen que concordar en tipo con la forma del patrón.

```

# let p = 1,2;;
# let x,y = p;;      x = 1 ..
y = 2.

# let _,z = p;;      _____ " _ " no es un nombre, no
                     queda asociado con nada.
z = 2.

# let x,y = p,x;;   x = (1,2)
y = 1.

```

$\left. \begin{array}{l} 3 :: 2 :: [] \\ 3 :: (2 :: []) \end{array} \right\} \quad :: \rightarrow \text{constructor de listas.}$

```
# let head (h::t) = h;;
```

Warning: el pattern matching. Cuando se
vaya a aplicar, puede coger valores que no
coinciden, como por ejemplo: "[]".

Manera que arreglarlo como:

```
# let hd (h::_) = h;;
# let tl (_::t) = t;;
```

```
# let rec length l =
  if l = [] then Ø
  else 1 + length [l-1];
        $\underbrace{[l-1]}_{(\text{list}, \text{tl } l)}$ .
```

```
# let rec length (-:: t) = 1 + length (t);
```

Cuando no venga con un patrón, se hace con varios.

```
# let rec length = function
  (-:: t)  $\rightarrow$  1 + length (t) } N° de reglas.
  []  $\rightarrow$  Ø ;;
  para separar patrones
```

Si se le da la vuelta a los patrones, el resultado siempre será Ø. El compilador nos dice que el segundo patrón no se va a usar nunca.

```
function <p1>  $\rightarrow$  <ex1>
| <p2>  $\rightarrow$  <ex2>
| <p3>  $\rightarrow$  <ex3>
|
| ...
| <pn>  $\rightarrow$  <exn>;
```

ELENA
DELAMAND
FREIJE

```
# Let tl = function  
  -:: t → t  
  | [] → [];;
```

```
# Let hd = function  
  h :: _ → h  
  | [] → raise (Failure "hd");;
```

} Muestra
error si se
le pasa [].

Failure "tl"
Failure "hd"
Division_by_zero

} Son valores de tipo exn:
error de ejecución.

```
# Let tl = function  
  -:: t → t  
  | [] → raise (Failure "tl");;
```

raise se aplica a un valor de exn → 'a

```
# Let falseltl l = try List.tl l with  
  _ → [];;  
  o Failure "tl" → [];;  
    └─┘ más seguro.
```

} Se ve con cuál
de los casos encaja
el error, y devuelve
el valor de la derecha.

```

try <e> with
  <pe1> → <ex1>
  | <pe2> → <ex2>
  |
  | ...
  | ...
  | <pen> → <exn>

```

donde $<e> \rightarrow \text{exp que puede provocar un error.}$

Factorial.

```

# let rec fact n =
  if n=0 then 1
  else n * fact (n-1);;

```

} No tiene sentido
para los números negativos.
Lo arreglamos con "raise".

Solamente hay que añadirle al código:

```
if (n<0) then raise (Invalid-argument "fact").
```

Pero, de esta manera, hemos perdido eficiencia.

En cada iteración calculará si ($n < 0$). Basta con comprobarlo una sola vez fuera de la recursividad.

```

# let fact n =
  if (n<0) then raise (Invalid-argument "fact")
  else let rec function
    if (n=0) then 1
    else n * fact (n-1);;

```

Otra opción es dejar la función inicial y poner:

```
# let fact n
  if (n >= 0) then fact n
  else raise (Invalid_argument "fact");;
```

Como no hemos implementado la función de forma recursiva, la llamada a "fact" utilizará la función que teníamos antes. (la inicial).

También podemos hacerlo con pattern-matching:

```
# let rec fact = function
  | 0 → 1
  | n → n * fact (n - 1);;
```

$$1 \cdot 2 \cdot 3 \cdots \cdots \cdots \cdots (n-2)(n-1)n$$

$$\frac{(n+1)n}{2}$$

```
# let rec sum-to = function
  | 0 → 0
  | n → n + sum-to (n - 1);;
```

Nos indica la suma de los n primeros números.

```

# let rec last = function
  h :: [] → h
  | _ :: t → last t
  | raise (Failure "last") ;;

```

Este algoritmo devuelve el último d de la lista.

```

last [3;2;1]
last [2;1]
last [1]
1

```

A diferencia de la función factorial (desarrollo abajo), esta función no usa espacio del stack, porque no hay cuentas pendientes. Esto se llama recursividad terminal o final. (tail-recursividad).

```

fact 3
3 · fact 2
3 · (2 · fact 1)
3 · (2 · (1 · fact 0))
3 · (2 · (1))
3 · (2)
6

```

va aumentando el espacio ocupado en la pila.

La función `Last` podría definirse también:

```
#let rec last l =  
  if l = [] then raise (Failure "last")  
  else if List.tl l = [] then List.hd l  
  else last (List.tl l);;
```

Recursiudad final.

Para que quede una recursividad terminal, tenemos que intentar que no queden cuentas pendientes. En el factorial, tenemos que ir realizando la multiplicación.

```
# let fact n =  
  let rec aux (f,i) =  
    if i = 0 then f  
    else aux (f * i, i-1)  
  in aux (1, n);;
```

```
f  
fact 3 (fact 2) | Ahora haremos otra forma  
6 (fact 1) | de calcularlo con rec-terminal.
```

```
# let fact n =  
  let rec aux (f,i) =  
    if i > n then f  
    else aux (f * i, i+1)  
  in aux (1, 1);;
```

```
# let sum-to n =
  let rec aux (f, i) =
    if i > n then f
    else aux (f+i, i+1)
  in aux (0, n);;
```

ELENA
DELAMANO
FREIJE

```
# let rec primeros n =
  if n <= 0 then []
  else primeros (n-1) @ [n]
```

let l primeros (100 000) \Rightarrow son una gran cantidad de pasos. No tiene problema de float, pero sí de tiempo, ya que tarda bastante.

Vamos a cambiar la definición:

```
# let rec from-to m n =
  if m > n then []
  else m :: from-to (m+1) n;;
```

Ahora, primeros está mejor implementado con el from-to. Es mucho más rápido, pero no es una recursividad terminal.

No se puede hacer una comprobación de list para 300 000, por lo que tenemos que conseguir una versión terminal de from-to.

Ejemplos de recursividad terminal:

MÓDULO LIST

```
let rec sumlist = function
  [] → φ
  | h::t → h + sumlist (t);;
;;
;;
```

} Como la aplicación de
sumlist no es la última,
no es recursividad terminal.

El funcionamiento es el siguiente:

```
sumlist [2;3;7]
2 + sumlist [3;7]
2 + (3 + sumlist [7])
2 + (3 + (7 + sumlist []))
2 + (3 + (7 + 0))
2 + (3 + 7)
2 + 10
12
```

La versión terminal es:

```
let sumlist l =
  let rec aux s = function
    [] → s
    | h::t → aux (s+h) t
  in aux φ l;;
```

```

let rec rev = function
  [] → []
  | h :: t → rev t @ [h];

```

En vez de poner $@$ se podría haber puesto el operando del módulo List: List.append.

No es una función recursiva terminal.

Su función es "darte la vuelta" a la lista.

Reverse $[1; 2; 3] \Rightarrow [3; 2; 1]$.

La definición de una función recursiva terminal: en ella, que no se puede usar dentro del cuerpo llamadas a funciones no terminales, por lo que, el siguiente código seguiría sin ser terminal.

```

let rev l =
  let rec aux li = function
    [] → li
    | h :: t → aux (li @ [h]) t
  in aux []

```

Dónde $@$ es una llamada a función no term.

Por lo tanto, esta definición sigue sin ser terminal o válida como terminal.

Además; ese código devuelve la lista tal cual, de manera que:

$$\begin{aligned}
&\text{rev } [1; 2] \\
&\text{aux } [] [1; 2] \\
&\text{aux } [1] [2] \\
&\text{aux } [1; 2] []
\end{aligned} \quad \left. \right\} [1; 2]$$

La implementación correcta terminal sería:

```
let rev l =  
    let rec aux li = function  
        [] → li  
        | h::t → aux (h::li) t  
    in aux [] l;;
```

li es como un acumulador. Ahora si que es una función recursiva terminal. Funciona de la siguiente manera:

```
rev [1;2]  
aux [] [1;2]  
aux [1] [2]  
aux [2;1] []  
[2;1].
```

Podría hacerse con "aux ([h] @ li) t", pero quedamos en lo mismo que antes; no es terminal. Tiene cabecera li y cola li \Rightarrow (h::li). No tiene que ver, ni relación, la eficiencia con que sea terminal o no. El funcionamiento no terminal sería:

```
rev [2;3;1]  
: rev [3;1] @ 2  
(rev [1] @ 3) @ 2  
((rev []) @ 1) @ 3) @ 2  
([1] @ 3) @ 2  
([1;3]) @ 2  
[1;3;2]
```

ORDENACIÓN POR SELECCIÓN

Para ello necesitamos dos funciones:

Función `minl` $\dashv : 'a \text{ list} \rightarrow 'a$

Función `remove` $\dashv : 'a \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$

```
let rec ordenacionporseleccion = function
  [] → []
  | l → let m → minl l in
    let resto → remove m l in
    m :: ordenacionporseleccion resto;;
```

No se le puede dar el valor de `n::t`, porque ya lo hacen todas las otras funciones, necesitamos la lista completa.

El problema que nos ofrece esta función, es que tiene complejidad cuadrática, ya que `minl` recorre la lista entera y `remove`, aunque haga la lista cada vez más pequeña, también la recorre entera.

```
let minl (n::t) =
  List.fold_left min n t;;
```

Nos da un aviso conforme no está definido para la lista vacía [], pero no hace falta, ya que se hace una comprobación en `ordenacionporseleccion` antes de la llamada a esta función.

Otra posible implementación, esta vez terminal, es:

```
let rec minl = function
  h :: t → h
  | h1 :: h2 :: t → minl (min h1 h2 :: t)
```

Para que nos sea más cómodo, haremos un "open List", de manera que podamos usar cualquier función del módulo List sin tener que usar la forma List.nombrefunción.

```
let rec remove v = function
  [] → []
  | h :: t → if h = v then t
              else h :: remove v t
```

Esta no es una función terminal.

Ahora ya funciona la ordenación por selección (select-sort)

Vamos a mirar el tiempo antes y después de ordenar. Si hacemos la ordenación del doble de elementos, el tiempo se multiplica por 4.

Serie de FIBONACCI.

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

Esta es la función no terminal, que tiene complejidad exponencial. En el TGR III, está implementada de una forma muy completa.

Sys.time ()

- : unit → float

No es funcional. Devuelve el tiempo en segundos que se ha consumido un proceso.

Ahora haremos una función que dé el máximo de una lista. Será de la forma maxl : 'a list → 'a

```
let rec maxl = function  
  [h] → h  
  | h :: t → max h (maxl t)  
  | [] → error;;
```

max es una función predefinida. Hacer terminales ambas funciones (fibonacci y maxl).

Funciones de ordenación. (\leq)

$\dashv : 'a \text{ list} \rightarrow 'a \text{ list}$

Existe una función predefinida en OCaml, perteneciente al módulo Random, que se llama `int` ($\dashv : \text{int} \rightarrow \text{int}$). Lo que hace es coger valores aleatorios. Si se le pasa un valor n , el número aleatorio estará dentro del rango (0 a $(n-1)$).

Si implementamos una función recursiva con esta otra función, podremos obtener una lista de valores aleatorios.

`random_list` : $\dashv : \text{int} \rightarrow \text{int} \rightarrow \text{int list}$.

```
let rec random_list r n
  =
  if (n >= 0) then
    random.int (n) :: random_list r (n - 1)
  else
    [];;
```

No es terminal.

ELENA
DELAMANO
FREIJE

Vamos a implementar una función que nos diga el tiempo que consume la aplicación de f a x .

```
let crono f x =
  let t = Sys.time() in
  let y = f x in
  Sys.time () -. t;;
```

En vez de utilizar la variable y , podemos poner un comodín, ya que no tiene ningún uso.

```
# List.map (crono selec) (List.map (randomlist
max_int) [1000; 2000; 4000; 8000]);;
```

Map aplica una función a cada uno de los elementos de una lista, dando como resultado otra lista.

Debemos mejorar lo de arriba en dos pasos el min y el resto. Coge una lista y pone ambos en un par → 'a list → 'a × 'alist.

En el caso del map de arriba, se devuelven una lista cuyo primer elemento es una lista de 1000 elementos, el segundo una lista de 2000, el tercero una de 4000 y el último, una de 8000. Después, con cada una de ellas se hace crono, y nos da una lista de 4 valores, resultados de cuánto tarda en ordenarse una lista de 1000, 2000, 4000 u 8000 elementos.

ORDENACION POR INSERCIÓN

Función a la que se le pasa una lista ordenada y un valor a ordenar, y que devuelva una lista también ordenada.

Crearemos para ello la función insert.

insert : $\text{List } \alpha \rightarrow \alpha \rightarrow \text{List } \alpha$.

```
let rec insert v :  
  [] → [v]  
  | h :: t → if v < h then  
    v :: t
```

Un modelo. Implementarlo.

No es un modelo de recursividad, pero mi implementación ha sido:

```
let rec insert v l = match l with  
  [] → [v]  
  | h :: t → if (v < h) then  
    v :: t  
    " else h :: insert v t;;
```

```

| h1::t1, h2::t2 -> (f h1 h2)::(map2 f t1 t2);;

(* Función Fold-left: fold_left *)
let rec fold_left f a l = match l with
    [] -> a
    | h::t -> fold_left f (f a h) t;;

(* Función Fold-right: fold_right *)
let rec fold_right f l a = match l with
    [] -> a
    | h::t -> f h (fold_right f t a);;

(* Función For all: for_all *)
let rec for_all p l = match l with
    [] -> true
    | h::t -> if not (p h) then false
                else for_all p t;; 

(* Función Exists: exists *)
let rec exists p l = match l with
    [] -> false
    | h::t -> if (p h) then true
                else exists p t;; 

(* Función Mem: mem *)
let rec mem e l = match l with
    [] -> false
    | h::t -> if (e = h)
                  then true
                  else mem e t;; 

(* Función Find: find *)
let rec find p l = match l with
    [] -> raise(Not_found)
    | h::t -> if p h
                  then h
                  else find p t;; 

(* Función Filter: filter *)
let filter p l =
    let rec aux p l1 l2 = match l1 with
        [] -> rev l2
        | h::t -> if p h
                      then aux p t (h::l2)
                      else aux p t l2
    in aux p l [];; 

(* Función Find all: find_all *)
let for_all p l =
    let rec aux p l1 l2 = match l1 with
        [] -> rev l2
        | h::t -> if p h

```

```

| _ -> raise(Failure "hd");;

(* Función Tail: tl *)
let tl = function
  _::t -> t
| _ -> raise(Failure "tl");;

(* Función Length: length *)
let length l =
  let rec aux l n = match l with
    [] -> n
    | _::t -> aux t (n+1)
  in aux l 0;;

(* Función Append: append *)
let rec append lista1 lista2 = match lista1 with
  [] -> lista2
  | h::t -> h::(append t lista2);;

(* Función Reverse: rev *)
let rev lista =
  let rec aux l1 l2 = match l1 with
    [] -> l2
    | h::t -> aux t (h::l2)
  in aux lista [];;

(* Función Nth: nth *)
let rec nth lista index = match lista, index with
  | [], _ -> raise(Failure "nth")
  | h::_, 0 -> h
  | _, t, _ -> nth t (index-1);;

(* Función Concat: concat *)
let rec concat lol = match lol with
  [] -> []
  | h::t -> append h (concat t);;

(* Función Flatten: flatten *)
let rec flatten lol = match lol with
  [] -> []
  | h::t -> append h (flatten t);;

(* Función Map: map *)
let rec map f l = match l with
  [] -> []
  | h::t -> (f h)::(map f t);;

(* Función Map2: map2 *)
let rec map2 f l1 l2 = match l1, l2 with
  | _, [] -> raise(Failure "map2")
  | [], _ -> raise(Failure "map2")
  | [], [] -> []

```

```

[] -> []
| h::t -> if h=e then (rev l2 @ t) else aux e t (h::l2)
in aux e l [];;
(* Función Remove all: remove_all *)
let remove_all e l =
  let rec aux e l1 l2 = match l1 with
    [] -> l2
    | h::t -> if h=e then aux e t l2 else aux e t (l2 @ [h])
  in aux e l [];;
(* Función Join: join *)
let rec join q l1 l2 = match l1,l2 with
  [],_ -> []
  | _,[] -> []
  | (a,c)::t1,(b,d)::t2 -> let resto = append (join q [(a,c)] t2) (join q t1 l2) in if q a b then
    (a,b,c,d)::resto
  else
    resto;;
(* Función Natural Join: natural_join *)
let rec natural_join l1 l2 = match l1,l2 with
  [][] -> []
  | [],_ -> []
  | (a,b)::t1,(c,d)::t2 -> let resto = append (natural_join [(a,b)] t2) (natural_join t1 l2)
    in if a = c then (a,b,d)::resto
    else resto;;
(**@Author Manoel - TutorialesNET
@Archivo sort.ml
**)
(* _____ *)
(* crono - función que mide el tiempo de ejecución de una función *)
let crono f x =
  let t = Sys.time () in
  let _ = f x in
  Sys.time () -. t;;
(* Pseudo-función rList - No es terminal*)
let rec rlist r l =
  if l <= 0 then []
  else Random.int r :: rlist r (l-1);;
(* >> La función rList propuesta no es recursiva terminal *)
(* Pseudo-función rList_t - Recursiva terminal*)
let rlist_t r l =
  let rec aux r l list = match l with
    x when x < 0 -> raise(Failure "rList")
    | 0 -> list

```

```

        then aux p t (h::l2)
        else aux p t l2
    in aux p l [];

(* Función Partition: partition *)
let rec partition p = function
    [] -> [], []
    | h::t -> let t1,t2 = partition p t in if p h
        then h::t1,t2
        else t1,h::t2;; 

(* Función Assoc: assoc *)
let rec assoc a l = match l with
    [] -> raise(Not_found)
    | h::t -> if a = (fst h)
        then snd h
        else assoc a t;; 

(* Función Mem Assoc: mem_assoc *)
let rec mem_assoc a l = match l with
    [] -> false
    | h::t -> if a = (fst h)
        then true
        else mem_assoc a t;; 

(* Función Remove Assoc: remove_assoc *)
let rec remove_assoc a l =
    let rec aux a l param = match l,param with
        [],_ -> []
        | h::t,1 -> h::(aux a t 1)
        | h::t,0 -> if a = (fst h)
            then aux a t 1
            else h::(aux a t 0)
        | _ :: _ -> []
    in aux a l 0;; 

(* Función Split: split *)
let rec split l = match l with
    [] -> [], []
    | (a,b)::t -> let t1,t2 = split t in
        a::t1, b::t2;; 

(* Función Combine: combine *)
let rec combine l1 l2 = match l1,l2 with
    _ :: _ :: [] -> raise(Failure "combine")
    | [] :: _ :: _ -> raise(Failure "combine")
    | [],[] -> []
    | h1::t1,h2::t2 -> (h1,h2)::(combine t1 t2);;

(* Función Remove: remove *)
let remove e l =
    let rec aux e l1 l2 = match l1 with

```

ORDENACIÓN POR FUSIÓN

- ① Dividir a la mitad.
- ② Ordenar por separado. (RecurSividad)
- ③ Reunir en un solo montón (sumar).

Va de ' a list' \rightarrow ' a list'.

La ordenaremos de menor a mayor según la relación de ($<=$).

Por lo tanto, para implementar la función de ordenación por fusión necesitamos dos funciones: una que divide la lista en dos partes del mismo tamaño, y otra que une las funda.

reporte : ' a list' \rightarrow (' a list * ' a list)

merge : (' a list * ' a list) \rightarrow ' a list

En la función principal necesitamos ordenar las dos mitades antes de fundirlas en una sola.

FUNCIÓN MERGE

```
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);;
```

Se podía haber escrito cambiando los argumentos:

merge (l1, l2) = match l1, l2 with.

FUNCIÓN REPARTE

Para repartir, se realiza de manera sencilla, como en la vida real: "uno para mí, uno para ti".

```
let rec reparte = function
  h1::h2::t -> let t1, t2 = reparte t in
    (h1::t1, h2::t2)
  | _ -> ([], []);;
```

En la última línea de código se mete todo en uno de los lados; o es una lista vacía o una lista con un sólo elemento la última iteración.

```
let rec m-sort l =  
    merge (reparte l);
```

Esta función no nos sirve. Tenemos que, como hemos dicho antes, ordenar los trozos antes de fundir las listas.

```
let rec m-sort = function  
    [] → [] | [h] → [h]  
    | l → let l1, l2 = reparte l in  
        merge (m-sort l1, m-sort l2);
```

Ahora sí que estaría correcta. Sería una función de 'a list → 'a list.

ELENA
DELAMANDO
FREIJE

REPASO GENERAL HASTA EL MOMENTO

TIPOS BÁSICOS

- unit → ()
- bool → true, false
- char → 'a', 'b', ... '=', ...
- int → 1, 2, 3 ... / min-int, max-int
- float → 1.5, 2.3 ...
- string → "abc", "hola" ...
- exn → exception → division-by-zero / failure of /
/invalid-argument / stack-overflow

PRODUCTO CARTESIANO

Tuplas:

$$\left[\begin{array}{l} \text{let } : \alpha_1 \\ \text{let } : \alpha_2 \\ \dots \\ \text{let } : \alpha_n \end{array} \right] \quad \text{let } \text{let } \dots \text{let } : \alpha_1, \alpha_2, \dots, \alpha_n$$

LISTAS.

Las listas son polimórficas, y homogéneas.

[] : 'a list → lista vacía polimórfica.

$$\left[\begin{array}{l} \text{let } := \alpha \\ \text{let } = \alpha \text{ list} \end{array} \right] \quad \text{let } := \alpha \text{ list.}$$

Vamos a ver cómo definir nuevos tipos a partir de los ya existentes (unión disjunta). Todos los valores son disjuntos, ya que no hay valores que estén en dos tipos a la vez. Los valores del nuevo tipo no pueden coincidir con ninguno de los existentes.

Imaginemos que queremos hacer una lista de int y char. Como las listas son homogéneas, para que se nos permita esta heterogeneidad, podemos crear un nuevo tipo que contenga estos dos básicos:

```
type charint = En of char  
| In of int;;
```

Cada constructor está asociado a un tipo que ya existe. Son funciones que insertan cada uno de los tipos en uno sólo.

Es obligatorio que el constructor empiece por mayúscula.

Utilizándose:

```
Ch 'a';; → -: charint = Ch 'a'  
Ch 'a' = Ch 'b';; → false  
In 8;; → -: charint = In 8  
In 8 = Ch 'a';; → false
```

Los constructores juegan un papel importante, ya que se pueden usar en el pattern matching.

No se pueden hacer estas comparaciones:

In $\emptyset = \emptyset$;;	Comparaciones que llevan a error, ya que son tipos distintos.
Ch 'a' = 'a';;	

Trabajemos ahora con funciones sobre este tipo:

FUNCIÓN INT-OF-CHARINT

Función que lleva el tipo charint a

- Char → código ASCII.
- Int → int.

```
let int-of-charint = function
  In n → n
  | Ch c → int-of-char (c);;
```

Para probarlo:

```
let L = [In  $\emptyset$ ; Ch 'a'; Ch 'z'];;
open List;;
map int-of-charint L;;
-: list int = [ $\emptyset$ , 97, 122]
```

Ahora crearemos un tipo que tenga el tipo int dos veces, clasificándolos en izquierda y derecha.

```
type dint = L of int | R of int;;
```

L 3;; → -: dint = L 3
R 5;; → -: dint = R 5
R 3;; → -: dint = R 3
R 3 = L 3 → false. Es distinto porque
no ha sido declarado con otro constructor.
R 3 = R 5 → false. Es distinto porque está
construido a partir de otro int.

Es un tipo que para cada int tiene dos valores.

FUNCIÓN DINTORD

Función que nos indica si está ordenado o no, dándole preferencia al constructor L como más pequeño, y teniendo en cuenta los valores, luego:

```
let dintord dint dint = match dint, dint with  
  L _, R _ → true,  
  | R _, L _ → false  
  | R m, R n → m <= n  
  | L m, L n → m <= n;;
```

Con los - (comodines) se hace que dependa sólo del constructor, independientemente del valor que a este se le pase.

Los tipos también nos sirven para hacer copias de tipos básicos:

```
type int 2 = D of int;;
```

Ahora vamos a crear un tipo que represente los números naturales.

```
type nat = Z of unit | S of nat;;
```

Las declaraciones de tipos, por defecto, son recursivas, por lo que no hace falta ponerle un rec.

Z ();; → es el único valor que se puede hacer con este constructor.

S (Z())
S (S (Z())) } los valores que se pueden declarar con S son infinitos.

Z es un φ.

S es el siguiente nat ...

FUNCIÓN NATORD

Vamos a definir el orden; cuando un nat es menor que otro.

```
let rec natord n1 n2 = match n1, n2, with
    | Z(), _ → true
    | Sm, Z() → false
    | Sm, Sn → natord m n;;
```

En la declaración del tipo, es mejor poner solamente Z, en vez de Z or unit. El valor "unit" ya se le asigna por defecto.

Son constructores constantes.

```
let palo = Picas | Diamantes | corazones | Tréboles;;
```

Cada constructor constante de un valor tiene 4 valores (palo). Recuerda a los tipos enumerados de Pascal.

ELENA
DELAMANO
FREITIE

EJERCICIOS PROPUESTOS PARA TGR III

- ① Realizar una función que devuelva el iésimo elemento de la serie de Fibonacci.

Para ello necesitaremos:

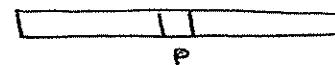
- `int-of-string`
- Módulo `sys.argv`. (1) ← que accede al elemento 1 del vector `argv` del módulo `sys`. Da como resultado un `string`, y tenemos que convertirlo a un entero.
- `print-endline : string → unit`.
- tener en cuenta los errores (mensajes de error).
- Límite de funcionamiento = 10 millones (como entrada). El problema es que el resultado supera el límite de los `int`, por lo que tendremos que usar la librería `Num`; módulo `Num`, con el tipo de dato definido `num`. Sirve para representar números enteros arbitrariamente grandes, y operaciones muy exactas. como sugerencia, pues, de `int → num`.
- Recursividad terminal, para que no exista desbordamiento de pila.

- ② Realizar lo mismo, pero para el factorial.

- ③ Ordenación por fusión no es terminal.

Implementarla de manera que sí lo sea para que no haya problemas de desbordamiento de la pila.

④ Ordenación por quicksort



Cogemos p como referencia, separamos para un lado aquellos que sean menores, y, para el otro lado, los que sean mayores. Luego, se unen los dos. El problema del Quicksort es el punto de referencia si se consigue implementarlo terminal.

```
# Sys.argv;;
# Array.length;;
#, Sys.argv.(0);;
```

Para acceder a las "celdas" del array, en Ocaml es entre paréntesis.

ÁRBOLES BINARIOS

Comenzaremos definiendo el tipo:

```
type inttree =
  Vacio | Nodo of (int * inttree * inttree);;
```

Ejemplos de inttree: type trifi = Empty | Comp of 'arifib * 'arifib

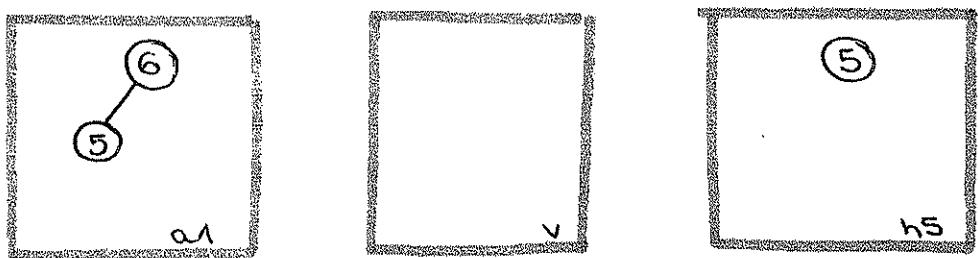
```
Vacio
Nodo (5, Vacio, Vacio)
```

```

let v = Vacio;;
let h5 = Nodo (5, Vacio, Vacio);;
let a1 = Nodo (6, h5, Vacio);;

```

val a1 : inttree = Nodo (6, Nodo (5, Vacio, Vacio) Vacio).



FUNCTION RAÍZ

Vamos a implementar una función que nos devuelva la raíz del árbol que se le pasa como parámetro.

```

let raiz := function
  Vacio → raise (Failure "raiz")
  | Nodo (r, _, _) → r;;

```

Si se quiere, podemos crear una nueva excepción y sustituirla por el Failure, de manera que:

```

exception No_Existe_Raiz;;
~ Vacio → raise No_Existe_Raiz ~

```

FUNCIÓN ALTURA

```
let rec altura = function
  | Vazio → 0
  | Nodo (l, c, d) ⇒ 1 + max (altura l) (altura d);;
```

ELENA
DELAMAND
FREIJE

ÁRBOLES BINARIOS POLIMÓRFICOS

```
type 'atree =
  Empty | Node of ('a * 'atree * 'atree);;
```

Ahora los árboles pueden ser de cualquier tipo.
 'a es un parámetro de tipo. Con esto estamos definiendo miles de tipos, uno para cada tipo de 'a.
 De esta manera,

```
Node (3, Empty, Empty) → inttree
Node ('a', Empty, Empty) → chartree
```

Función que sirve para crear un árbol con 'x' raíz, y que tiene las ramas vacías.

```
let crearArbol v = Node (v, Empty, Empty);;
```

FUNCIÓN ROOT

Función idéntica a la "raiz" implementada antes, sólo que adaptada a este tipo de datos.

```
let root = function
    Empty → raise (Failure "root")
  | Node (r, _, _) → r;;
```

FUNCIÓN NNODOS

Función que nos indica el número de nodos que tiene el árbol. ('a tree).

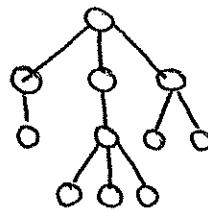
```
let rec nnodos = function
    Empty → 0
  | Node (_, i, d) → 1 + (nnodos i) + (nnodos d);;
```

ÁRBOL

```
type 'a arbol =
  | Hoja of 'a
  | Nodo of ('a * 'a arbol * 'a arbol);;
```

Ahora no se puede tener una rama o un árbol vacío. Tiene que ponerse \emptyset o algo.

ÁRBOLES CON X RAMAS



Ahora, el número de ramas puede ser mayor que dos.

'a gtree → árbol general.

```
type 'a gtree =  
    Vacio | Nodo of ('a * 'a gtree)
```

Esta definición sería incorrecta, ya que sería solamente una raíz y una rama (un nodo). Necesitamos una lista, así meteremos un valor en la raíz y una lista de ramas (lista de árboles realmente) para representar la colección de ramas.

```
type 'a gtree =  
    Vacio | Nodo of ('a * 'a gtree list);
```

Aún así podemos hacerle una modificación más: Podemos quitar el constructor "Vacío", ya que ahora no hace falta considerarlo. Lo más sencillo serían árboles que sólo tienen raíz.

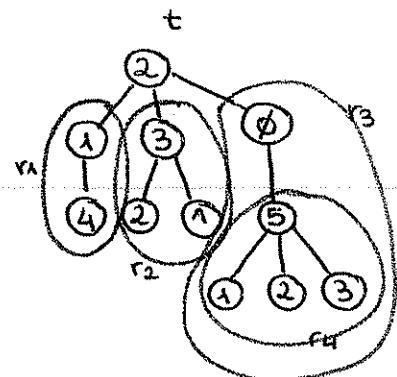
```
type 'a gtree =  
    Nodo of ('a * 'a gtree list);
```

Para un árbol del tipo 

```
let t1 = Nodo (1, []);  
let t2 = Nodo (2, []);
```

Let ar x = Nodo (x, []); → Función que sirve
para crear árboles que
sólo tienen raíz.

Para crear un árbol sólo hay que ir con calma,
haciendo las ramas, y luego lo principal!



```
let r4 = Nodo (5,[Nodo (1,[]); Nodo (2,[]);  
Nodo (3,[])]);;  
let r3 = Nodo (4,[r4]);;  
let r2 = Nodo (3,[Nodo (2,[]);  
Nodo (1,[])]);;  
let r1 = Nodo (1,[Nodo (4,[])]);;  
let t = Nodo (2,[r1;r2;r3]);;
```

Es importante poner siempre el constructor, y
respetar su forma.

Las funciones son valores

$$\text{abs}(2-5) = 3$$

↓
Primero se evalúa el argumento,

y después se aplica la función

> function $x \rightarrow 2 * x$

• El compilador entiende que es una función de int a int

> $(\text{function } x \rightarrow 2 * x)(2+1);;$
 ↓ ↓
 función argumento

> int = 6

> let $x=2+1$ in $2 * x$

> int = 6

> let doble = function $x \rightarrow 2 * x;;$ let doble $x = 2 * x$

abreviatura

> doble(2+1);;

> int = 6.

> let abs = function $x \rightarrow \text{if } x >= 0 \text{ then } x \text{ else } -x;;$

que esto de un float

if $\langle b \rangle$ then $\langle e_1 \rangle$ else $\langle e_2 \rangle$

ESTRUCTURA DE IF

.se considera una expresión

> abs(-1.5)

$\langle b_1 \rangle \text{ || } \langle b_2 \rangle$ OR $\boxed{\langle b_1 \rangle \text{ y } \langle b_2 \rangle \text{ bool}}$ if $\langle b_1 \rangle$ then true else $\langle b_2 \rangle$
 $\langle b_1 \rangle \& \langle b_2 \rangle$ AND if $\langle b_1 \rangle$ then $\langle b_2 \rangle$ else false

let rec fact = function n →

↓
que va a haber
recursividad.

if $n > 0$ then $n * \text{fact}(n-1)$
else 1

||

(Forma abreviada) let rec fact n = if $n > 0$ then $n * \text{fact}(n-1)$ else 1;;

REGAS

α $\alpha \text{ list}$ Ejemplo int int list	$\textcircled{1} \quad [\] \propto \text{list}$ <small>valor más simple</small>	$\textcircled{2} \quad \begin{matrix} \text{hd} \\ + \\ \text{tad} \end{matrix} \text{ list} \rightarrow \begin{matrix} \text{cabeza lista} \\ \text{cola lista} \end{matrix}$ $(\text{hd} :: \text{tad}) \propto \text{list}$	 <i>(es la parte de la lista que no es la cabeza)</i>
--	---	---	--

LISTA DE ENTEROS

$[] \text{ int list}$

$[5; 1] = 5 :: (1 :: [])$

$(1 :: []) \text{ int list}$

$(3 :: []) \text{ int list}$

$(5 :: (1 :: [])) \text{ int list}$

List.hd (lista) → devuelve la cola

List. (funciones para listas)

no se puede pasar
una lista vacía

let rec last e = if List.hd e = $[]$ then List.hd e
else last (List.tl e);;

devuelve el último elemento, pero no
puede pasar listas vacías

$'a \text{ lista} \rightarrow 'a$

cualquier tipo POLIMORFISMO

Let uppercase = function \rightarrow

$(c \geq 'a' \& c \leq 'z') \text{ || } (c = '\backslash224' \& c = '\backslash240') \text{ || }$ rango de valores
 $(c \geq '\backslash224' \& c \leq '\backslash254') \text{ then }$ donde funciona
 Uppercase..

char_of_int (int_of_char $c - 32$) else c ; ;;

ISO 8859 -1 Tabla

ISO 8859 -15 Tabla variante de la 1

for i = 0 to 255 do buckle "for"

recibe n, imprime numeros hasta n.

int \rightarrow int list

Let rec primeros n =

\emptyset n=0 then [] else

primeros (n-1) @ [n];;

Darle para concatenar listas, ya que el segundo elemento, no es un elemento, sino una lista.

Let rec primeros_chars n =

\emptyset n=0 then [] else

primeros_chars (n-1) @ [char_of_int (n-1)];;

Uppercase c = Char_uppercase c & Comprobar (te c)

equivalencia

Let comprobar = function

[] \rightarrow true

| h : t \rightarrow (uppercase h = Char_uppercase h) & &
 Comprobar t;;;

Let comprobar l =

if l = [] then true else Let

c = hd l
 in

if uppercase c = Char_uppercase c then
 Comprobar (tl l) else false

Let suma $p = fst \cdot p + snd \cdot p$

: Val suma : int × int → int

suma(3,7)

Let suma $(x,y) = x + y$

Let suma = function $(x,y) \rightarrow x + y$

Let $p = (2,3)$

: Val $p : int \times int = (2,3)$

Let $(x,y) = p$

: Val $x : int = 2$

: Val $y : int = 3$

Let $(x,y) = p$ in $x + y$

pattern - matching → comparación de patrones

Let $l = [3; 5; 7];;$

Val $l : int list = [3; 5; 7]$

Let $(h:t) = l$

Val $h : int = 3$

Val $t : int list = [5; 7]$

Si $l = int list []$

Error de ejecución / error de pattern-matching

Haskell Curry

let sum n = function x → n + x

val sum : int → (int → int)

$$(\text{Sum } 3) 5 = 8$$

let sum = function n → (function x → n + x)

let sum n x = n + x

$$3 + 5$$

$$(+) 3 5$$

(^) (Concatenar strings)

: string → string → string

(@) (Concatenar lists)

: 'a list → 'a list → 'a list

let head (h:t) = h (devuelve cabeza lista)

let head (h:_) = h

→ Abre la misma función que un nombre,
sin usar el nombre.

Dar el error identificado.

raise: exn → 'a

let hd = function (h:_) → h ;;

let hd l = if l = [] then raise (Failure "hd")

else [let h:t = l in h ;;
hd l → no es recursividad, llama a la otra función.
más ejemplos]

Tercera operación:

función es más compleja:

function $\langle p_1 \rangle \rightarrow \langle \beta_2 \rangle$
| $\langle p_2 \rangle \rightarrow \langle \beta_3 \rangle$

} varios patrones que se interpretan
por orden.

let rec f = function

$h :: _- \rightarrow h$
 | [] \rightarrow raise (failure "hd")

↑ en este caso el orden es irrelevante.

Redefinir length.

let rec length l =

 if $l = []$ then \emptyset
 else $_ + \text{length}(\text{tl } l)$

↑
Redefinida antes

On pattern-matching:

let rec length = function

$_ :: t \rightarrow 1 + \text{length} t$
 | [] $\rightarrow \emptyset$

uso de match.

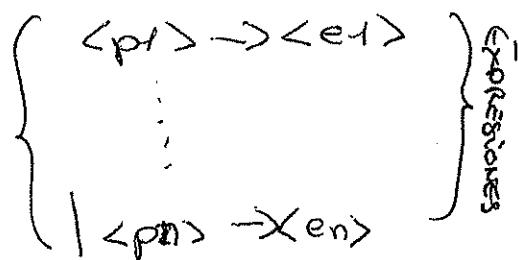
match <expresión> with
 <patrón> \rightarrow <expresión>
| $\langle p_2 \rangle \rightarrow \langle e_2 \rangle$
 ⋮

let rec nth l n = match (n, l) with
 (-, []) \rightarrow raise (failure "nth") |
 (0, $h :: _-$) \rightarrow ~~nth h~~ h |
 (n, $_ :: t$) \rightarrow nth t (n-1);;

"~~Interceptar errores~~" = "Interceptar una excepción."

`try <e> with`

PATRONES



`let rec length l = 1 + length (list.tl l);` → en el caso de lista vacía no funciona,
 luego la recursividad también fallaría (porque el caso base está mal)

|| = try
`1 + length (list.tl l)`
`with failure "tl" → 0;;`

• Se ha interceptado exactamente el error que ya conocíamos

Ahora el caso base ya está solucionado (con la lista vacía, ahora la longitud es 0)

`let rec fronto n = if n = 0 then []`

`else n :: (fronto (n-1))`

función si una lista está ordenada o no.

sort : 'a list → 'a list

let rec sorted = function

| $h_1::h_2::t \rightarrow h_1 \leq h_2 \& \text{sorted}(h_2::t)$ (listas con al menos 2 elementos)

| - → true;; (cualquier otra lista)

Métodos para ordenar (ordenación por inserción)

insert : 'a → 'a list → 'a list

insert : 'a → 'a list → 'a list

let rec insert x = function

| [] → [x]

| $h::t \rightarrow \text{if } x \leq h \text{ then } x::h::t$
else $h::\text{insert } x \ t;;$

} insertar en una lista
ordenada

let rec isort = function

| [] → []

| $h::t \rightarrow \text{isort } h(\text{isort } t);;$

let sort l =

let rec aux = function

([], [], res) → res ($\epsilon, \leq, h::\text{res}$) | (n::t, res, []) → aux(t, res, [h])

([], n::t, res) → aux((), res)

| (h::t, res, h2::t2) → if $h \leq h_2$ then aux(t, h::res, h2::t2)

in aux(l, [], [])

else aux(t, h2::res, h::t2)

$x=1$

$a = 4 + x$

$a = 2 \quad x = 2$

$b = x + x$

$b = 3 \quad x = 3$

let rec fact n = if $n > 0$ then $n * \text{fact}(n-1)$
else 1;

NO TERMINAL

PP

let fact2 n =

let rec aux (i, f) = if $i > n$ then f

else aux(i+1, (i+1)*f)

TERMINAL

no deja cuentas pendientes
la aplicación se hace

in aux(0, 1);

al final de todo.

let fact3 n =

let rec aux (i, f) =

if $i > 0$ then aux($i-1, i*f$)

else f

in aux(n, 1);

let rec rev = function

$[T \rightarrow T]$

| $h : E \rightarrow \text{append}(\text{rev } t) [^h]$

\Rightarrow

let rev l =

let rec aux f2 = function
[] -> f2

| h::t -> aux(h; f2) t

in aux [] l

Fibonacci (n-1) + (n-2) + (n-3) * TGR

Sys.time int → float ¿Cuánto tiempo ha consumido lo que estoy

Ejecutando?

Random.int int → int

(devuelve un valor en 0 y n-1)

let rec rlist r n = ~~list~~ ⁿ⁻¹ elementos

if n > 0 then Random.int :: rlist r (n-1)

else []

let count f x =

(let t = Sys.time() in

let _ = f x in

Sys.time() - t

let merge = function

(l, []) | ([] e) -> l

| (h1::t1; h2::t2) -> if h1 < h2
then h1::merge(t1, h2::t2)
else h2::merge(h1::t1, t2)

let rec msort l =

let f1, f2 divide l in

merge (msort f1, msort f2);;

[] | [] -> l

let rec divide = function

h1::h2::t -> let t1, t2 = divide t in
h1::t1, h2::t2

| l -> l, [];;

~~fixed~~

faltan los casos base
solo falta en listas con
1 o 0 elementos.

Práctica de PP

List nth : a list → int → 'a
()

Devuelve el
n-ésimo término.

-: unit ()

2+5*3;;

-: int = 17

1.0;;

-: float = 1.

1.0 *2;;

Error (está multiplicando dos tipos distintos)

2-2.0;;

mismo error

30 +2.0;;

mismo error (operando para float → +)

5/3;;

: int = 1

5 mod 3;;

: int = 2

3.0 *. 2.0 ** 3.0;;

: float = 24.

3.0 = float_of_int 3;;

: true

sqrt 4;;

Error (esperaba un float) int_of_float 2.1 + int_of_float (-2.9);;
: int = 0

TEORÍA.

Tipos de datos en Ocaml

LAP

Type numero = In of int | Fp of float

type in2 = I2 of int;;

type i3 = I3 of int | I2 of int;;

let in2le x y = match (x, y) with
 $(I2x, I2y) \rightarrow$

type intplus = I of int | Err of unit

Type nor =

O : S of int

type ilist

V (lista vacía)
con (esta lista)

let rec ilength = function

$V \rightarrow 0$

| com (x, l) $\rightarrow 1 + ilength(l)$

reinas : int \rightarrow (int \times int) list

type 'a option

None

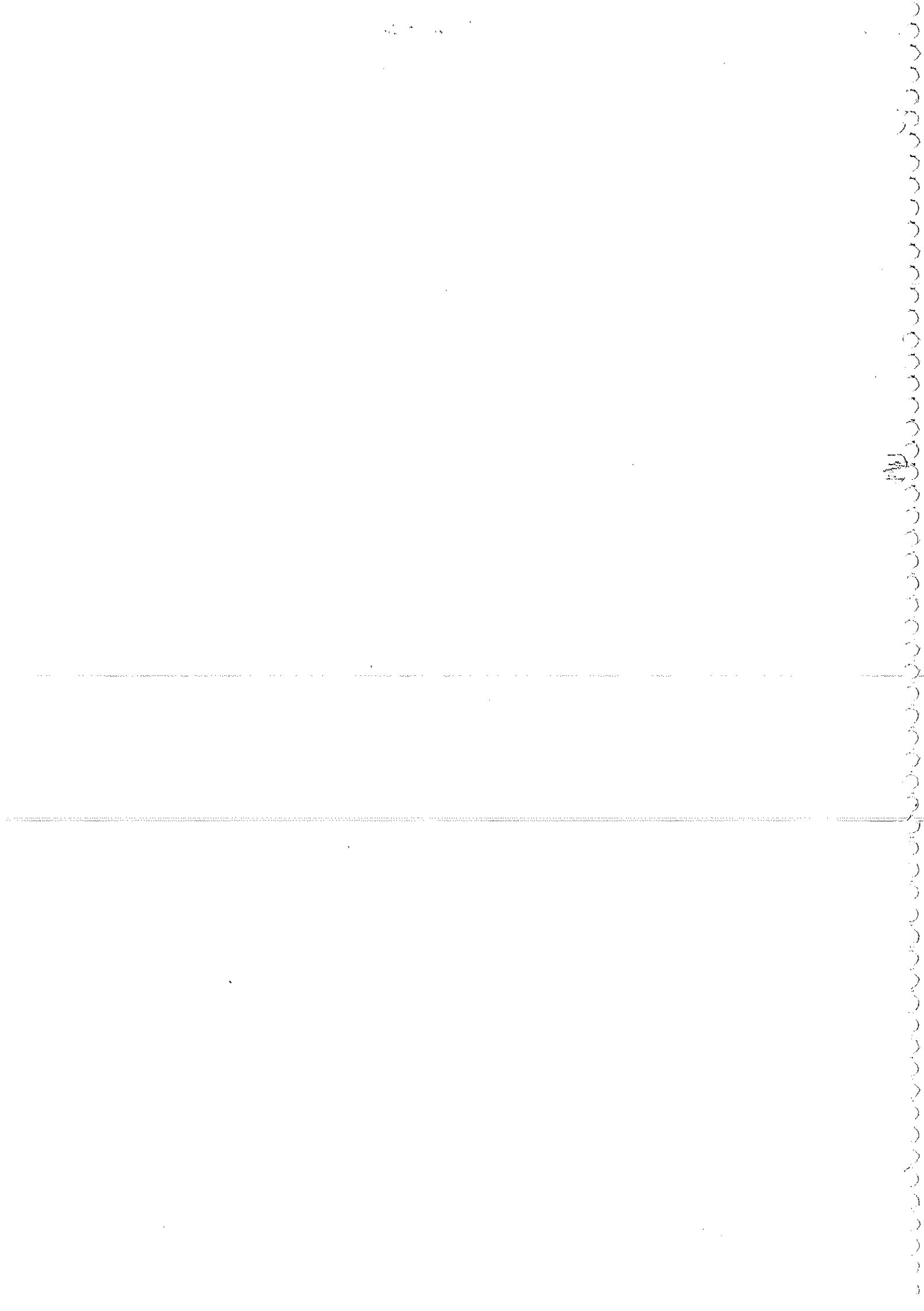
| Some of 'a

all_reinas : int \rightarrow (int \times int) list list

TGR Knights Tour KT : int \rightarrow int \rightarrow (int \times int) \rightarrow (int \times int) list

Buscar camino para que un caballo recorra todo el tablero de ajedrez sin repetir casillas. Hasta 7x7 debería funcionar.

(función que verifica si la lista de soluciones es correcta.)



```
typ 'a treebi =
| Empty
| Comp of 'a treebi * 'a * 'a treebi;
```

```
let rec inorden = function
| Empty → []
| Comp(i, r, d) → inorden i @ r :: inorden d;
```

```
let rec hojas = function
| Empty → []
| Comp(Empty, r, Empty) → [r]
| Comp(i, r, d) → hojas i @ hojas d;
```

```
insert : 'a → á tree bi → 'a treebi
```

```
let rec insert x = function
```

```
| Empty → Comp(Empty, x, Empty)
| Comp(i, r, d) → if x = r
    then Comp(insert x i, r, d)
    else Comp(i, r, insert x d);
```

```
let rec totree = function
```

```
| [] → Empty
| h::t → insert h(totree t);;
```

```
let qt sort l = inorden(totree l);;
```

```
type 'a nelist → Ejemplo S6
```

```
| S of 'a
```

```
| C of 'a * 'a nelist
```

```
C(6, 55)
```

```
C(2, 53)
```

```
C(2, C(2, 54))
```

```
type 'a sbtree =
```

```
| H of 'a
```

```
| N of 'a * 'a sbtree * 'a sbtree;;
```

```
→ Ejemplo H7
```

Las hojas con una sola rama no se pude

```
type á tree =
```

```
| S of 'a
```

```
| T of 'a .. 'a tree ..
```

```
S3
```

```
T(3, [S2, S3])
```

type tree

~~solo~~

una definición

type 'a tree =

T of 'a tree list

} así solo hay una manera
de representar los árboles

~~f~~ función nodos : 'a tree → int

~~nodos~~

let rec nodos = function

$T(-, e) := \text{List.fold left} (+)$

(list.map nodos e)

↓
lista de
árboles

let rec nodos = function

$T(-, e) \Rightarrow 1 | T(r, h::t) \rightarrow \text{nodos } h + \text{nodos } (T(r, t))$

FUNCTION NNODES

```
let rec nnode = function  
  | Nodo (_, []) -> 1  
  | Nodo (r, n::t) -> nnode h + nnode Nodo (r, t);;
```

Es la función que nos indica cuántos nodos tiene el árbol.

Podemos encontrar más maneras de implementarla, aunque son más complejas, como por ejemplo:

```
let rec nnode (Nodo (_, l)) =  
  1 + fold-left (t) $ (map nnode l);
```

El fold-left va sumando, y el \$ es el caso base.

De las anteriores funciones, quizás también podemos sacar alguna idea para implementar la función que calcule la altura de un árbol.

REPASO GENERAL HASTA EL MOMENTO

- Tenemos
- expresiones
 - definiciones → valores
 - tipos
- Ambo's con
- # [• comandos del compilador interactivo de Ocaml.
 - Load

<exp>

<def-valor> → Let

<def-tipo> → type

open <mod>

exception {<const>

<Vconst> of <tipo>

↳ "Exception Hd → -:<exp>"

Exception Fallo of int.

↳ <declaraciones>

type <nt> = <r1>

| <r2>

| <r3>

:

[<cc>
<r1>
<Vcs> of <tipo>]

type 'a <nt>

*(nt) → nombre tipo.

*name → palabras reservadas

ELENA
DELAMANO
FREIJE

Let $\langle p \rangle = \langle e \rangle$

$\langle p_1 \rangle, \langle p_2 \rangle \dots$] DEFINICIÓN GLOBAL
 $\langle p_1 \rangle :: \langle p_2 \rangle$

$\langle \text{def} \rangle \text{ in } \langle \text{exp} \rangle \rightarrow$ DEFINICIÓN LOCAL

$\langle \text{rec} \rangle$ → variante de la def de valores.

Let $\langle p_1 \rangle = \langle \text{expr}_1 \rangle$
and $\langle p_2 \rangle = \langle \text{expr}_2 \rangle$
:
and $\langle p_n \rangle = \langle \text{expr}_n \rangle$] DEFINICIÓN MÚLTIPLE O
EN PARALELO

Let $x, y = (x+1), 5 ::$

Let $x = x + y$
and $y = x - y ::$] Si cambiamos el orden es
equivalente. $x = 8 ; y = -2$

Let $x = x + y ::$
Let $y = x - y ::$] Ahora ya no son los mismos
resultados. Ahora son
dependientes. $x = 8 ; y = 3$

Let $\langle p_1 \rangle, \langle p_2 \rangle, \langle p_3 \rangle, \dots, \langle p_n \rangle = \langle e_1 \rangle, \langle e_2 \rangle, \langle e_3 \rangle, \dots, \langle e_n \rangle ::$

Let $x, y = x + y, x - y ::$

FUNCIONES MUTUAMENTE RECURSIVAS

```
let rec par n =
  (n = 0) || (impar (n-1));
```

```
and impar n =
  (n ≠ 0) & (par (n-1));
```

PALABRAS RESERVADAS.

~

function

```
function <p1> → <e1>
  | <p2> → <e2>
  | ...
  | <pn> → <en>;;
```

function <p1> → (function l <p2> → <e2>)
 fun <p1> <p2> <e2> $\xrightarrow{\uparrow}$ es equivalente.

function x → (function y → x+y)
 fun x y → x+y $\xrightarrow{\uparrow}$ es equivalente

~

let:

let <n> = function

<p> → <e>;;

let <n><p> = <e>;;

$\xrightarrow{\uparrow}$ es equivalente.

2

match

match $\lambda e \rangle$ with

| $\lambda p_1 \rangle \rightarrow \lambda e_1 \rangle$
| $\lambda p_2 \rangle \rightarrow \lambda e_2 \rangle$,
| $\lambda p_3 \rangle \rightarrow \lambda e_3 \rangle$
| ... $\rightarrow \dots$
| $\lambda p_n \rangle \rightarrow \lambda e_n \rangle //$

2

try

try $\lambda e \rangle$ with

$\lambda p_1 \rangle \rightarrow \lambda e_1 \rangle$
| $\lambda p_2 \rangle \rightarrow \lambda e_2 \rangle$
| ... $\rightarrow \dots$
| $\lambda p_n \rangle \rightarrow \lambda e_n \rangle //$

2

if

if $\lambda b \rangle$ then $\lambda e_1 \rangle$ else $\lambda e_2 \rangle //$

$\lambda b \rangle = \text{bool}$

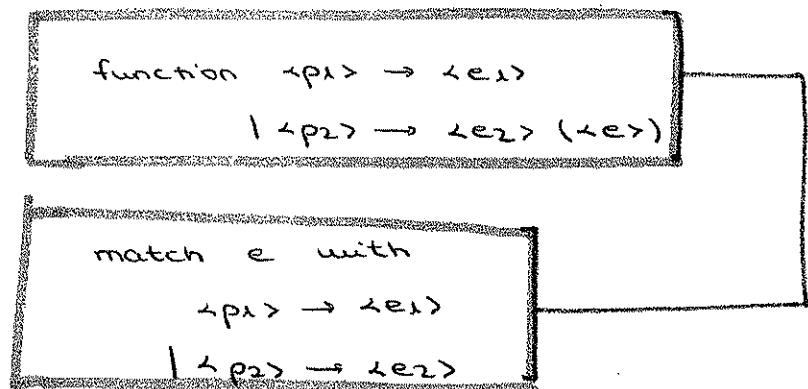
$\lambda e_1 \rangle = 'a$ } mismo tipo.

$\lambda e_2 \rangle = 'a$ }

$\lambda e_1 : \alpha \rightarrow \beta$

$\lambda e_2 : \alpha$

$\underbrace{\lambda e_1 \lambda e_2}_{\beta} \rightarrow \text{aplica } \lambda e_1 \text{ a } \lambda e_2.$



Realizar una función turno, que vaya de
 $\text{unit} \rightarrow \text{int}$.
 $\text{turno}() \rightarrow$ número siguiente al de la vez
anterior que se le ha llamado.

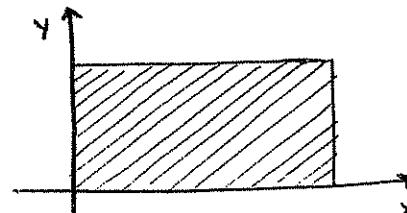
ELENA
DELAHANZO
FREIJJE

Logo interpreter

Logo es un lenguaje de programación de alto nivel, en parte funcional y en parte estructurado. Es conocido por su característica más explotada, los "gráficos tortuga". Esto consiste en dar instrucciones a una tortuga virtual, que es un cursor gráfico usado para crear dibujos. Se maneja mediante palabras que representan instrucciones.

www.calormen.com/jstlogo

Nuestra práctica consiste, pues, en crear un intérprete de Logo, utilizando para ello el módulo Graphics de Ocaml.



La conversión del movimiento según las instrucciones se realizará en este plano.

INSTRUCCIONES

- sl → sube lápiz.
- bl → baja lápiz.
- av → avanza + arg.
- re → retrocede + arg.
- gi → girar izquierda + arg.
- gd → girar derecha + arg.
- bp → borrar pantalla.
- repite {< >} <instr>

Para todo esto necesitamos:

- Crear un TAD que represente las instrucciones.
- Posibilidad de pasárselas cualquier expresión aritmética que tenga como resultado un valor numérico, no sólo constantes.
- Tener una función "print" que nos muestre la solución de una expresión aritmética.
- Posibilidad de concatenación de instrucciones.
- Un tipo de dato que nos indique el estado.
- Una función ejecutar; estado \rightarrow inst \rightarrow nuevoestado.
- Otro tipo de dato para los argumentos. Pero hay que tener en cuenta que, aparte de ser constantes, pueden ser expresiones aritméticas. Por lo que:

- Un caso para cte.	[+
- Un caso para suma.	[-
- Un caso para resta.	[*
- ...	[- ()

Es una declaración de tipo recursiva, ya que, por ejemplo, en la suma, no se van a sumar dos float, si no que se van a sumar dos expresiones aritméticas.

He hecho, en el tipo que creamos para "instrucción", vamos a usar este TAD, y podemos ver que la instrucción "repetir" va a llevar una expresión aritmética y una instrucción, por lo que también va a ser recursivo este tipo. Inst puede ser secuencia de instrucciones.

Para implementar eso, utilizaremos el módulo Graphics en Ocaml, para abrir la ventana, dibujar las líneas (ángulo, posición, etc).

Para abrir este módulo, es necesario poner:

```
open Graphics;;
#load "graphics.cma";;
```

Para cada ' α ' en Ocaml, existe un tipo de variable que permite guardar valores de ese tipo.

```
 $\alpha$  ref : int  $\rightarrow$  int ref.
```

Necesitamos una operación para obtener el valor y otra para modificarlo:

```
(!) :  $\alpha$  ref  $\rightarrow$   $\alpha$ 
```

```
(:=) :  $\alpha$  ref  $\rightarrow$   $\alpha \rightarrow$  unit
```

```
ref :  $\alpha \rightarrow \alpha$  ref
```

No tiene que devolver ningún valor. Efecto colateral fuerte.

Con este nuevo concepto, podemos realizar el programa del "turno" propuesto unas páginas anteriores.

```
let turno() =  
  i := !i + 1;  
  !i;;
```

} muestra el valor ya incrementado, por lo que si inicializáramos $i = \text{ref } \emptyset$, nunca se mostraría el \emptyset .

```
let turno() =  
  let r = !i in  
    i := !i + 1;  
  r;;
```

} Esta función, en cambio, muestra el valor actual de i , y luego lo incrementa. Aquí sí que se mostraría el \emptyset .

```
let c = ref  $\emptyset$ ;,  
  -: int ref { contents = 0 }
```

```
!c;;
```

```
-: int =  $\emptyset$ 
```

```
(:=) c := ;;
```

```
!c;;
```

```
-: int = 1
```

```
i := 2;;
```

```
!c;;
```

```
-: int = 2
```

Podríamos también definir una función de reseteo, que simplemente tendría que poner el valor de i a \emptyset .

```
let reset () =  
  i :=  $\emptyset$ ;;
```

PROGRAMACIÓN IMPERATIVA, BUCLES.

Lo que se va a dar a continuación, viene en el manual, en el capítulo de expresiones (6.7), que se corresponde a la parte 2 \Rightarrow Part II. The Ocaml language.

BUCLE WHILE

```
white <bool> do <e> done;;
```

Si la expresión booleana da como resultado false, se devuelve unit. Si da verdadera, true, se pasa a evaluar $<e>$.

En caso de que la expresión booleana no dependa de nada, nos encontramos ante un bucle vacío o un bucle infinito.

Lo que realmente importa es el cambio que realiza $<e>$.

BUCLE FOR

```
for <i> = <e1> to <e2> do <es> done;;
```

<e1> } Son dos valores de tipo int. Se comienza
<e2> } evaluándolos en orden creciente (ambos inclusive).

let <i> = V in <es>

En vez del <to>, podemos poner <downto>,
pasando a evaluarse así de forma decreciente.

O

Hagamos ahora las funciones fib y fact, que
ya hemos visto, pero ahora usando programación
imperativa.

```
let fact n =  
  let f = ref 1 in  
    for i=1 to n do  
      f := !f * i  
    done;  
  !f;;
```

!f es el valor de f.

La i no es ninguna variable de referencia, por eso
no lleva ninguna admiración.

```

let fib n =
[ let i = ref  $\emptyset$ 
  and f = ref  $\emptyset$ 
  and u = ref 1 in ]
  while (!i < n) do
    i := !i + 1;
    f := !f + !u;
    u := !f - !u;
  done;
!f;;

```

Con ref se crean,
se inicializan las variables.

ELENA
DELAMANO
FREIJUE

FUNCIÓN SWAP

swap : ('a ref * 'a ref) \rightarrow unit.

Intercambia los valores entre dos variables de referencia.

```

let swap (v1, v2) =
  let a = !v1 in
    v1 := !v2;
    v2 := a;;

```

Hacemos igual que en Pascal: almacenamos en valor en una variable auxiliar, de manera que no nos la carguemos al intentar intercambiarla.

CUIDADO !

```
let x = ref(); → int ref = {contents = ()}
```

```
| x(); ⇒ ()
```

```
let z = x();
```

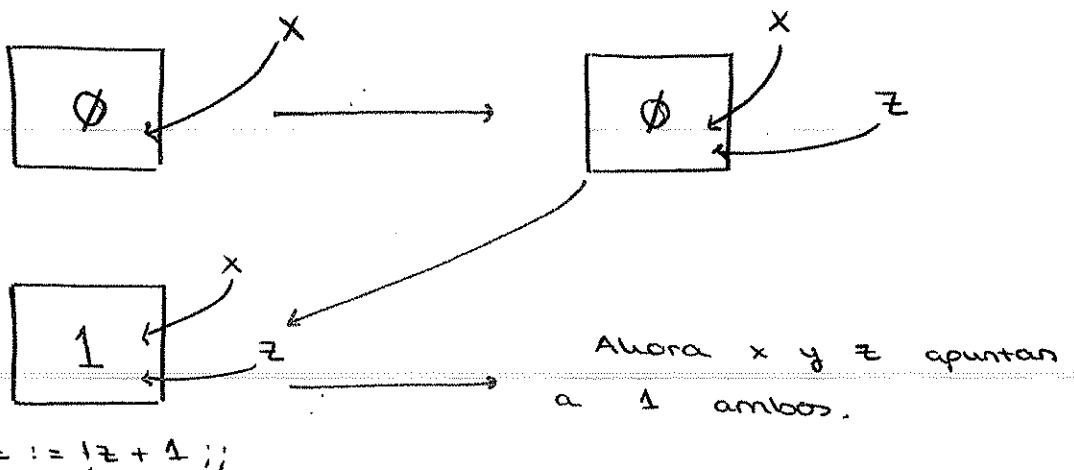
```
| z(); ⇒ int = ()
```

```
z := | z + 1();
```

```
| z(); ⇒ int = 1
```

```
| x(); ⇒ int = 1
```

Hay que tener cuidado. Ref crea algo similar a un puntero, de manera que si se le cambia el valor a z, el de x también cambia.



Para el estado de la tortuga (LOGO), necesitamos una tupla que contenga sus características, como pueden ser: las coordenadas, la posición del lápiz (levantado o no), ...).

TIPO REGISTRO

Equivalente al RECORD en Pascal.

```
type persona = { nombre : String; apellido : String; edad : int }
```

El nombre de los campos debe ir con minúscula.

```
let p = { apellido = "Molinelli";  
edad = 52;  
nombre = "José M." };
```

```
let nombre : { apellido = _; edad = _; nombre = _ } = p;  
  
nombre p;;  
_: string = "José M."
```

```
let nombre : { nombre = _ } = p;;  
  
p.nombre;;
```

Vemos que podemos obtener el valor de uno de los campos del registro de varias maneras diferentes. Aunque quizás, la más cómoda sea la última.

Con el funcionamiento y características que estamos viendo para los arrays, también funcionan los strings; ya que, realmente, son arrays de caracteres. De manera que:

```
let s = "Hola" ;;
s.[i]; → sirve para obtener el dato de la posición i.
s.[∅]; → 'h'
s.[∅] ← 'K' → se sustituye el carácter [∅] por 'K'.
|
→ Ahora, s = "Hola".
```

LOGO

```
make "i (2+1)
print :i
print (:i + 3)
print (:0 + 3) → variable no asignada.
make "i (i + 1)
print :i → 4
make "i 10
while [:i > ∅] [ordenes].
```

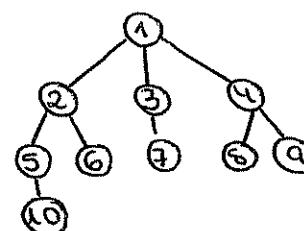
EJERCICIOS PROPUESTOS PARA TGA

- Métodos de ordenación con vectores.
- Función anchura, ' $\text{a gtree} \rightarrow \text{'a list}$

Demuestra la lista de nodos ordenados según niveles, de un ' a gtree '. (cualquier número de aristas).

type ' a gtree ' =
 $\text{gt of } \text{'a} * \text{'a gtree list}$

De manera que:

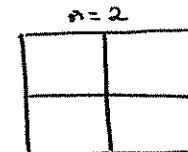


Demuestra:

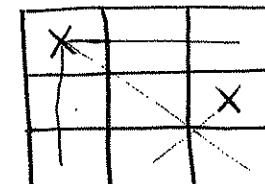
1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

- Función de los reyes., que en un tablero ($n \times n$) de $n \times n$, demuestra las posiciones donde colocar n reinas sin que se coman una a otras.

$n=1$
X



No hay
solución



No
solución

- Es una función de tipo: $\text{int} \rightarrow (\text{int} * \text{int}) \text{ list}$
- En caso de que no haya solución, se lanza una excepción.

OPERACIONES DE ENTRADA / SALIDA

Para representar canales de entrada y salida, en el módulo `Pervasives` tenemos dos tipos de datos:

- in-channel.
- out-channel.

Tenemos que tener presente el funcionamiento del redireccionamiento de ficheros (entrada/salida) en el shell:

```
$ ./miproq < fichero_entrada  
$ ./miproq > fichero_salida.
```

Ahora expandiremos diferentes funciones relacionadas con estos tipos.

- `print-char` → es la función más básica. Imprime por pantalla un sólo carácter.
- `print-string` → envía los strings, sin salto de linea.
- `print-endline` → envía los caracteres del string + '`\n`', es decir, más el salto de linea.
- `print-int` → envía los caracteres, no el número.
- `print-float` → (dem que `print-int`

Cualquiera de esas funciones puede ser implementada a partir de la función "print-char".

```
let print-endline s =  
    print-string (s^"\n");;
```

Si ejecutamos esta función en el compilador interactivo, tenemos que tener en cuenta que se comparte la salida de nuestro código con la del compilador.

```
let s = "Hola";;  
print-string s;;  
Hola -: unit()  
print-endline s;;  
Hola  
-: unit()
```

Ejemplos de ejecución.
Diferencia entre
print-string y
print-endline.

La función de print-endline que hemos definido es igual a la del módulo Pervasives.

Cuando un programa lee o escribe en un canal de entrada o salida, estos accesos no se realizan de modo directo, si no que se realiza a través de un espacio temporal y es el sistema operativo el que, cuando lo cree conveniente, las vuela físicamente sobre ese dispositivo de salida, todo para optimizar el rendimiento de la máquina. Esta optimización está pensada porque las operaciones entre entrada-salida interrumpen bastante a la máquina.

Si un programa está constantemente realizando operaciones de entrada/salida, eso va a perjudicar mucho el rendimiento general. La gestión realizada por el sistema operativo es transparente para el programa.

Este es relevante en el sentido de que, cuando se manda imprimir, por ejemplo, un string con la función "print_string", no tiene por qué aparecer en ese momento. Aparecerá, realmente, cuando el sistema operativo le mande la señal de volcarse. Si está ocupado con otras cosas, puede tardar bastante. Pero, hay momentos en los que necesitamos que sea en ese mismo instante. Esto es posible, forzando a que todo aquello que está pendiente sea mandado al dispositivo; es decir, forzando a que vuelque el buffer del SO.

Un ejemplo es un programa que le hace una consulta al operador. Si el operador no responde, el programa no puede continuar. Hay que enviar una señal de "AHORA". Se le dice que vuele el buffer de la salida estándar.

El "print-endline" del módulo `Pervasives` tiene esta característica, mientras que la nuestra no. Por lo tanto, la afirmación que hemos hecho anteriormente de que ambas funciones, aparentemente, son iguales, es incorrecta.

- `print-newline` → envía un salto de línea con vueltas de buffer. Podríamos definirlo como el `print-endline` del string vacío.

LECTURA DE LA ENTRADA ESTÁNDAR (TECLADO)

- Lectura básica, leemos un carácter (un byte) hasta que se dé el retorno de carga, el proceso no estará listo. (caracteres no listos ni disponibles hasta darte a enter).
- `read-line` → es una aplicación sobre `unit`, que nos devuelve el string que se haya introducido hasta el primer salto de línea (o hasta que se termine la entrada estándar).
- `read-int` → lee y aplica "int-of-string".
- `read-float` → lee y aplica "float-of-string".

Estas dos últimas funciones no soy muy útiles, ya que tienen una probabilidad alta de fallos.

OPERACIONES DE LA SALIDA ESTÁNDAR

Valor de tipo out-channel.

Eso nos lo da "open-out". Se le pasa un string y nos devuelve tipo out-channel, que estará asociado al archivo. Es decir, open-out" crea un archivo en el disco con el nombre que le digamos, para escribir en él. Si ya existe, se "machaca" al existente. En caso de no tener permiso, se produce un error (una excepción): "Sys.error".

Cuando se acaba de escribir, es necesario cerrar el canal con la orden close-out. De todas maneras, los procesos, cuando terminan, cierran los canales, por precaución.

Para escribir, enviamos un byte → output-byte, se le indica el canal, int (mod 256). (Básico) Todos los demás casos a partir de este, como era el caso de los print.

En estos canales hay como una especie de puntero que permite volver hacia atrás del el archivo; pudiendo borrar algo. La función para esto es seek-out.

Si queremos saber la posición → pos-out

Para mandar la señal de "AHORA" de la que hemos hablado antes, utilizamos flush stdout. Con esta función podemos arreglar la función que hemos implementado antes de print-endline.

Si queremos volcar el buffer de todos los procesos, debemos usar "flush-all"
output-binary-int.

ENTRADA

- open-in → abre un canal de entrada asociado a lo que signifique para el sistema operativo. En caso de que no exista, o de que no tengamos permisos, se activa la excepción "Sys.error". Coloca un puntero al principio del fichero, para comenzar a leer.
- close-in → no es tan importante el cerrarlo como es en la salida, porque estamos leyendo, no escribiendo ni modificando nada; pero debemos cerrarlo igual.
- input-byte → lee un valor y lo devuelve como int.
- input-char → idem, pero lo devuelve como char.
- input-line → lee como string desde el char en el que se está posicionado, hasta el primer salto de linea.
- input-binary-int.
- seek-in
- pos-in
- input-value → lee valores de cualquier tipo.
- stdin.

ELENA
DELAMANO
FREIJE

- `read-line ()`; → queda en espera de la entrada, hasta que se le dé al enter.

`let s = open-out "salida";;` → Crea un archivo de \emptyset bytes.

`output s "Hola";;` → Sigue \emptyset bytes, ¿porque -val int → int → unit() no se ha mandado todavía?

`flush s;;` → Sigue con \emptyset , y con `cat`, la salida sigue sin ser nada.

`close-out s;;` → tampoco se trata de eso.

El fallo está en que NO se trataba de la función de `output`. Por lo que volvemos a empezar.

`let s = open-out "salida";;`
`output-string s "Hola";;`
`-: unit.();`

`flush s;;` → Hasta que mandamos la señal, el sistema Operativo no lo escribe.

`output-byte s 65;;` → Añade 'A' al fichero de salida, porque $165 = 'A'$.
`flush;;`
`"HolaA".`

`output-binary-int s 100;;` → se añaden 4 bytes
`flush;;` (de un entero). La codificación de 4 coincide con 'd'.
`"HolaAd".`

Tenemos una lista de strings, y queremos hacer una lectura de todos ellos juntos. Para ello tenemos que añadir saltos de linea.
Tenemos que coger la lista de strings y los escribe en un canal de salida.

```
let rec output_string-list = function
  [] → unit
  | h::t → output_string h ^ "\n";
            output_string-list t;;
```

tipo: out-channel → string list → unit.

```
let l = ["Uno"; "Dos"; ""; "Cuatro"; "5"];;
let s = open_out "lista";;
output_string-list s l;;
close_out s;;
```

Si ahora miramos con cat el contenido de la lista:



```
let e = open-in "lista";;
input-byte e;;
:- int = 117 → valor del código ASCII de la 'u'.
input-line e;;
:- string = "no"
input-char e;;
:- char = 'o';
input-binary-int e;;
:- int = -277673462.
```

Hacer `input-string-list` \Rightarrow `input-channel` \rightarrow `string list`,
que es la recíproca de la que acabamos de hacer.

¡OJO! Hay que tener presente el orden en el
que se hacen los casos.

PROGRAMACIÓN ORIENTADA A OBJETOS

CLASE

Es la que define cómo es, cómo se crea y cómo se comporta el objeto.

Es el modelo a partir del cual creamos las instancias del objeto.

Define:

① Variablos de instancia (atributos). Contienen los datos asociados a esa instancia y representan sus propiedades / atributos.

② Métodos, son operaciones de manipulación de dichos datos asociados.

OBJETO

No es más que una instancia de la clase, es decir, un caso concreto de ella (una concreción de la misma).

SINTAXIS

Declaramos la clase, con la palabra clave "class"
y junto a la definición de parámetros del constructor
de la clase. No se indica el tipo de manera
obligatoria. Los parámetros pueden ser puestos en
curry o de cualquier manera.

```
class    nombre-clase  p1...pn =  
  
Object  [ alias ]  
  
[  
  val [mutable] nombre-var1 = expr1  
  :  
  :  
  val [mutable] nombre-varv = exprv  
  :  
  
[  
  method nombre-metodo1  pm1...pmn = expr1  
  :  
  :  
  method nombre-metodom  pm1...pmn = exprm  
  
]  
end ;;
```

[mutable] → opcional. Poder variar su valor. Si no, sería
constante.

Las expresiones de val, inicializan su valor,
sin separadores. Es obligatorio.

(alias) → equivalente al this de java. Por lo
general, se usa this o self.

① Los atributos son siempre privados, es decir, no son visibles fuera de la clase. Si queremos que así sea, tenemos que definir métodos getters y setters.

② Debemos tener cuidado con la notación de manejo de los atributos (!, :=, ref).

③ Cuando no hay argumentos en un método, por convenio se tiene que:

si es un getter, no se le pasa nada;
pero si no es un getter, hay que forzarlo pasándole `unit '()'`.

class point (`x-init, y-init`) =

Object

val mutable `x` = `x-init`

val mutable `y` = `y-init`

method `get x` = `x`

method `get y` = `y`

method `moveto (a,b)` = `x ← a ; y ← b`

method `rmoveto (dx,dy)` = `x ← x + dx ;
y ← y + dy`

method `toString ()` = `" (" ^ (string-of-int x) ^
", " ^ (string-of-int y) ^ ")"`.

`end;`

ELENA
DELAMANO
FREIJE

El tipo del objeto, aquí, no es point.

```
let o1 = new point (0,1);  
val o1 : point = <obj>
```

No es de tipo point, ya que no es nominal, sino estructural.

Realmente, el tipo de o1 es:

```
get-x : int  
get-y : int  
moveTo : int* int → unit  
rmveto: int* int → unit  
toString : unit → string.
```

El tipo se define por las características de la clase, no por el nombre que se le asigna; sólo que en Ocaml se pone este nombre como abreviatura de todos esos caract.

TIPO DEL OBJETO

Es la lista de los métodos y sus respectivos tipos. Los atributos no.

Ocaml es un lenguaje multiparadigma.

El tipo no es la clase.

Para crear instancias de una clase:

```
let p1 = new nombre-clase p1...pn
```

Teniendo dos clases diferentes que hagan cosas diferentes pero que tengan el mismo tipo, lo que se hace es instanciarlas y distinguirlas en base a esa instancia. Sin embargo, al tener el mismo tipo, se podrían almacenar en una lista (a pesar de ser dos clases diferentes).

No existe el concepto de constructor, porque de eso ya se encarga la propia clase.

Podemos crear una función que llame a la clase, es decir, que automáticamente cree instancias sin andar teniendo que poner "new" seguido.

```
let make-point x = new point (x,x);;
let p2 = make-point 3;;
```

Para llamar a los métodos, utilizamos "#":

idobjeto # nombre-metodo p1...pn

Como por ejemplo:

p2# to-string();

El tipo de funciones que se definen para la creación de objetos, como es el caso de 'make-point', se llaman funciones factoría.

Una vez tenemos una instancia del objeto, podemos manipular dicha instancia como cualquier otra definición del lenguaje. Por ejemplo: podemos almacenarla en listas, compararlo, etc. Es igual que cualquier otra "variable".

Respecto a la comparación de objetos, hay algo que tener en cuenta: un objeto puede ser igual a otro física o estructuralmente.

Para la igualdad física se utiliza el operador (=), pero compara las direcciones de memoria.

Este difiere del caso general, donde los operadores (`=`) y (`==`) se refieren a la igualdad estructural, mientras que los operadores (`==`) y (`!=`) se referían a la igualdad física.

En caso de querer una igualdad estructural entre dos objetos, tenemos que definir un método "equales" que haga esta función, al igual que en Java.

```
let p3 = new point(3,3);
p3# to-String() // → "(3,3)"
p2# to-String() // → "(3,3)"
p2 = p3; // → false
```

No misma dirección
de memoria.

RELACIONES

① AGREGACIÓN

Cuando alguno de los atributos almacena datos, ya sea individualmente o en grupo, como

val x = _____

o

val a = [1____,____,____]

```

class arista1 (p1,p2) =
object
    val vertice-A = p1
    val vertice-B = p2
end;;

```

```

class arista2 (p1,p2) =
object
    val vertices = (p1,p2)
end;;

```

→ Getters con second
y first (fst y snd)

```

class aristas3 (p1,p2) =
object
    val vertices = [1 p1 ; p2]
end;;

```

→ con array.

```

class arista4 (p1,p2) =
object
    val vertices = [p1 ; p2]
end;;

```

→ con lista.

El problema que podemos encontrarle a estas definiciones, es que cambia el tipo. Así se nos permitirá cualquier cosa en el par, por lo que será un par '`('a,'b)'`, porque el motor de inferencia de tipos de Ocaml no es capaz de detectar nada que le indique qué tipo es o si son del mismo.

Dice que ambos son del mismo tipo 'a' sólo en caso del array o de la lista. El problema viene dado en el par. Veremos esto con más calma en la página: 44

El otro tipo de relación es

② HERENCIA

La subclase tendrá ("heredará") los mismos atributos o variables de instancia, y métodos que la superclase, pudiendo:

- Añadir nuevos atributos.
- Modificar atributos o métodos ya existentes.

Para utilizar la herencia:

```
class nombre-subclase pt..pn =  
object  
    inherit nombre-superclase pp1..ppn as alias  
  
    val ...  
    method ...  
  
end;;
```

Para el alias, habitualmente, se usa super.

En cuanto a añadir atributos y métodos, no hay problema. Dónde sí que nos los podemos encontrar, es al modificarlos. Cuando modificamos, tenemos que tener presentes ciertas cosas:

- Tenemos que respetar el tipo original.
- En el caso de los métodos, podemos acceder a los de la superclase sin problema, ya que siguen siendo accesibles vía el "super" (o el alias que se le haya dado).
- En el caso de los atributos, no es igual. No son accesibles de NINGUNA manera, incluido los getters.

¿Cómo funciona el mecanismo de herencia?

La "regla de oro":

- ① Para cualquier atributo o método, la última definición prevalece.
- ② La herencia funciona como inclusión textual: cuando se tengan dudas de con qué herencia se quedó uno, se coge el código de la superclase y se hace un "replace" en el inherit, es decir, en vez de poner el inherit, copiamos y pegamos el código de la superclase, y luego le aplicamos el paso ^, lo de la prevalencia.

Cuando definimos las diferentes clases de arista, hablábamos del problema que nos encontrábamos en el par en cuanto al tipo de los dos elementos. Para definirlo de un tipo concreto, lo que tenemos que hacer es:

```
class arista ((p1,p2) : (point * point)) =  
object
```

...

Definamos ahora una nueva clase de punto, esta vez añadiéndole un atributo que represente su color.

```
class colored-point (x-init, y-init) c-init =  
object (this)  
    inherit point (x-init, y-init)  
    val mutable color = c-init  
    method get-c = color  
    method set-c c = color ← c  
    method to-string () = "copiar el método de  
        point y añadir "["^ color ^"]" o  
        this# get-c.  
end;;
```

ELENA
DELAMANO
FREIJE

```
let p1 = new point (0,1);  
let p2 = new point (0,2);  
let c1 = new colored-point (0,3) "azul";  
p1#to-string() // →  
p2#to-string() // →  
c1#to-string() // →
```

Se podría crear una lista con p1 y p2, porque son del mismo tipo; pero no podemos meterlos con c1, porque mucho que sea un subtipo.

Otra posibilidad para la implementación de la clase sería:

Obrégale un alias a la superclase, como por ejemplo, "super", y definir el método to-string así:

```
method to-string() = super#tostring() + "[" + this#get-c + "]";
```

Cuando se quiera referir a un método de la propia clase, es obligatorio usar "this#" (o el alias que se le haya puesto a la clase), si no, ocaml lo interpreta como un valor en vez de como un método.

107
108
109

REFERENCIAS (THIS / SELF / SUPER)

- No es posible llamar a tus propios métodos, a menos que utilices un alias (this / self).
- En ocasiones, también se incluye un identificador de tipo:

```
class nombre-clase pt..pn =  
object (self : 'self')
```

Referencia a sí mismo y a "su propio tipo".

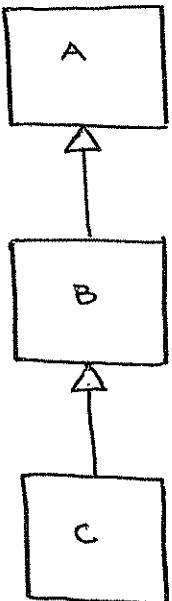
- Para referenciar al padre, necesitamos definir un alias de la superclase para poder acceder a los métodos del padre. Si se redefinen los atributos, estos se pierden.

DELAYED BINDING

Que en otros manuales puede aparecer como "Dynamic Binding" o "Dynamic look-up".

En Ocaml, el método, código o expresión, concreto a llamar es determinado en tiempo de ejecución. No es definido a priori.

Por ejemplo, teniendo un objeto `a` de tipo `c`:



El método `to_string()` está en la clase A, por lo que si hacemos `c1#to_string()`, se va buscando en la implementación de la superclase inmediata de manera sucesiva.

Supongamos ahora que el código del método `to_string()` definido en A llama a un método `get_id()`, pero que éste está definido (redefinido) en las 3 clases.

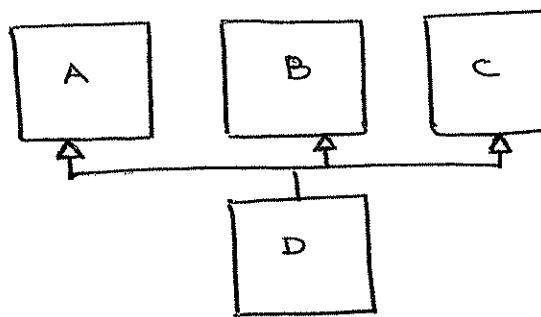
Llamamos, pues, a `c1#to_string()`, sube, como antes, hasta encontrar el respectivo método en la clase A, llama al código y vuelve a la clase concreta (la del objeto: C) a buscar la última definición (redefinición) que se ha hecho de `get_id()`, usando, por lo tanto, la de la propia clase C.

De suponer que `get_id()` sólo estuviese en A y en B, cuando se vuelve a la clase concreta, se seguirán las pautas comentadas al principio, subiendo a las superclases inmediatas sucesivamente hasta encontrar el método, por lo que se usaría el código de `get_id()` de la clase B.

HERENCIA MÚLTIPLE

Se permite heredar simultáneamente de varias clases, cosa que no está disponible en Java.

PROBLEMA 1: ¿Qué sucede cuando hay métodos o atributos con el mismo nombre entre padres? ¿Cuál es el que se hereda?



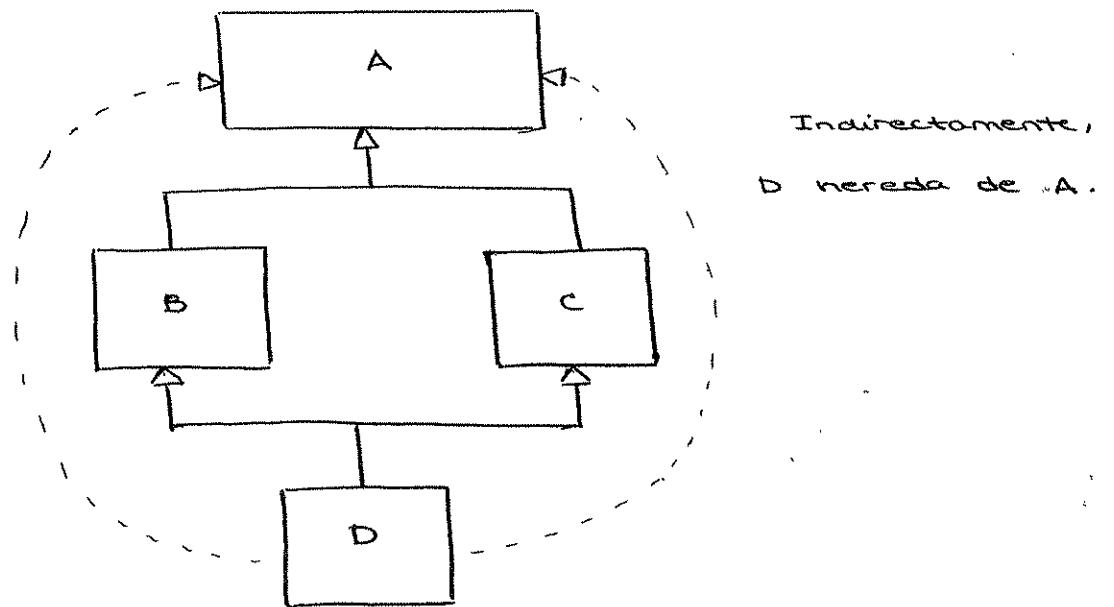
Clase A → val c
método get-x()
Clase B → método get-x()
Clase C → val c.

- Deben coincidir en tipo.
- Se hereda de la "última superclase" en el inherit (coincide con el concepto de inclusión textual).

```
class nombre-clase p1..pn =  
object  
inherits A, B, C  
:  
end;;
```

Por lo que D tendrá el atributo c de la clase C y el método get-x() de la clase B.

PROBLEMA 2 : ¿Qué pasa si hay una misma herencia "repetida"?



Esto sucede cuando varios padres tienen antepasados comunes. Se aplica la misma solución que se ha propuesto para el problema 4.

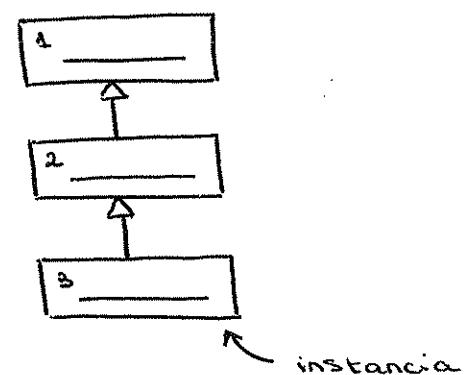
Seguiremos viendo la herencia múltiple más adelante, incluyendo ejemplos con su respectivo código.

INICIALIZACIÓN

```
class nombre-clase pc1 ... pcn =  
object  
    initializer expr  
    :  
end //
```

El "initializer" es similar al constructor:

- ① Se ejecuta una sola vez, automáticamente durante la creación del objeto.
- ② Es lo primero que se evalúa inmediatamente después de que haya sido creado. Esto, a su vez, implica:
- ③ Que puede emplear tanto métodos como atributos del objeto.
- ④ Si existe una jerarquía, se evaluarán todos los initializers desde la superclase más general hacia abajo.



Otra alternativa es usar "let ... in" en la definición de la clase. Pero, sin embargo, éste se ejecutaría justo antes de la creación del objeto, por lo que no se pueden acceder a / no se puede hacer uso de los métodos y atributos del objeto.

```
class verbose-point p =  
object (self)  
  inherit point p  
  
  initializer  
    let xm = string-of-int x  
    and ym = string-of-int y  
    in Printf.printf "» Creando punto en  
    coordenadas (%.s , %.s)\n" xm ym;  
    Printf.printf "a distancia %.f del origen\n"  
    (self#distance ());  
  
  end;;
```

distance () es un método añadido a point.

```
let v1 = new verbose-point (0,1);;  
» Creando punto en coordenadas (0,1) a  
distancia 1 del origen.
```

Ya lo muestra en la inicialización o creación del objeto (con el initializer).

Queremos crear una clase que recibe como parámetro un número n y que comprueba si es mayor que ϕ , dando lugar a una excepción en caso de que no se cumpla.

Vamos a poner 4 opciones de clase, y luego analizaremos si son correctas las implementaciones o no.

```
class ClaseClaval n =  
    if (n ≤ φ) then  
        raise (Failure "Menor o igual a φ")  
    else  
        object  
            :  
        end;;
```

```
class ClaseClaval n =  
    object  
        initializer  
            if (n ≤ φ) then  
                raise (Failure "Menor o igual a φ")  
            :  
        end;;
```

- ClaseClaval → Error de sintaxis, dado que la parte de "Object → end" no es una expresión a evaluar.
- ClaseClaval → Ineficiente, crea el objeto igualmente aunque sea una excepción. Objeto desechable. Tampoco es válida.

```

class Clasecilla3 n =
    let _ = if (n ≤ Ø) then
        raise -----
    else ()
in
object
    :
end;;

```

```

class Clasecilla4 n =
object
    val atributo = if (n ≤ Ø) then
        raise -----
    else
    n.
    :
end;;

```

- Clasecilla3 → es correcta, sintácticamente válida.
- Clasecilla4 → también es válida, se realiza la comprobación durante la inicialización del objeto.

-
- O
- ① ERROR SINTÁCTICO.
 - ② INEFICIENTE → si 'n' va a ser utilizado en la posteridad, puede llegar a ser peligroso.
 - ③ CORRECTO.
 - ④ VÁLIDO.

VISIBILIDAD

Tanto de atributos como de métodos.

- Atributos → "privada", es decir, sólo se puede acceder desde el propio objeto.

- Métodos → por defecto, es "pública", pero se puede hacer privada mediante:

method private nombre-metodo p1...pn = expr.

IMPORTANTE → este "private" no se corresponde con el de Java, sino que es más similar al "protected", debido a:

- ① Se puede usar desde dentro de la clase.
- ② Se puede usar también desde sus subclases.

Si se intenta llamar a un método privado desde fuera, ocam nos dirá que no existe.

Ahora veremos lo que en Java llamamos "abstract".

ELENA
DELAMANO
FREIJÉ

CLASES, MÉTODOS Y ATRIBUTOS VIRTUALES

MÉTODO ABSTRACTO: Aquel que no está implementado, del que solo declaramos su interfaz

Sintaxis:

```
method virtual nombre-metodo pl...: tipo
```

ATRIBUTO ABSTRACTO: Aquel que no está inicializado, sino que solamente está declarado su tipo. Sintaxis:

```
val [mutable] virtual nombre-atributo : tipo
```

CLASES ABSTRACTAS: Son aquellas que tienen algún método o atributo abstracto, y hay que declararlas como tal.

Sintaxis:

```
class virtual nombre-clase pl...pn =  
object  
:  
:  
end;;
```

Al declarar un método abstracto no se ponen los parámetros, ya que se implican al indicar el tipo:
int → int → char → string donde los parámetros son los subrayados.

Una clase virtual no se puede instanciar directamente.

con "initializer" se puede acceder a los atributos y métodos de la clase, mientras que con la "inicialización normal" de un atributo no se puede.

○

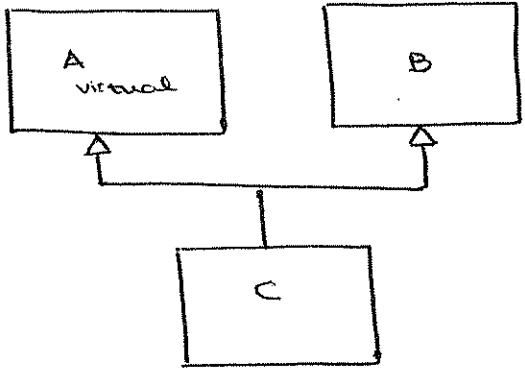
CÓMO COMBINAR LAS CLASES ABSTRACTAS Y LA HERENCIA MÚLTIPLE?

PROBLEMA 1 → teníamos métodos y atributos de igual nombre en varias clases de manera simultánea.

PROBLEMA 2 → si hacía herencia "repetida" (inclusión textual).

PROBLEMA 3 → ¿y si hay superclases virtuales?

Para solucionar estos problemas, se pueden considerar dos métodos.



CASO 1:

Solucionarlo "como siempre".
se encargaría la subclase de implementar los métodos o atributos virtuales.

Para observar mejor este caso, vamos a poner un ejemplo más concreto.

- A tiene un método virtual `A1`.
- C tendría que redefinirlo, implementarlo.
- B tiene un método virtual `B1`.
- C tendría que redefinirlo, implementarlo.

Por lo que C tendría que implementar ambos métodos `A1` y `B1`.

CASO 2:

Lo llamado un "Mixin". Otra de las superclases es la que implementa de forma indirecta, vía herencia, los elementos virtuales.

Ahora:

A tiene un método virtual A1: tipoAA.

B implementa A1, método virtual A: tipo AB.

B implementa el método de A, por lo que no hace falta que C redefina el mismo; B ya le está dando su implementación.

MANEJO DEL POLIMORFISMO A NIVEL DE OBJETO.

Para ello comentaremos hablando de las clases parametrizadas:

OBJETIVO:

Permitir contenidos polimórficos y su manejo.

Puede verse como una "evolución" de los tipos parametrizados.

Debe de quedarnos muy claro que en el momento en el que se cree una instancia de ese objeto, esa instancia ya quedará ligada al tipo concreto que se le vaya pasado en la creación!

SINTAXIS

```
class ['a,'b,...] nombre-clase pl...pn =  
object  
:  
:  
end;;
```

Los paréntesis son "reales", obligatorios, no implican opcionalidad.

Un nombre de tipo polimórfico para cada parámetro que queramos.

```
class por x0 y0 =  
object  
val x = x0  
val y = y0  
method fst = x  
method snd = y  
end;;
```

Clase
erronea.

Esta clase no es válida, porque nos da un error al no ser capaz de identificar de qué tipo son los parámetros. → "The method fst has type 'a where 'a is unbound".

Para solucionar este error, simplemente hay que:

- indicar los parámetros de clase (clase parametrizada, ['a,'b]).
- ligar dichos tipos a los elementos correspondientes para que el motor de inferencia de tipos de Ocaml.

Quedaría, pues:

```
class ['a,'b] par (x0:'a)(y0:'b) =  
object  
    val x = x0  
    val y = y0  
  
    method fst = x  
    method snd = y  
  
end;;
```

En caso de querer forzar a que los dos parámetros sean del mismo tipo 'a', simplemente habría que cambiar la cabecera por:

```
class ['a] par (x0:'a)(y0:'a) =
```

Ahora pasamos los atributos a mutable:

```
class ['a,'b] par (x0:'a) (y0:'b) =  
object  
    val mutable x = x0  
    val mutable y = y0  
  
    method fst = x  
    method snd = y  
  
    method set-fst x1 = x <- x1  
    method set-snd y1 = y <- y1  
  
end;;
```

```
let p1 = new par "hola" 1;;  
p1#fst;;  
- : string = "hola"  
p1#snd;;  
- : int = 1  
p1#set-fst "adios";;  
p1#fst;;  
- : string = "adios"  
p1#set-snd 2;;  
p1#snd;;  
- : int = 2  
p1#set-fst 545;;  
Error.
```

Esa última instrucción da error debido a que el objeto ya está instanciado y, por lo tanto, los tipos ya están fijados.

Como ya hemos dicho, los tipos polimórficos quedarán instanciados 'para siempre' para ese objeto a su tipo inicial. Si creamos otro objeto, no pasa nada, puede tener tipos diferentes.

```
let p2 = new par 3'14 'd';
p2#fst;;
-: float = 3'14
p2#snd;;
-: char = 'd'.
```

¿Qué sucederá si ahora le incluimos la herencia?

CLASES PARAMETRIZADAS Y HERENCIA

OPCIÓN A:

Podemos mantener su carácter polimórfico (por ejemplo, si solo se quieren añadir métodos o atributos de un tipo concreto).

ELENA
DELAMANO
FREIJJE

```

class ppair xx yy =
object
inherit par xx yy
method to-pair() = (x,y)
end;;

```

ERROR!

Da error debido a que el inherit par necesita dos argumentos de tipo y no se le están pasando. En caso de hacerse de forma suelta o separada no hace falta, pero al ser con herencia si. Tendría entonces que quedar:

```

class ['c,'d] ppar (xx:'c)(yy:'d) =
object
inherit ['c,'d] par xx yy
method to-pair() = (x,y)
end;;

```

En el inherit se le indica cual va a ser la 'a ('c) y cual va a ser la 'b ('d). Luego ya se encarga él de ligarla.

Opción 2:

Especializar la clase completamente a unos tipos concretos, es decir, que deje de ser polimórfica.

Queremos que `par` sea sólo para enteros y floats, por lo que vamos a crear una subclase que sirva sólo para esos tipos:

```
class int-float-par <> yy =  
object  
    inherit [int, float] par <> yy  
end;;
```

Opción 3:

Especializar la clase parcialmente, concretizar alguno de los tipos polimórficos, pero no todos. Por ejemplo:

```
class ['a] int-x-par <> (yy:'a) =  
object  
    inherit [int,'a] par <> yy  
end;;
```

METODOS POLIMORFICOS

method nombre : tipo1 tipo2 ... tipos. tipo del método

Tipo del método es de la manera: $'a \rightarrow 'b \rightarrow 'c$

Tipos abiertos \rightarrow let $x = x\#get_x \Rightarrow x$ no tiene un tipo concreto
va a ser cualquier clase que lleve un método llamado `get`, y devolverá un $'a$. El motor de tipos escogerá el más general

↳ sigue pas siguiente

EJERCICIOS PROPUESTOS PARA TGR

- ① ¿cómo modificar la clase de la práctica 9, vector-dinámico, para hacerla parametrizable?
- ② ¿cómo extender las dichas clases para trabajar con enteros?

Una cosa es que las clases sean parametrizadas con contenidos polimórficos, y otra es que sean métodos polimórficos.

ERROR

```
class classeilla =  
object  
method create-pair (x:'a)(y:'b) = (x,y)  
end;;
```

Da error al no saber qué tipo aceptar.

```
class ['a,'b] classeilla =  
.object  
method create-pair (x:'a)(y:'b) = (x,y)  
end;;
```

* tipos abiertos → class couple a0 b0 =
object val a = a0
val b = b0
method to-string = a#to-string ^ b#to-string

solo sabemos que tiene un método to-string, sabemos que solo un string, pero no sabemos si es un string

```

let c1 = new clasecilla();
c1# create-pair 1 2;;
c1# create-pair "Hola" "mundo";;
ERROR.

```

Da error porque la instancia ya ha definido el tipo, se han instanciado los tipos tipificándose a int.

¿Cómo podemos conservar ese polimorfismo?
Dejaré de ser una clase parametrizada:

```

class clasecilla =
object
  method create-pair : 'a*'b . 'a+`b -> 'a+`b = fun x y => (x,y)
end;;

```

```

let c2 = new clasecilla;;
c2# create-pair 1 2;;
  -: int * int = (1,2)
c2# create-pair "Hola" "mundo";;
  -: string * string = ("Hola", "mundo")
c2# create-pair 1 "hola";;
  -: int * string = (1, "hola")

```

Mientras que las clases parametrizadas quedan ligadas al primer tipo que se les pasa, los métodos no.

OPENTYPES

Tipos abiertos, concepto que sólo se aplica a las clases objetos.

Ocaml tiene tipado estático, es decir, antes de evaluar una expresión, se comprueba su tipo. Si este tipo es correcto, es cuando compila, si no, da error; sin embargo, los tipos que hemos visto hasta ahora son tipos cerrados.

Los tipos abiertos son mucho más permisivos:

El tipo asociado a la clase a la que pertenece un objeto es también denominado "NOTACIÓN".
Como abreviatura sólo nombrelas.

Podemos, si queremos, especificar la "Notación". Podemos explicitar el tipo listando entre '<>' sus nombres y tipos. Ponemos por ejemplo la clase point:

```
< distance : unit → float ;  
getx : int ;  
gety : int. ;  
moveTo : int * int → unit  
..... >
```

Se listan únicamente los métodos.

ELENA
DELAMANO
FREIJJE

```
let f x = x# get-x;;
val f : <get-x:'a ; ...> -> 'a = <fun>.
```

↑
tipo abierto.

El doble punto recibe el nombre de "elipsis". Se corresponde con un significado similar al de etcétera o los puntos suspensivos de nuestro lenguaje. En Ocaml, en este caso, se refiere a que tiene el método `get-x` y alguno más. Es decir, que al menos tiene ese método, pudiendo tener sólo ese o alguno a mayores.

Vamos a poner un ejemplo, recordando las clases `point` y `colored-point`.

```
class chorras =
object
  method get-x = ["Hola", "mundo"];
end;;
```

y ahora, teniendo: obtenemos mismo resultado que con `f`.

```
let pt = new point 1 0;;
let ct = new colored-point 100 "azul";;
let chl = new chorras;;
pt# get-x;; → 1 ← f pt
ct# get-x;; → 10 ← f ct
chl# get-x;; → [("Hola", "mundo")] ← f chl
```

El opentype puede definirse de dos maneras:

O bien poniendo `#nombredclase`, como
podría ser `# point`, que es el tipo abierto asociado
a `point`. Con eso se le indica que eso matchea
cualquier objeto que tenga al menos los métodos
de `point` ("como mínimo").

O también terminando la "notación" de
la clase con `...>` de la siguiente manera:

```
point
  ↳ distance : unit → float;
    get-x : int;
    get-y : int;
    moveto : int * int → unit;
    rmoveto : int * int → unit;
    tostring : unit → string;
  ...
  ...>
```

Vamos a implementar 4 funciones para
'experimentar' con los tipos abiertos.

① let f-getx-etal (x: < get-x: 'a; ...>) = x#get-x;;

② let f-only-getx (x: < get-x: 'a>) = x#get-x;;

③ let f-getxy-etal (x: < get-x: 'a; get-y: 'b; ...>) = x#get-x;;

④ let f-onlygetxy (x: < get-x: 'a; get-y: 'b>) = x#getx;;

① Tipo de la función f-getx-etal:

tiene al menos un método `get-x:a`.

② Tipo de la función f-only-getx:

tiene sólo el método `get-x:a`.

③ Tipo de la función f-getxy-etal:

tiene al menos los métodos `get-x:a` y
`get-y:b`.

④ Tipo de la función f-onlygetxy

tiene sólo los métodos `get-x:a` y `get-y:b`.

El parámetro `x` se refiere a un objeto.

Si definimos una clase "nopoint" que tiene los mismos métodos con los mismos nombres y tipos que `point`, pero que realizan cosas diferentes:

si tenemos funciones como la de devolver `get-x`, si especificamos que la entrada sea de tipo `point`, `nopoint` o `#point`, ¿qué pasará si las metemos en un `point` o en un `nopoint`?

Definimos también una clase `no-point-extra`, que, a mayores de los de `no-point`, tiene un método `metodillo` que es de `unit → unit`.

```

let f x = x# get -x;;
let f-point (x:point) = x# get -x;;
let f-open-point (x:#point) = x# get -x;;

```

	let pt = new no-point	let pt = new point(1,2)	let xp = new no-point-extra
f	∅	↑	∅
f-point	∅*	↑	Error ***
f-open-point	∅ **	↑	∅

no-point y no-point-extra devuelven ∅ al llamar
a get-x.

* Es válido porque point y no-point tienen el
mismo tipo... (mismos métodos).

** Es válido porque tiene al menos los mismos
métodos que point.

*** Da error porque no coincide el tipo. Tiene un
método más que point, por lo que falla.

LIMITACIÓN

Un opentype no puede aparecer en el tipo de un método, ni como entrada ni como salida. Esto sólo es posible en las funciones, como por ejemplo:

```
f (x: #point) = x#get-x ;;
```

```
class clase1 (a: #point) :  
object  
    val opoint = a  
    method get-x () = opoint#get-x  
end;;
```

Es correcta. El método es de tipo unit → int.

```
class clase_2 =  
object  
    method get-x (a: #point) = a#get-x  
end;;
```

No es válida. Tipo # point → int , NO PERMITIDO!

```
class clase3 (a: #point) =  
object  
    val opoint = a  
    method get-opoint = opoint  
end;;
```

Tampoco es válido, ya que el tipo es
unit \rightarrow #point , y Ocaml tampoco los permite
(los open types) en la salida.

Lo mejor es poner equivalencias para solucionarlo:

```
let f (a:#point) = a#get-x;; #point  $\rightarrow$  int  
let f (a:#point) = a;; #point  $\rightarrow$  #point
```

Al ejecutarlo lo denota:

```
(# point as 'a)  $\rightarrow$  'a
```

EJEMPLO DE HERENCIA MÚLTIPLE

```
class acuatico =
object
    val extremidades = "aletas"
    method nadar () =    print_string ("nadando con ...^extremidades");
                         print_newline
    method saltar () =   print_string ("saltando con...^extremidades^".
Splash!!!");
                         print_newline
end;;
(* class acuatico :
object
    val extremidades : string
    method nadar : unit -> unit -> unit
    method saltar : unit -> unit -> unit
end *)

----- 0 -----



class terrestre =
object
    val altura = 10;
    val extremidades = "patas"
    method correr () =   print_string ("corriendo con ...^extremidades");
                         print_newline
    method saltar () =   print_string ("saltando con...^extremidades^".
Tromp!!!");
                         print_newline
end;;
(* class terrestre :
object
    val altura : int
    val extremidades : string
    method correr : unit -> unit -> unit
    method saltar : unit -> unit -> unit
end *)
```

```
>
>
>
> class anfibio =
>   object
>
>     inherit acuatico as super_acu
>     inherit terrestre as super_terr
>
>     method saltar_acu () = super_acu#saltar()
>     method saltar_terr () = super_terr#saltar()
>
>     method saludar () =  print_string ("saludando con ..." ^ extremidades);
>                           print_newline
>     method get_altura () = altura
>
>
>   end;;

```

```
>
>
>
> (* Warning V: the instance variable extremidades is overridden.
>    The behaviour changed in ocaml 3.10 (previous behaviour was
> hiding.)
>
> class anfibio :
>   object
>     val altura : int
>     val extremidades : string
>     method correr : unit -> unit -> unit
>     method get_altura : unit -> int
>     method nadar : unit -> unit -> unit
>     method saltar : unit -> unit -> unit
>     method saltar_acu : unit -> unit -> unit
>     method saltar_terr : unit -> unit -> unit
>     method saludar : unit -> unit -> unit
>   end *)

```

HERENCIA MÚLTIPLE

Siguiendo el ejemplo, crearemos 3 instancias:

```
let pez = new acuatico();  
let gato = new terrestre();  
let rana = new anfibio();
```

Nos quedamos con los valores del último inherit en anfibio, es decir, en el caso de las extremidades, nos quedaremos con "patas".

La última definición es la que queda.
Los atributos se sobreescreiben quedando la última, pues.

```
pez#nadar(); → nadando con aletas.  
pez#saltar(); → saltar aletas splash!  
gato#correr(); → corriendo con patas.  
gato#saltar(); → saltar patas tromp!  
rana#nadar(); → nadar con patas.  
rana#correr(); → corriendo con patas.  
rana#saltar(); → saltar patas tromp!  
rana#saltar_aqua(); → saltar patas splash!  
rana#saltar_terra(); → saltar patas tromp!  
rana#saludar(); → saludar patas.  
rana#get_altura(); → 10.
```

Paradigmas de Programación

Prácticas.

Profesor:

Jorge Graña.

Despacho:

D 4.14.

E-mail:

jorge.grana @ udc.es

Utilizar mejor ledit ocaml.

Para compilar, por ejemplo: hello.ml:

ocaml -o hello hello.ml.

./hello.

**Primeras sesiones de Prácticas Ocaml.
Cosas básicas de interés.**

ELENA
DELAMANDE
FREIJE

(Para mayor información, mirar el anexo Práctica 1)

El lenguaje Ocaml es un lenguaje declarativo funcional frente al imperativo de Pascal. En el imperativo se necesitan todos los datos para que funcione correctamente, mientras que en los lenguajes declarativos, se necesita más de las características y no tanto de la forma concreta de resolver el problema.

Si accedemos a la página "<http://caml.inria.fr/pub/distrib/>" y entramos en Ocaml 4.00, podemos encontrarnos en archivos comprimidos todos los manuales de referencia (*refman*).

La versión que estamos utilizando en los laboratorios es

Objective Caml version 3.12.1

ocamlc -v → Simplemente sirve para indicarnos qué versión de Ocaml tenemos instalada en nuestro ordenador.

o - Objective
c
a
m
l
c - Compiler

Si solamente escribimos el comando "ocaml", también nos lo indica, sólo que, aparte de mostrarnos la versión, también nos ejecuta el compilador interactivo.

Para salir de compilador interactivo, tenemos diferentes opciones:

- Podemos hacer "Ctrl+D", pero se interrumpe el flujo del programa.
- Podemos escribir "#quit;".
- Mediante la función "exit" también es posible, pero es más dirigido a salida de funciones. Para que sea válido para salir del compilador, tenemos que introducir el comando "exit 0;".

Char.code 'ñ' → Nos muestra un error cuando no debería, ya que es una línea de código válida. Para que nos funcione correctamente, deberemos ir a las opciones del terminal → “Terminal – Set character encoding – Western 1|15”.

print_endline;; → indica que es una función. Sirve para escribir en pantalla.
print_endline "ñ"; → Utilizar esta línea de código es un buen método para saber si tenemos seleccionado el código de caracteres correcto.

Para abrir el compilador interactivo de Ocaml, debemos abrir el terminal y escribir “*ocaml*”. Se iniciará así el compilador, y con ello podremos empezar la práctica. Existe otra posibilidad de inicio del compilador, que es “*ledit ocaml*”. Esto es más cómodo a la hora de realizar las prácticas, ya que nos permite acceder a las líneas anteriores de código y el desplazamiento por ellas mediante las flechas del cursor.

COMPILADOR INTERACTIVO.

Cuando se introduce un float en el compilador (como por ejemplo 1.0), la salida nos muestra su tipo (que es *float*), y lo iguala a “1.”, es decir, al ser exacto, le saca los decimales y le deja simplemente el punto. En caso de que los decimales no fueran ceros, se mostraría el mismo número que se ha introducido. Por ejemplo:

```
# 1.54;;
(* - : float = 1.54 *)
```

```
# 3.0 *. 2.0 ** 3.0;;
(* - : float = 24.00 *)
```

El resultado es 24.00, debido a que primero tiene más preferencia la potencia, 2^3 , y luego se multiplica por 3.

$$\begin{aligned}2^3 &= 8 \\3 * 8 &= 24\end{aligned}$$

El ‘*.’ es la multiplicación de números en punto flotante.
El ‘**’ es la potencia de números en punto flotante.

Con el valor “*int_of_float*” podemos pasar un número de float a int. Para ello, lo que hace es truncar el número en coma flotante y quedarse solamente con la parte entera del número, de la siguiente manera:

```
# int_of_float (14.63);;
(* - : int = 14 *)
```

Para redondear los números en coma flotante existen dos valores diferentes: “*floor*” y “*ceil*”. El valor “*floor*” se encarga de redondear los números hacia abajo. En cambio, el valor “*ceil*”, simplemente se encarga de redondear los números hacia arriba. La salida en cualquiera de las dos “funciones” es un float.

Instrucciones para la primera entrega de prácticas de Paradigmas de la Programación (PP)

Las prácticas se entregarán por SVN. La fecha límite para la entrega de las tres primeras prácticas es el jueves 31 de octubre de 2013 (a cualquier hora de este día).

Cada alumno debe subir a su repositorio tres directorios: P1, P2 y P3. El directorio P1 debe contener el fichero expresiones.ml. El directorio P2 debe contener el fichero miList.ml. El directorio P3 debe contener el fichero sorted_by.ml.

El repositorio de PP de cada alumno es:

<https://svn.fic.udc.es/grao2/pp/13-14/nombre.apellido>

Si *ruta_directorio* es la ruta a un directorio que contiene únicamente los tres directorios y ficheros citados anteriormente, para subirlo al repositorio, hay que utilizar el siguiente comando:

```
svn import --username=nombre.apellido ruta_directorio
https://svn.fic.udc.es/grao2/pp/13-14/nombre.apellido
-m "Mensaje"
```

Para comprobar los contenidos del repositorio cada alumno puede conectarse a:

<https://svn.fic.udc.es/grao2/pp/13-14/nombre.apellido>

Y para bajar el contenido del repositorio a fecha *aaaa/mm/dd*, se puede utilizar el comando:

```
svn checkout https://svn.fic.udc.es/grao2/pp/13-14/nombre.apellido
-r "{aaaa-mm-dd}"
```

Ejemplo

A continuación se muestra un ejemplo de preparación y envío de las prácticas. Supongamos que el *login* del alumno es jose.molineli. Las operaciones siguientes se realizan en un directorio que contiene ÚNICAMENTE los directorios y archivos que hay que enviar:

```
usuario@maquina:~/PP/PrimeraEntrega$ ls -R
./P1:
expresiones.ml
./P2:
miList.ml
./P3:
sorted_by.ml
```

```
usuario@maquina:~/PP/PrimeraEntrega$ svn import --username=jose.molineli
https://svn.fic.udc.es/grao2/pp/13-14/jose.molineli -m "Primera
entrega de PP de Jose Molineli"
```

El alumno jose.molineli podría descargar el contenido de su repositorio a fecha 01/11/2013 con el siguiente comando:

```
usuario@maquina:~/PP/PrimeraEntrega/Comprobacion$ svn checkout
--username=jose.molineli https://svn.fic.udc.es/grao2/pp/13-14/jose.molineli -r "{2013-11-01}"
```

Manejo del repositorio SVN de prácticas

Para **actualizar el contenido del repositorio** de prácticas de un usuario, llamémosle `jose.molinelli` seguiremos los siguientes pasos:

1- Descargamos el contenido actual de nuestro repositorio en cualquier directorio local:

```
svn checkout --username=jose-molinelli https://svn.fic.udc.es/grao2/pp/13-14/jose.molinelli
```

Esto creará un directorio dentro denominado `jose.molinelli` y que contiene lo que habíamos subido a nuestro repositorio. Importante: comprobar que ese directorio contiene un archivo denominado ".svn"

2- Realizamos los cambios que consideremos convenientes dentro del directorio `jose.molinelli` que acabamos de descargar. Si los cambios afectan al listado de archivos y subdirectorios que queremos que haya en el repositorio, deberemos indicarlo con distintas llamadas al comando `svn`. Por ejemplo:

- Si queremos que un archivo (o subdirectorio entero) que hayamos creado o copiado al directorio local sea añadido posteriormente al repositorio, usaremos la orden:

```
svn add miarchivol
```

- Si queremos que un archivo (o subdirectorio entero) que existía en el repositorio desaparezca de él, deberemos escribir la orden

```
svn delete miarchivol
```

- Si lo que queremos es simplemente modificar el contenido de algún archivo que ya existía, lo podemos cambiar con un editor o machacar con otra copia que tengamos más actualizada, siempre que lo coloquemos en el mismo sitio y con el mismo nombre.

3- En ninguna de las operaciones anteriores se produce un cambio inmediato en el repositorio. Lo único que hacen es "anotar" las modificaciones que se realizarán mas adelante, todas simultáneamente. Cuando hayamos hecho los cambios que creamos convenientes como se indica arriba, ejecutamos, desde el directorio local `jose-molinelli`, el comando:

```
svn commit .
```

4- Por último, comprobamos usando un navegador que los cambios se han subido correctamente, conectándonos a la URL

<https://svn.fic.udc.es/grao2/pp/13-14/jose.molinelli>

Instrucciones de entrega de las prácticas 4 y 5

1. Cada alumno deberá crear en su repositorio los directorios P4 y P5.
2. El directorio P4 contendrá exclusivamente el archivo `sort.ml` con la definición de todas las funciones que se solicitaban en el enunciado de esta práctica.
3. El directorio P5 contendrá los siguientes archivos:
 - o Los ficheros `fb_tree.ml` y `fb_tree.mli` con la implementación que se haya realizado del módulo `Fb_tree`, tal y como se pedía en los ejercicios 1 y 2 de esta práctica.
 - o El fichero `fb_tree_plus.ml` con las implementaciones que se pedían en los ejercicios 4 y 5 de esta práctica.
4. La fecha límite para la entrega de estas prácticas es el viernes 22/11/2013 (a cualquier hora de este día).

Instrucciones de entrega de la práctica 6

1. Cada alumno deberá crear en su repositorio el directorios P6.
2. El directorio P6 contendrá exclusivamente el archivo `logo_v0.ml` descrito en el apartado 1 del enunciado de la Práctica 7.
3. El archivo `logo_v0.ml` debe contener necesariamente las definiciones de los siguientes tipos y valores, según lo especificado en dicho apartado.

```
type exp_ari
type estado
type programa
val eval_exp: exp_ari -> float
val ejecutar: estado -> programa -> estado
val st0: estado
val p1: programa
val p2: programa
val p3: programa
```

4. La fecha límite para la entrega de esta práctica es el lunes 2 de diciembre a las 10:00 horas.

Instrucciones de entrega de las prácticas 7, 8 y 9

1. Las prácticas 7 y 8 se entregarán conjuntamente. Así pues, cada alumno deberá crear en su repositorio únicamente los directorios P8 y P9.
2. El directorio P8 contendrá exclusivamente:
 - o El archivo `logo.ml` descrito en el apartado 3 del enunciado de la práctica 7.
 - o Y el archivo `main_loop.ml` descrito en el apartado 1 del enunciado de la práctica 8.

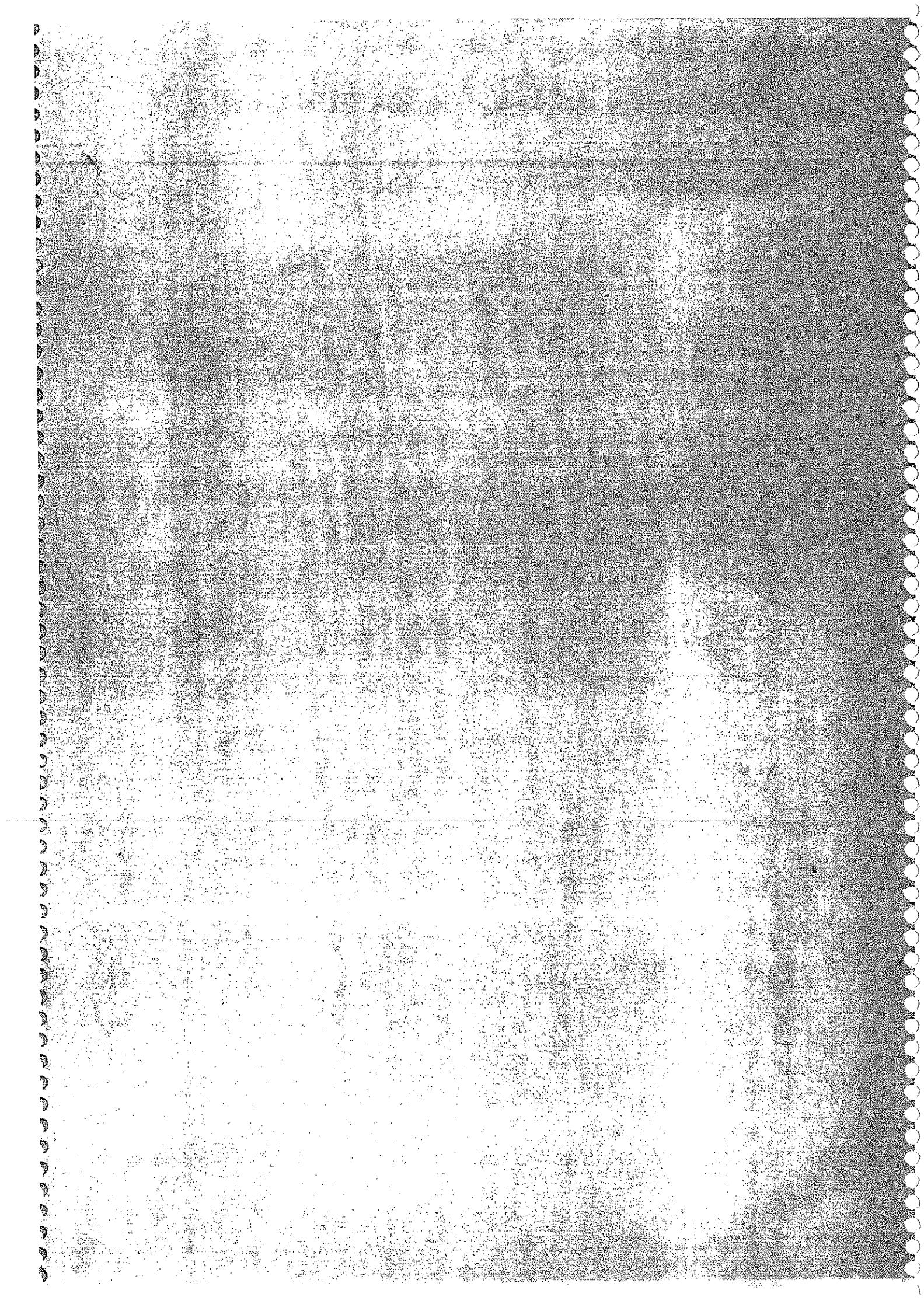
Asegúrese de que estos dos archivos (en combinación con los otros archivos que aparecen en el directorio `src` que se obtiene al descomprimir el paquete `logo.zip`) compilan con el comando `make` y generan correctamente un archivo ejecutable de nombre `logo`.

3. El directorio P9 contendrá exclusivamente el archivo `vectordinamico.ml`. Asegúrese de que este archivo compila correctamente con la orden

```
ocamlc -c vectordinamico.ml
```

para garantizar que su implementación pueda ser utilizada como un módulo de Ocaml.

4. La fecha límite para la entrega de estas prácticas es el **viernes 27 de diciembre (a cualquier hora de este día)**.



Práctica 1

```
( );  
2 + 5 * 3;;  
1.0;;  
1.0 * 2;;  
2 - 2.0;;  
3.0 + 2.0;;  
5 / 3;;  
5 mod 3;;  
3.0 *. 2.0 ** 3.0;;  
3.0 = float_of_int 3;;  
sqrt 4;;  
int_of_float 2.1 + int_of_float (-2.9);;  
truncate 2.1 + truncate (-2.9);;  
floor 2.1 +. floor (-2.9);;  
ceil 2.1 +. ceil -2.9;;  
'B';;  
int_of_char 'A';;  
char_of_int 66;;  
Char.code 'B';;  
Char.chr 67;;  
\067;;  
Char.chr (Char.code 'a' - Char.code 'A' + Char.code 'Ñ');;  
Char.uppercase 'ñ';;  
Char.lowercase 'ó';;  
"this is a string";;  
String.length "longitud";;  
"1999" + "1";;  
"1999" ^ "1";;
```

ELENA
DELAMANO
FREIJE

```
int_of_string "1999" + 1;;
"\064\065";;
string_of_int 010;;
not true;;
true && false;;
true || false;;
true or false;;
true and false;;
(1 < 2) = false;;
"1" < "2";;
2 < 12;;
"2" < "12";;
"uno" < "dos";;
2,5;;
"holo", "adios";;
0, 0.0;;
fst ('a',0);;
snd (false, true);;
(1,2,3);;
(1,2),3;;
fst ((1,2),3);;
(),abs;;
if 3 = 4 then 0 else 4;;
if 3 = 4 then "0" else "4";;
if 3 = 4 then 0 else "4";;
(if 3 < 5 then 8 else 10) + 4;;
let pi = 3.14;;
sin (pi /. 2.);;
```

```
let x = 1;;
let y = 2;;
x - y;;
let x = y in x - y;;
x - y;;
z;;
let z = x + y;;
z;;
let x = 5;;
z;;
let y = 5 in x + y;;
x + y;;
let p = 2,5;;
snd p, fst p;;
p;;
let p = 0,1 in snd p, fst p;;
p;;
let x,y = p;;
let z = x + y;;
let x,y = p,x;;
let x = let x,y = 2,3 in x * x + y;;
x + y;;
z;;
let x = x + y in let y = x * y in x + y + z;;
x + y + z;;
int_of_float;;
float_of_int;;
```

```
int_of_char;;
char_of_int;;
abs;;
sqrt;;
truncate;;
ceil;;
floor;;
Char.code;;
String.length;;
fst;;
snd;;
function x -> 2 * x;;
(function x -> 2 * x) (2 + 1);;
function (x,y) -> x;;
let f = function x -> 2 * x;;
f (2+1);;
f 2 + 1;;
```

(* Práctica 1. Paradigmas de Programación - Expresiones.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2 *)

0;;
(* - : unit = 0*)

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

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

1.0 * 2;;
(* Error de tipos, '*' es la multiplicación para tipo int.
'Error: This expression has type float but an expression was expected of type int'. *)

2 - 2.0;;
(* Error de tipos, '-' es la resta para tipo int.
'Error: This expression has type float but an expression was expected of type int'. *)

3.0 + 2.0;;
(* Error, '+' es la suma de enteros.
'Error: This expression has type float but an expression was expected of type int'. *)

5 / 3;;
(* - : int = 1 *)

5 mod 3;;
(* - : int = 2 *)

3.0 *. 2.0 ** 3.0;;
(* - : float = 24.00 *)

3.0 = float_of_int 3;;
(* La igualdad es verdadera -> tipo bool.
- : bool = true *)

sqrt 4;;
(* Error porque el valor 'sqrt' sólo es válido para tipo float.
'Error: This expression has type int but an expression was expected of type float' *)

int_of_float 2.1 + int_of_float (-2.9);;
(* - : int = 0 *)

truncate 2.1 + truncate (-2.9);;
(* - : int = 0 *)

floor 2.1 +. floor (-2.9);;

(* - : float = -1.

Mediante el redondeo sería 2 + (-3) *)

ceil 2.1 +. ceil -2.9;;

(* Error, dado que tienen que ser tipos float. El error salta porque '-2.9' no está entre paréntesis. Colocándolo entre paréntesis, el resultado sería:

- : float = 1.

Mediante el redondeo, esta vez sería 3 + (-2)

El error que sale en el compilador es:

Error: This expression has type float -> float but an expression was expected of type float.

*)

'B';;

(* - : char = 'B' *)

int_of_char 'A';;

(* - : int = 65 *)

char_of_int 66;;

(* - : char = 'B' *)

Char.code 'B';;

(* - : int = 66 *)

Char.chr 67;;

(* - : char 'C' *)

'\067';;

(* - : char 'C' *)

Char.chr (Char.code 'a' - Char.code 'A' + Char.code 'Ñ');;

(* Produce un error de sintaxis porque el paréntesis no está cerrado.

En caso de que el paréntesis se cerrase, la sintaxis es correcta y el resultado sería el char equivalente al int resultante de las operaciones.

- : char '\241' *)

Char.uppercase 'ñ';;

(* - : char = '\209'

El resultado es el código de la 'ñ' mayúscula. *)

Char.lowercase 'O';;

(* - : char = 'o'

El resultado ahora, es el correspondiente carácter en minúscula. *)

"this is a string";;

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

String.length "longitud";;

(* : - int = 8

Indica la longitud de la cadena.*)

```

# "1999" + "1";;
(* Se produce un error. El '+' sólo sirve para int, no para cadenas. *)

# "1999" ^ "1";;
(* - : string = "19991"
Concatena las dos cadenas. *)

# int_of_string "1999" + 1;;
(* - : int = 2000 *)

# "\064\065";;
(* - : string = "@A" *)

# string_of_int 010;;
(* - : string = "10" *)

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

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

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

# true or false;;
(* - : bool = true *)

# true and false;;
(* Syntax error. No existe el valor 'and' *)

# (1 < 2) = false;;
(* - : bool = false *)

# "1" < "2";;
(* - : bool = true
El carácter "2" es mayor en la tabla Ascii *)

# 2 < 12;;
(* - : bool = true *)

# "2" < "12";;
(* - : bool = false
La cadena "12" es más larga. *)

# "uno" < "dos";;
(* - : bool = false *)

# 2,5;;
(* - : int * int = (2, 5)
Es un par. *)

```

ELENA
DE LAMANO
FREIJE

```

# "hola", "adios";;
(* - : string * string = ("hola", "adios") *)

# 0, 0.0;;
(* - : int * float = (0, 0.) *)

# fst('a',0);;
(* El valor 'fst' devuelve el primer valor de un par. En este caso:
   - : char = 'a' *)

# snd (false, true);;
(* El valor 'snd' devuelve el segundo valor de un par:
   - : bool = true *)

# (1,2,3);;
(* - : int * int * int = (1, 2, 3) *)

# (1,2),3;;
(* - : (int * int) * int = ((1, 2), 3) *)

# fst ((1,2),3);;
(* - : int * int = (1, 2) *)

# 0, abs;;
(* - : unit * (int -> int) = ((), <fun>) *)

# if 3=4 then 0 else 4;;
(* - : int = 4 *)

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

# if 3=4 then 0 else "4";;
(* Surge un error, dado que las dos posibilidades tienen que ser del
   mismo tipo.
   'Error: This expression has type string but an expression was expected of type int' *)

# (if 3 < 5 then 8 else 10) + 4;;
(* - : int = 12 *)

# let pi = 3.14;;
(* val pi : float = 3.14 *)

# sin (pi /. 2.);;
(* - : float = 0.99999968293183461 *)

# let x = 1;;
(* val x : int = 1 *)

# let y = 2;;
(* val y : int = 2 *)

```

x - y;;
(* - : int = -1 *)

let x = y in x - y;;
(* Define x:=y, y luego sustituye en la expresión 'x - y'. Sólo se define x con ese valor para esa expresión, no sustituye al valor que tenía anteriormente.
- : int = 0 *)

x - y;;
(* - : int = -1 *)

z;;
(* Error. La 'z' no está definida.
Error: Unbound value z *)

let z = x + y;;
(* val z : int = 3 *)

z;;
(* - : int = 3 *)

let x = 5;;
(* val x : int = 5 *)

z;;
(* - : int = 3 *)

let y = 5 in x + y;;
(* - : int = 10 *)

x + y;;
(* - : int = 7 *)

let p = 2,5;;
(* val p : int * int = (2, 5) *)

snd p, fst p;;
(* - : int * int = (5, 2) *)

p;;
(* - : int * int = (2, 5) *)

let p = 0,1 in snd p, fst p;;
(* - : int * int = (1, 0) *)

p;;
(* - : int * int = (2, 5) *)

let x,y = p;;
(* val x : int = 2
val y : int = 5 *)

```
# let z = x + y;;
(* val z : int = 7 *)

# let x,y = p,x;;
(* val x : int * int = (2, 5)
   val y : int = 2 *)

# let x = let x,y = 2,3 in x * x + y;;
(* val x : int = 7 *)

# x + y;;
(* - : int = 9 *)

# z;;
(* - : int = 7 *)

# let x = x + y in let y = x * y in x + y + z;;
(* - : int = 34 *)

# x + y + z;;
(* - : int = 16 *)

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

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

# int_of_char;;
(* - : char -> int = <fun> *)

# char_of_int;;
(* - : int -> char = <fun> *)

# abs;;
(* - : int -> int = <fun> *)

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

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

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

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

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

```

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

# fst;;
(* - : 'a * 'b -> 'a = <fun> *)

# snd;;
(* - : 'a * 'b -> 'b = <fun> *)

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

# (function x -> 2 * x) (2 + 1);;
(* Sustituye la 'x' de la función por el '2 + 1'.
   - : int = 6 *)

# function (x,y) -> x;;
(* - : 'a * 'b -> 'a = <fun> *)

# let f = function x -> 2 * x;;
(* val f : int -> int = <fun> *)

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

# f 2 + 1;;
(* Al no haber paréntesis que indique que la función tiene que ser de la suma (como en el
ejemplo anterior), ahora se realiza la función f(2) y luego se le suma 1:
   - : int = 5 *)

```

ELENA
DELAMANO
FREIJE

Paradigmas de Programación

Práctica 2

Redefina las funciones ***hd***, ***tl***, ***length***, ***nth*** y ***append***, del módulo *List*, sin utilizar ese módulo ni la función (@) del módulo *Pervasives*.

Redefina las funciones ***rev***, ***rev_append***, ***concat***, ***flatten***, ***map***, ***map2***, ***fold_left*** y ***fold_right***, del módulo *List*, sin utilizar ese módulo.

Redefina las funciones ***find***, ***for_all***, ***exists***, ***mem***, ***filter***, ***find_all***, ***partition***, ***split*** y ***combine*** del módulo *List*, sin utilizar ese módulo.

Defina una función ***remove***: '*a* -> '*a* list -> '*a* list, que “elimine la primera aparición, si la hay, de un valor en una lista”; de forma que, por ejemplo *remove* 3 [2; 6; 3; 4; 3] sea la lista [2; 6; 4; 3] y *remove* 3 [1; 2; 4] sea la lista [1; 2; 4].

Defina una función ***remove_all***: '*a* -> '*a* list -> '*a* list, que “elimine todas las apariciones de un valor en una lista”; de forma que *remove_all* 3 [2; 6; 3; 4; 3] sea la lista [2; 6; 4].

Defina una función ***ldif***: '*a* list -> '*a* list -> '*a* list, de forma que *ldif l1 l2* elimine de *l1* todas las apariciones de todos aquellos valores que aparezcan en *l2*. Así, por ejemplo, *ldif* [1;2;3;2;4] [2;3;3;5] debería ser la lista [1;4].

Defina una función ***lprod***: '*a* list -> '*b* list -> ('*a* * '*b*) list, de forma que *lprod l1 l2* calcule el “producto cartesiano” de *l1* y *l2*. Así, por ejemplo, *lprod* [1;3;1;2] ['*a*'; '*b*'] debería ser la lista [(1,'*a*'); (1,'*b*'); (3,'*a*'); (3,'*b*'); (1,'*a*'); (1,'*b*'); (2,'*a*'); (2,'*b*')].

(*

Práctica 2. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

(* ----- *)

(* Funcion hd *)

```
let hd l = match l with
[] -> raise (Failure "hd")
| (h::_) -> h;;
```

(*En vez de poner '(Failure "hd")' podría ponerse como 'failwith "hd"'*)

(* ----- *)

(* Funcion tl *)

```
let tl l = match l with
[] -> failwith "tl"
| (_::t) -> t;;
```

(* ----- *)

(* Funcion length *)

```
let length l =
  let rec aux sum = function
    [] -> sum
    _::t -> aux (sum+1) t
  in aux 0 l;;
```

(* ----- *)

(* Funcion nth *) Devuelve el n-th elemento de la lista

```
let nth lista pos =
  if (pos < 0) then
    raise (Invalid_argument "nth")
  else
    let rec aux = function
      ([],_) -> raise (Failure "nth")
      | (h::_, 0) -> h
      | (h::t, n) -> aux (t, (n-1))
    in aux (lista, pos);;
```

(* ----- *)

(* Funcion append *) Concatena dos listas

```
let rec append l1 l2 = match (l1,l2) with
  | ([], l) -> l
  | (h::t, l) -> h :: (append t l);;
```

(* ----- *)

(* Funcion rev *) Reverso

```
let rev l =
  let rec aux acum = function
    | [] -> acum
    | h::t -> aux(h::acum) t
  in aux [] l;;
```

(* ----- *)

(* Funcion rev_append *) Reversa l1 y la concatena con l2

```
let rec rev_append l1 l2 = match l1 with
  | [] -> l2
  | (h::t) -> rev_append t (h :: l2);;
```

(* ----- *)

(* Funcion concat *) Concatena listas de listas

```
let rec concat = function
  | [] -> []
  | h::t -> append h (concat t);;
```

(* ----- *)

(* Funcion flatten *)

```
let flatten = concat;;
```

(* ----- *)

(* Funcion map *)

aplica la función f a cada elemento
de la lista ejemplo $[f(a_1); f(a_2); \dots; f(a_n)]$

```
let rec map f l = match l with
  | [] -> []
  | h::t -> f(h)::map f t;;
```

(* Funcion map2 *)

Ejemplo: $f(a,b) = (f(a,b), f(a^2,b^2), \dots, f(a^n,b^n))$

```
let rec map2 op l1 l2=
  if (length l1 != length l2) then
    raise (Invalid_argument "map2")
  else
    if (length l1 == 0 || length l2 == 0) then
      []
    else
      (op (hd l1)(hd l2))::map2 op (tl l1)(tl l2);;
```

(* Coge una expresión, 2 arrays y devuelve un array con el resultado de
 $a_1(\exp)b_1, a_2(\exp)b_2, \dots$ *)

(* Funcion fold_left *)

$\int a [b^1 \dots] = \int (\dots (\int (f a b_1) b_2 \dots)$

```
let rec fold_left f accum l = match l with
  [] -> accum
  | h::t -> fold_left f (f accum h) t;;
```

(* Funcion fold_right *)

$\int a_1 (f a_2 (\dots (\int a_n b) \dots))$

```
let rec fold_right f l accum = match l with
  [] -> accum
  | h::t -> f h (fold_right f t accum);;
```

(* Funcion find *)

Devuelve el primer elemento de la lista que satisface el predicable e

```
let rec find e l = match l with
  [] -> raise (Not_found)
  | (h::t) -> if (e h) then
    h
  else
    find e t;;
```

(*)

(* Funcion for_all *)

```
let rec for_all p = function
  [] -> true
  | h::t -> (p h) && (for_all p t);;
```

chequea si todos los elementos de la lista satisfacen el predicado p

(* ----- *)

(* Funcion exists *)

```
let rec exists p = function
  [] -> false
  | h::t -> if p h then
    true
  else
    exists p t;;
```

chequea si al menos hay un elemento que satisface p

(* ----- *)

(* Funcion mem *)

```
let rec mem n = function
  [] -> false
  | h::t -> compare h n = 0 || mem n t;;
```

mem al es true si, y solo si a es igual a un elemento de l

(* ----- *)

(* Funcion filter *)

```
let filter p =
  let rec find acum = function
    [] -> rev acum
    | h::t -> if p h then
      find (h::acum) t
    else
      find acum t in find [];;
```

distribuye todos los elementos que satisfacen un 'predicado'

ELENA
DELAMANO
FREIJE

(* ----- *)

(* Funcion find_all *)

```
let find_all = filter;;
```

(* ----- *)

(* Funcion partition *)

(* ----- *)

(* Funcion split *)

transforma una lista de pares
en un par de listas

```
let rec split = function
  [] -> ([] , [])
  | (x,y)::t -> let (lx, ly) = split t in (x::lx, y::ly);;
```

(* ----- *)

al revés de split

(* Funcion combine *)

```
let rec combine l1 l2 = match (l1, l2) with
  ([], []) -> []
  | (a1::t1, a2::t2) -> (a1, a2) :: combine t1 t2
  | (_, _) -> raise (Invalid_argument "combine");;
```

(* ----- *)

(* Funcion remove *)

```
let rec remove n l = match l with
  [] -> []
  | h::t -> if (n = h) then
    t
  else
    h::(remove n t);;
```

(* ----- *)

(* Funcion remove_all *)

```
let rec remove_all n l = match l with
  [] -> []
  | h::t -> if (n = h) then
    remove_all n t
  else
    h::(remove_all n t);;
```

(* ----- *)

(* Funcion ldif *)

```
let rec ldif l1 l2 = match l1, l2 with
  ([], _) -> raise (Failure "ldif")
  | (l1, []) -> l1
  | (h1::t1, h2::t2) -> ldif (remove_all h2 l1) t2;;
```

(* ----- *)

(* Funcion lprod *)

```
let rec lprod l1 l2 = match l1 with
  [] -> []
  | h::t -> (map (function x -> (h, x)) l2) @ (lprod t l2);;
```

(* ----- *)

Apuntamientos.

En las listas, los ":" evalúan por la derecha:

```
1 :: 2 :: 3 :: []
1 :: 2 :: [3]
1 :: [2; 3]
[1; 2; 3]
```

ELENA
DELAMANO
FREIJE

Con las potencias (***) pasa exactamente lo mismo, asocia por la derecha:

```
2.0 *** 3.0 *** 2.0;;
2.0 *** 9.0;;
```

raise es una función de un valor fijo a uno polimórfico.

: exn -> 'a = <fun>

Failure, por sí mismo no es nada. Es un constructor.

Si ponemos Failure "hd", sí que ya completa la instrucción.

raise Failure "hd"

Da error, porque al haber espacio, aplica raise sobre Failure, y este último sobre "hd".
Necesitamos poner () para forzar a aplicarselo a (failure "hd").
Informa del error y aborta el "programa" (función).

```
append [1; 2; 3] [4; 5]
1 :: (append [2; 3] [4; 5])
1 :: (2 :: (append [3] [4; 5]))
1 :: (2 :: (3 :: (append [] [4;5])))
1 :: (2 :: (3 :: [4; 5]))
1 :: (2 :: [3; 4; 5])
1 :: ([2; 3; 4; 5])
[1; 2; 3; 4; 5]
```

En la función nth pondremos en los errores → nth en ambos.

Paradigmas de Programación

Práctica 3

Defina una función **sorted** : '*a list* -> *bool* que indique, para cada lista, si sus elementos están ordenados según la relación de orden habitual (\leq).

Una relación de orden en un tipo '*a* puede representarse mediante una función de tipo '*a* -> '*a* -> *bool*.

Defina una función **sorted_by** : ('*a* -> '*a* -> *bool*) -> '*a list* -> *bool*, que sirva para comprobar si una lista está ordenada mediante una relación de orden dada. En general, *sorted_by r* [*a₁*; *a₂*; *a₃*; ...; *a_{n-1}*; *a_n*] debería ser equivalente a *r a₁ a₂ && r a₂ a₃ && ... && r a_{n-1} a_n* (es decir, cada elemento de la lista debe estar relacionado, mediante *r*, con el siguiente elemento).

Redefina la función **sorted**, del primer apartado, utilizando la función **sorted_by**.

Defina las siguientes funciones utilizando **sorted_by**:

crec : '*a list* -> *bool* (* indica si la lista está en orden creciente *)

st_crec : '*a list* -> *bool* (* indica si la lista es estrictamente creciente *)

decr : '*a list* -> *bool* (* indica si la lista está en orden decreciente *)

st_decr : '*a list* -> *bool* (* indica si la lista es estrictamente decreciente *)

plana : '*a list* -> *bool* (* indica si todos los elementos de la lista son iguales *)

doblada : '*int list* -> *bool* (* indica si cada elemento de la lista es el doble del anterior *)

Dado *s* : *string*, en ocaml podemos referirnos al carácter que ocupa la posición *i*-ésima del *string* utilizando la notación *s.[i]* (donde *i* debe ser una expresión de tipo *int* tal que $0 \leq i < \text{String.length } s$).

Defina (sin usar otras funciones del módulo *String*) las funciones:

char_in_st : '*char* -> '*string* -> *bool* (* indica si un carácter aparece en un *string* *)

st_search_fst : '*string* -> '*char* -> *int* (* devuelve el índice de la primera aparición de un carácter en un *string*; *st_search_fst s c* activa la excepción *Failure "st_search_fst"* si *char_in_st c s* es *false* *)

st_search_all : '*string* -> '*char* -> '*int list* (* devuelve la lista de todas las posiciones en las que aparece un carácter dado dentro de un *string*, en orden creciente *)

(*

Práctica 3. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

(* ----- *)

(* Funcion sorted *)

```
let rec sorted = function
[] -> true
| (h::l) -> true
| (h::t) -> if (h <= List.hd(t)) then
    sorted(t)
    else
    false
;;

```

(* ----- *)

(* Funcion sorted_by *)

```
let rec sorted_by r l = match l with
a::b::t -> if r a b then sorted_by r (b::t) else false
| _ -> true;;
```

(* ----- *)

(*Funcion sorted redefinida*)

```
let sorted l = sorted_by (<=) l;;
```

(* ----- *)

(* Funcion crec*)

```
let crec l = sorted_by (<=) l;;
```

(* ----- *)

```

(* Funcion st_crec*)

let st_crec l = sorted_by (<) l;;

(* ----- *)

(* Funcion decr*)

let decr l = sorted_by (>=) l;;

(* ----- *)

(* Funcion st_decr*)

let st_decr l = sorted_by (>) l;;

(* ----- *)

(* Funcion plana*)

let plana l = sorted_by (=) l;;

(* ----- *)

(* Funcion doblada*)

let doblada l = sorted_by (function x -> function y-> y=2*x) l;;

(* ----- *)

(* Funcion char_in_st*)

let char_in_st c s =
  let rec aux i =
    if (i < String.length s) then
      if (s.[i] == c) then
        true
      else aux (i+1)
    else false;
  in aux 0;

```

(* ----- *)

```
(* Funcion st_search_fst *)  
  
let st_search_fst c s =  
    let rec aux i =  
        if (i < String.length s) then  
            if (s.[i] == c) then  
                i  
            else aux (i+1)  
        else raise (Failure "st_search_fst");  
  
    in aux 0;;
```

(* ----- *)

```
(* Funcion st_search_all *)  
  
let st_search_all s c =  
    if char_in_st c s = false then  
        raise (Failure "st_search_all")  
    else  
  
        let rec aux i =  
            if (i < ((String.length s) - 1)) then  
                (if (s.[i] == c) then  
                    i::aux (i+1)  
  
                else aux (i+1))  
            else [i];  
  
        in aux 0;;
```

ELENA
DELAMANO
FREIJE

Paradigmas de Programación

Práctica 4

La "pseudofunción" rlist definida a continuación permite crear listas "aleatorias" de enteros de la longitud deseada:

```
let rec rlist r l =
  if l <= 0 then []
  else Random.int r :: rlist r (l-1);;
```

¿Es recursiva terminal? Si no lo fuera, ¿qué inconveniente podría tener?
Si no lo es, haga una versión recursiva terminal de la "misma función".

Las siguientes definiciones implementan en ocaml algoritmos de ordenación siguiendo los métodos de selección y de inserción:

```
let rec remove x = function
  []    -> []
  | h::t -> if x = h then t else h::remove x t;;
```



```
let lmin (h::t) = fold_left min h t;;
```



```
let rec s_sort = function
  []    -> []
  | l    -> let m = lmin l in
              let resto = remove m l in
                m::s_sort resto;;
```



```
let rec insert x = function
  []    -> [x]
  | h::t -> if x <= h then x::h::t
              else h::insert x t;;
```



```
let rec i_sort = function
  []    -> []
  | h::t -> insert h (i_sort t);;
```

¿Son ambas implementaciones recursivas terminales? Si no lo son, ¿supone eso en la práctica algún tipo de limitación?

Compare empíricamente la rapidez en la aplicación de ambos métodos. Puede utilizar para ello la "pseudofunción" rlist definida anteriormente y la "pseudofunción" crono definida a continuación:

```
let crono f x =
  let t = Sys.time () in
  let _ = f x in
  Sys.time () -. t;;
```

¿Cuál es más rápida?

¿Cómo crece el tiempo de ejecución con la longitud de las listas?

¿Existen casos especialmente "fáciles" o "difíciles"?

Si determinó que las implementaciones anteriores no eran recursivas terminales, realice implementaciones alternativas (`t_s_sort` y `t_i_sort`) que sí lo sean. ¿Presentan en la práctica alguna ventaja? ¿Y algún inconveniente? ¿Cuáles son más rápidas?

Si la implementación terminal realizada para el método de inserción resultase más lenta que la original, intente mejorarla de modo que no lo sea. Debe tratarse, en todo caso, de una implementación recursiva terminal basada en el método de inserción.

(*

Práctica 4. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

La "pseudo-función" rlist definida a continuación permite crear listas "aleatorias" de enteros de la longitud deseada:

*)

```
open List;;
```

```
let rec rlist r l =  
    if l <= 0 then []  
    else Random.int r :: rlist r (l-1);;
```

(* - ¿Es recursiva terminal?

No es recursiva terminal, porque cuando hace la llamada recursiva aún quedan operaciones pendientes.

- Si no lo fuera, ¿qué inconveniente podría tener?

Que si se quisiera hacer una lista muy grande de elementos, podría desbordarse la pila de la memoria.

Si no lo es, haga una versión recursiva terminal de la "misma función".

*)

```
let nuevo_rlist r l =  
    let rec aux accum1 = function  
        0 -> accum1  
        | n -> aux ((Random.int r)::accum1) (n-1)  
    in aux [] l;;
```

(* Las siguientes definiciones implementan en ocaml algoritmos de ordenación siguiendo los métodos de selección y de inserción: *)

```
let rec remove x = function
  [] -> []
  | h::t -> if x = h then
    t
  else
    h::remove x t;;
```

```
let lmin (h::t) = fold_left min h t;;
```

```
let rec s_sort = function
  [] -> []
  | l -> let m = lmin l
  in let resto = remove m l
  in m::s_sort resto;;
```

ELENA
DELAMANO
FREIJE

(* -----*)

```
let rec insert x = function
  [] -> [x]
  | h::t -> if x <= h then
    x::h::t
  else
    h::insert x t;;
```

```
let rec i_sort = function
  [] -> []
  | h::t -> insert h (i_sort t);;
```

(* - ¿Son ambas implementaciones recursivas terminales?

No, ninguna de las dos son recursivas terminales. Ambas tienen operaciones pendientes ("m::" en caso del s_sort; e "insert h" en caso de i_sort).

- Si no lo son, ¿supone eso en la práctica algún tipo de limitación?

La limitación vendrá dada en función de la lista que se quiera ordenar. En caso de ser de grandes dimensiones, como ninguna de las dos funciones es terminal, puede existir desbordamiento de pila.

Compare empíricamente la rapidez en la aplicación de ambos métodos. Puede utilizar para ello la "pseudo-función" rlist definida anteriormente y la "pseudo-función" crono definida a continuación: *)

```
let crono f x =
    let t = Sys.time () in
    let _ = f x in
    Sys.time () -. t;

(* - ¿Cuál es más rápida?
   # let lista = nuevo_rlist 10 1000;;
   # let tiempo_isort = crono i_sort lista;;
   val tiempo_isort : float = 0.020002000000033604
   # let tiempo_ssort = crono s_sort lista;;
   val tiempo_ssort : float = 0.0760039999999548854
```

Es más rápida la función "i_sort".

- ¿Cómo crece el tiempo de ejecución con la longitud de la listas?

```
# let tiempo_isort = map (crono i_sort) (map (rlist 300) [1000;2000;5000;8000]);;
val tiempo_isort : float list =
[0.0160010000000738728; 0.0520040000000108193; 0.344020999999997912;
 0.96806099999920568]

# let tiempo_ssort = map (crono s_sort) (map (rlist 300) [1000;2000;5000;8000]);;
val tiempo_ssort : float list =
[0.068004999999971; 0.28001700000043481; 1.8121129999999539;
 4.8483029999998715]
```

Se puede observar que en ambos métodos de ordenación, cuántos más elementos hay, más tarda en ordenarse. Es decir, la complejidad de la ordenación depende del tamaño de la lista, del número de elementos que haya.

- ¿Existen casos especialmente "fáciles" o "difíciles"?

```
let desc n1 =
  let rec ord1 n1 l c =
    if (n1==0) then l
    else ord1 (n1-1) (c::l) (c+1)
  in ord1 n1 [] 1;

let asc n1 =
  let rec ord1 n1 l c =
    if (n1==0) then l
    else ord1 (n1-1) (c::l) (c-1)
  in ord1 n1 [] n1;

# let tiempo_isort = map (crono i_sort) (map (desc) ([1000;2000;5000;8000]));;
val tiempo_isort : float list =
[0.032002000000340588; 0.10400699999910477; 0.764047000000005;
 1.95212200000003122]

# let tiempo_isort = map (crono i_sort) (map (asc) ([1000;2000;5000;8000]));;
val tiempo_isort : float list = [0.; 0.; 0.; 0.];

# let tiempo_ssort = map (crono s_sort) (map (desc) ([1000;2000;5000;8000]));;
val tiempo_ssort : float list =
[0.24801600000006898; 0.89605599999994413; 2.6801670000000441;
 7.2484529999999781]

# let tiempo_ssort = map (crono s_sort) (map (asc) ([1000;2000;5000;8000]));;
val tiempo_ssort : float list =
[0.032001999999985316; 0.136009000000001379; 0.80404999999996601;
 2.06012900000000343]
```

Con el método de ordenación de *i*_sort, la ordenación de un vector ordenado de manera ascendente es mucho más rápida que la de un vector ordenado de manera descendente. Es el mejor caso.

En cambio, con el método de ordenación de *s*_sort, la ordenación de un vector ordenado de manera ascendente es más lenta que la de uno ordenado de manera descendente.

Si determinó que las implementaciones anteriores no eran recursivas terminales, realice implementaciones alternativas (`t_s_sort` y `t_i_sort`) que sí lo sean. *)

```
let rec remove2 l n= match l with
  []->n
  | h::t-> remove2 t (h::n);;

let tremove x l =
  let rec remove1 x l n= match l with
    [] -> n
    | h::t-> if h=x then
      remove2 t n
    else
      remove1 x t (h::n)
  in remove1 x l [];;
```



```
let t_s_sort l =
  if l = [] then
    []
  else
    let rec s_sort2 l1 n1=
      if l1 = [] then List.rev(n1) else
        let m = lmin l1 in
        let resto = tremove m l1 in
        s_sort2 resto (m::n1)
    in s_sort2 l [];;
```

ELENA
DELAMANO
FREIJE

(* ----- *)

```
let t_insert x l =
let rec aux accum1 = function
    [] -> List.rev (x::accum1)
    | h::t -> if x <= h then
                List.rev_append accum1 (x::h::t)
              else
                aux (h::accum1) t
in aux [] l;;
```

```
let t_i_sort l =
let rec aux accum1 = function
    [] -> accum1
    | h::t -> aux (t_insert h accum1) t
in aux [] l;;
```

(* - ¿Presentan en la práctica alguna ventaja?

La ventaja principal que se nos presenta, es la posibilidad de manejar lista de grandes dimensiones sin riesgo a que la pila de memoria desborde.

- ¿Y algún inconveniente?

Son un poco más lentas que las no terminales, dado que tienen que realizar más pasos.

- ¿Cuáles son más rápidas?

Las funciones implementadas de manera no terminal.

```
# let tiempo_t_s_sort = map (crono t_s_sort) (map (asc) ([1000;2000;5000;8000]));;
```

```
val tiempo_t_s_sort : float list =
[0.064003999999970632; 0.244015000000004534; 1.58409900000000192;
 4.1602599999999385]
```

```
# let tiempo_t_i_sort = map (crono t_i_sort) (map (asc) ([1000;2000;5000;8000]));;
```

```
val tiempo_t_i_sort : float list =
[0.06400400000000616; 0.22001399999999932; 1.44408900000000084;
 3.952247999999991]
```

Si la implementación terminal realizada para el método de inserción resultase más lenta que la original, intente mejorarla de modo que no lo sea. Debe tratarse, en todo caso, de una implementación recursiva terminal basada en el método de inserción. *)

Paradigmas de Programación

Práctica 5

Los árboles “binarios llenos” o “estrictamente binarios” o “propriamente binarios” son árboles binarios en los que de todo nodo, que no sea hoja, “cuelgan dos ramas”. No hay “árboles estrictamente binarios” vacíos; los más sencillos tienen un solo nodo que es raíz y hoja a la vez.

La presente práctica tiene dos objetivos fundamentales:

1. Implementar en ocaml un módulo `Fb_tree`, donde esté definido un tipo de dato `'a fb_tree` que sirva para representar “árboles binarios llenos” (full binary trees) con nodos etiquetados con valores de tipo `'a`. En dicho módulo estarán definidas también algunas funciones básicas para la creación y manipulación de “árboles binarios llenos”. Este objetivo quedará cubierto mediante la realización de los ejercicios 1, 2 y 3.
2. Ponerse ahora, no en el papel del desarrollador del módulo `Fb_tree`, sino en el papel de un usuario de dicho módulo (que conoce la interfaz del módulo, pero no su implementación interna) y ampliar el conjunto de funciones de manipulación de “árboles binarios llenos”. Este objetivo quedará cubierto mediante la realización de los ejercicios 4 y 5.

Se recomienda realizar la lectura completa de este enunciado antes de acometer la resolución de cada uno de los ejercicios propuestos.

Ejercicio nº 1 (creación de la interfaz del módulo)

El módulo `Fb_tree` debe tener la siguiente interfaz:

```
type 'a fb_tree

val string_of_tree : ('a -> string) -> 'a fb_tree -> string

exception Branches

val single : 'a -> 'a fb_tree

val comp : 'a -> 'a fb_tree * 'a fb_tree -> 'a fb_tree

val root : 'a fb_tree -> 'a

val branches : 'a fb_tree -> 'a fb_tree * 'a fb_tree
```

Por tanto, cree un fichero llamado `fb_tree.mli` cuyo contenido sea el conjunto de líneas anteriormente citadas.

Ejercicio nº 2 (creación de la implementación del módulo)

La función `string_of_tree` servirá para “visualizar como `string`” el contenido de un árbol. Habrá que pasarle como parámetro una función que permita convertir a `strings` las etiquetas de los nodos. Esta función se incluirá en el módulo sólo a efectos de facilitar la “visualización” de árboles para la realización de pruebas y no debe utilizarse en los desarrollos que se pedirán más adelante.

La función `single` servirá para construir un árbol-hoja con una etiqueta dada.

La función `comp` servirá para construir un árbol a partir de una etiqueta para la raíz y de un par de árboles que serán sus ramas.

La función `root` devolverá el valor (etiqueta) de la raíz de un árbol.

La función `branches` devolverá las ramas de un árbol. Si se aplica a un árbol que no tiene ramas (un árbol-hoja) se activará la excepción `Branches`.

Por tanto, cree un fichero llamado `fb_tree.ml` cuyo contenido sea el siguiente conjunto de líneas, completando previamente la definición de las funciones `single`, `comp`, `root` y `branches`, de forma que respeten la interfaz del módulo y las especificaciones anteriormente citadas:

```
type 'a fb_tree =
  Leaf of 'a
  | Node of 'a * 'a fb_tree * 'a fb_tree;; 

let rec string_of_tree f = function
  Leaf a ->
    "(" ^ (f a) ^ ")"
  | Node (a, t1, t2) ->
    "(" ^ (f a) ^ " " ^ (string_of_tree f t1)
      ^ " " ^ (string_of_tree f t2) ^ ")";;

exception Branches;; 

let single = ... ;;

let comp = ... ;;

let root = ... ;;

let branches = ... ;;
```

Ejercicio nº 3 (compilación del módulo)

Compile los ficheros anteriores con la orden:

```
ocamlc -c fb_tree.mli fb_tree.ml
```

Compruebe que dicha orden genera correctamente los ficheros `fb_tree.cmi` y `fb_tree.cmo`.

Ejercicio nº 4 (ampliación del conjunto de funciones de manipulación de “árboles binarios llenos”)

Utilizando el módulo `Fb_tree`, pero asumiendo que sólo se conoce la interfaz del módulo, y no su implementación interna, defina en un fichero `fb_tree_plus.ml` las siguientes funciones:

```
val is_single : 'a Fb_tree.fb_tree -> bool (* identifica árboles-hoja *)
val l_branch : 'a Fb_tree.fb_tree -> 'a Fb_tree.fb_tree (* rama izquierda *)
```

```

val r_branch : 'a Fb_tree.fb_tree -> 'a Fb_tree.fb_tree (* rama derecha *)
val size : 'a Fb_tree.fb_tree -> int (* número de nodos *)
val height : 'a Fb_tree.fb_tree -> int (* altura *)
val preorder : 'a Fb_tree.fb_tree -> 'a list
val postorder : 'a Fb_tree.fb_tree -> 'a list
val inorder : 'a Fb_tree.fb_tree -> 'a list
val leafs : 'a Fb_tree.fb_tree -> 'a list (* lista de hojas *)
val mirror : 'a Fb_tree.fb_tree -> 'a Fb_tree.fb_tree (* imagen especular *)
val treemap : ('a -> 'b) -> 'a Fb_tree.fb_tree -> 'b Fb_tree.fb_tree
(* aplica una función a todos los nodos *)
val is_perfect : 'a Fb_tree.fb_tree -> bool (* un árbol lleno es perfecto si
todas sus hojas están en el mismo nivel *)

```

Ejercicio nº 5 (opcional)

Utilizando el módulo `Fb_tree`, pero asumiendo que sólo se conoce la interfaz del módulo, y no su implementación interna, añada al fichero `fb_tree_plus.ml` la siguiente función:

```

val is_complet : 'a Fb_tree.fb_tree -> bool (* un árbol es completo si todo
nivel, excepto quizás el último, está lleno y todos los nodos del último
nivel están lo más a la izquierda posible *)

```

Sugerencia nº 1

Para utilizar el módulo `Fb_tree` puede cargarse el archivo `fb_tree.cmo` desde el compilador interactivo de ocaml, con la directiva

```
#load "fb.tree.cmo";;
```

El archivo `fb_tree.cmi` debe estar presente en el mismo directorio.

Puede evitarse el tener que calificar los nombre de los valores con el nombre del módulo si se ejecuta la sentencia

```
open Fb_tree;;
```

Sugerencia nº 2

Para definir la función `is_single` puede interceptarse la excepción `Branches` utilizando una construcción `try-with`.

Por ejemplo:

```
let is_single t =
  try let _ = branches t in false
  with Branches -> true;;
```

que también puede escribirse:

```
let is_single t =
  try branches t ; false
  with Branches -> true;;
```

ya que `<exp1>; <exp2>` es una expresión que se evalúa en ocaml evaluando primero la expresión `<exp1>` y a continuación la expresión `<exp2>` y cuyo valor es el de `<exp2>`.

Ejemplos de ejecución

A continuación se transcriben unos ejemplos de uso de las funciones que hay que definir con los resultados que deberían dar si las definiciones son correctas.

```
# let h x = single x;;
val h : 'a -> 'a Fb_tree.fb_tree = <fun>

# let h1 = h 1 and h2 = h 2 and h3 = h 3;;
val h1 : int Fb_tree.fb_tree = <abstr>
val h2 : int Fb_tree.fb_tree = <abstr>
val h3 : int Fb_tree.fb_tree = <abstr>

# let ta = comp 1 (h 2, h 3);;
val ta : int Fb_tree.fb_tree = <abstr>

# let tb = comp 4 (ta, h 5);;
val tb : int Fb_tree.fb_tree = <abstr>

# let tc = comp 4 (ta, ta);;
val tc : int Fb_tree.fb_tree = <abstr>

# let td = comp 4 (h 3, ta);;
val td : int Fb_tree.fb_tree = <abstr>

# let te = comp 6 (tb, h 0);;
val te : int Fb_tree.fb_tree = <abstr>

# let print = string_of_tree string_of_int;;
val print : int Fb_tree.fb_tree -> string = <fun>

# print te;;
- : string = "(6 (4 (1 (2) (3)) (5)) (0))"

# l_branch h1;;
Exception: Fb_tree.Banches.

# r_branch h2;;
Exception: Fb_tree.Banches.

# is_single h3;;
- : bool = true
```

```

# is_single ta;;
- : bool = false

# is_single (l_branch ta);;
- : bool = true

# let l = [h1; h2; h3; ta; tb; tc; td; te];;
val l : int Fb_tree.fb_tree list =
  [<abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>]

# List.map root l;;
- : int list = [1; 2; 3; 1; 4; 4; 4; 6]

# List.map size l;;
- : int list = [1; 1; 1; 3; 5; 7; 5; 7]

# List.map height l;;
- : int list = [1; 1; 1; 2; 3; 3; 3; 4]

# preorder tb;;
- : int list = [4; 1; 2; 3; 5]

# preorder te;;
- : int list = [6; 4; 1; 2; 3; 5; 0]

# inorder tb;;
- : int list = [2; 1; 3; 4; 5]

# postorder tb;;
- : int list = [2; 3; 1; 5; 4]

# let et= mirror te;;
val et : int Fb_tree.fb_tree = <abstr>

# preorder et;;
- : int list = [6; 0; 4; 5; 1; 3; 2]

# print et;;
- : string = "(6 (0) (4 (5) (1 (3) (2))))"

# leafs h1;;
- : int list = [1]

# leafs tb;;
- : int list = [2; 3; 5]

# leafs te;;
- : int list = [2; 3; 5; 0]

# print (treemap (function x -> x * x) td);;
- : string = "(16 (9) (1 (4) (9)))"

# List.map is_perfect l;;
- : bool list = [true; true; true; true; false; true; false; false]

# List.map is_complet l;;
- : bool list = [true; true; true; true; true; true; false; false]

```

Sugerencia nº 3

Para hacer estas pruebas con mayor comodidad, puede descargar el archivo `test.ml` y compilar `fb_tree_plus.ml` como un módulo con la orden

```
ocamlc -c fb_tree_plus.ml
```

Después puede arrancar el compilador interactivo de ocaml, cargar los módulos, abrirlos, y evaluar el archivo de pruebas `test.ml` con la directiva

```
#use "test.ml";;
```

Tenga en cuenta que el archivo `test.ml` no incluye los ejemplos que activan alguna excepción ni los ejemplos de uso de la función `is_complet`. Por tanto, la salida debería coincidir exactamente con:

```
val h : 'a -> 'a Fb_tree.fb_tree = <fun>
val h1 : int Fb_tree.fb_tree = <abstr>
val h2 : int Fb_tree.fb_tree = <abstr>
val h3 : int Fb_tree.fb_tree = <abstr>
val ta : int Fb_tree.fb_tree = <abstr>
val tb : int Fb_tree.fb_tree = <abstr>
val tc : int Fb_tree.fb_tree = <abstr>
val td : int Fb_tree.fb_tree = <abstr>
val te : int Fb_tree.fb_tree = <abstr>
val print : int Fb_tree.fb_tree -> string = <fun>
- : string = "(6 (4 (1 (2) (3)) (5)) (0))"
- : bool = true
- : bool = false
- : bool = true
val l : int Fb_tree.fb_tree list =
  [<abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>]
- : int list = [1; 2; 3; 1; 4; 4; 4; 6]
- : int list = [1; 1; 1; 3; 5; 7; 5; 7]
- : int list = [1; 1; 1; 2; 3; 3; 3; 4]
- : int list = [4; 1; 2; 3; 5]
- : int list = [6; 4; 1; 2; 3; 5; 0]
- : int list = [2; 1; 3; 4; 5]
- : int list = [2; 3; 1; 5; 4]
val et : int Fb_tree.fb_tree = <abstr>
- : int list = [6; 0; 4; 5; 1; 3; 2]
- : string = "(6 (0) (4 (5) (1 (3) (2))))"
- : int list = [1]
- : int list = [2; 3; 5]
- : int list = [2; 3; 5; 0]
- : string = "(16 (9) (1 (4) (9)))"
- : bool list = [true; true; true; true; false; true; false; false]
```

```

#load "fb_tree.cmo"
#load "fb_tree_plus.cmo"
open Fb_tree;;
open Fb_tree_plus;;
let h x = single x;;
let h1 = h 1 and h2 = h 2 and h3 = h 3;;
let ta = comp 1 (h 2, h 3);;
let tb = comp 4 (ta, h 5);;
let tc = comp 4 (ta, ta);;
let td = comp 4 (h 3, ta);;
let te = comp 6 (tb, h 0);;
let print = string_of_tree string_of_int;;
print te;;
(* l_branch h1;; *) (* Exception: Fb_tree.Banches. *)
(* r_branch h2;; *) (* Exception: Fb_tree.Banches. *)
is_single h3;;
is_single ta;;
is_single (l_branch ta);;
let l = [h1; h2; h3; ta; tb; tc; td; te];;
List.map root l;;
List.map size l;;
List.map height l;;
preorder tb;;
preorder te;;
inorder tb;;
postorder tb;;
let et= mirror te;;
preorder et;;
print et;;
leafs h1;;
leafs tb;;
leafs te;;
print (treemap (function x -> x * x) td);;
List.map is_perfect l;;
(* List.map is_complet l;; *) (* Ejercicio opcional. *)

```

ELENA
DELAMANZO
FREIJJE

(*

Práctica 5. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freijke

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

```
type 'a fb_tree
val string_of_tree : ('a -> string) -> 'a fb_tree -> string
exception Branches
val single : 'a -> 'a fb_tree
val comp : 'a -> 'a fb_tree * 'a fb_tree -> 'a fb_tree
val root : 'a fb_tree -> 'a
val branches : 'a fb_tree -> 'a fb_tree * 'a fb_tree
```

(*
Práctica . Paradigmas de Programación.
Alumna: Elena M^a Delamano Freije.
Login: elena.m.delamano.freije
Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

```
type 'a fb_tree =  
  Leaf of 'a  
  | Node of 'a * 'a fb_tree * 'a fb_tree;;
```

(* ----- *)

```
let rec string_of_tree f = function  
  Leaf a -> "(" ^ (f a) ^ ")"  
  | Node (a, t1, t2) ->  
    "(" ^ (f a) ^ " " ^ (string_of_tree f t1)  
    ^ " " ^ (string_of_tree f t2) ^ ")";;
```

(* ----- *)

exception Branches;;

(* ----- *)

```
let comp a (d,i) = Node(a,d,i);;
```

(* ----- *)

```
let single tree = Leaf tree;;
```

(* ----- *)

```
let branches tree = match tree with  
  Node (tree,d,i) -> (d,i)  
  | Leaf h -> raise(Branches);;
```

(* ----- *)

```
let root tree = match tree with  
  Node(c,...)-> c  
  | Leaf h -> h;;
```

(*

Práctica 5. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

(* ----- *)

open Fb_tree;;

(* ----- *)

(* Funcion is_single *)

let is_single t =
try let _ = branches t in false
with Branches -> true;;

(* ----- *)

(* Funcion l_branch *)

let l_branch t = let(l,_) = branches t
in l;;

(* ----- *)

(* Funcion r_branch *)

let r_branch t = let(_,r) = branches t
in r;;

(* ----- *)

(* Funcion size *)

let rec size t = if is_single t then
1
else
let (l,r) = branches t
in 1+size(l)+size(r);;

(* ----- *)

(* Funcion height *)

```
let rec height t = if is_single t then  
    1  
  else  
    1+max(height (l_branch t))(height (r_branch t));;
```

(* ----- *)

(* Funcion preorder *)

```
let rec preorder t = if is_single t then  
    [root t]  
  else  
    root t::preorder (l_branch t)@preorder (r_branch t);;
```

(* ----- *)

(* Funcion postorder *)

```
let rec postorder t = if is_single t then  
    [root t]  
  else  
    postorder (l_branch t)@postorder (r_branch t)@[root t];;
```

(* ----- *)

(* Funcion inorder *)

```
let rec inorder t = if is_single t then  
    [root t]  
  else  
    inorder(l_branch t)@[root t]@inorder(r_branch t);;
```

(* ----- *)

(* Funcion leafs *)

```
let rec leafs t =  
  if is_single t then  
    [root t]  
  else  
    leafs(l_branch t)@leafs(r_branch t);;
```

(* ----- *)

(* Funcion mirror *)

```
let rec mirror t =
  if is_single t then
    single(root t)
  else
    comp (root t) (mirror(r_branch t), mirror(l_branch t));;
```

(* ----- *)

(* Funcion treemap *)

```
let rec treemap op t =
  if is_single t then
    single(op (root t))
  else
    comp (op (root t)) ((treemap op (l_branch t)),(treemap op (r_branch t)));;
```

(* ----- *)

(* Funcion is_perfect *)

```
let rec is_perfect t =
  if is_single t then
    true
  else
    height(l_branch t) = height(r_branch t)
    && is_perfect (l_branch t)
    && is_perfect (r_branch t);;
```

(* ----- *)

(* Ejercicio 5 *)

```
let rec is_complet t =
  if is_single t then
    true
  else
    (height(l_branch t) - height(r_branch t) = 0
     || height(l_branch t) - height(r_branch t) = 1)
    && is_complet (l_branch t)
    && is_complet (r_branch t);;
```

ELENA
DELAMANZO
FREIJE

Paradigmas de Programación - Práctica 6

1 Introducción

Logo es un lenguaje de programación que permite dibujar gráficos en pantalla controlando un cursor con forma de tortuga. Para ello el lenguaje proporciona instrucciones clásicas de programación como variables, bucles o condiciones, pero también instrucciones específicas para mover a la tortuga por la pantalla y pintar.

La tortuga se mueve a través de una ventana gráfica dividida en puntos. Las instrucciones de logo permiten hacer avanzar y retroceder a la tortuga un número de puntos, girarla un cierto número de grados, y hacer que al moverse pinte o no. El objetivo de los programas en Logo suele ser dibujar figuras en la pantalla utilizando los movimientos de la tortuga.

2 Tareas

La práctica consiste en desarrollar el motor de un interprete de un subconjunto del lenguaje Logo formado por las siguientes instrucciones:

- Instrucciones para mover a la tortuga:
 - Avanzar una cierta cantidad de puntos.
 - Retroceder una cierta cantidad de puntos.
 - Girar a la izquierda un número de grados
 - Girar a la derecha un número de grados.
 - Bajar la tortuga, lo que hará que pinte al moverse.
 - Subir la tortuga, que se moverá sin pintar.
 - Borrar la pantalla. Borra todo lo dibujado en la pantalla, y sitúa a la tortuga en el centro mirando hacia arriba.
- Ejecutar instrucciones en secuencia. Esto permite escribir programas formados por más de una instrucción, por ejemplo, la secuencia de avanzar y girar.
- Repetir una instrucción un cierto número de veces. Esta instrucción proporciona un bucle básico. Por ejemplo, repetir 4 veces la secuencia de avanzar 100 y girar 90° dibujará un cuadrado.

Las instrucciones que necesitan cantidades, como avanzar, deben permitir el uso de expresiones aritméticas. Para no acumular errores de redondeo, todas las operaciones se deben realizar utilizando números flotantes.

Las expresiones aritméticas que debe implementar nuestro intérprete son las siguientes:

- Constantes.
- Suma de dos expresiones.
- Resta de dos expresiones.
- Producto de dos expresiones.
- División de dos expresiones.
- Resto de la división entera de dos expresiones.
- Potencia de dos expresiones.
- Opuesto de una expresión.

Partiendo de los tipos instrucción y expresión aritmética, hay que realizar las siguientes tareas:

- Definir un tipo de dato `exp_ari` que represente las expresiones aritméticas del lenguaje.
- Implementar un evaluador de expresiones aritméticas, es decir, una función `eval_exp: exp_ari -> float` que dado un valor de tipo `exp_ari` calcule el resultado de la expresión.
- Definir un tipo de dato `estado` que represente el estado del programa en un momento determinado. Definir también un estado inicial que tenga a la tortuga en el centro de la pantalla, mirando hacia arriba y con el lápiz bajado.
- Definir un tipo de dato `programa` que represente las instrucciones del lenguaje.
- Implementar un ejecutador de expresiones del lenguaje (`ejecutar: estado -> programa -> estado`), que dado el estado actual del programa y una instrucción ejecute esa instrucción cambiando el estado y dibujando en una ventana gráfica si es necesario.

Para implementar la ventana gráfica se recomienda usar el módulo `graphics` de caml.

(*

Práctica 6, 7. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

```
open Graphics;;
```

```
open_graph "";;
set_window_title "Logo Ocaml P6";;
```

```
set_color blue;;
```

(* Non fai falla darlle unhas dimensións concretas á pantalla, xa que pídesenos unha de
600x450, e é a que aparece por defecto. *)
(* Transladamos o punto ao que será a posición inicial (coordenadas (300,225)), que é o centro
da pantalla que imos utilizar. *)

```
moveto 300 225;;
```

(* Tipo que define as distintas expresións aritméticas. *)

```
type exp_ari = Const of float
| Suma of (exp_ari * exp_ari)
| Resta of (exp_ari * exp_ari)
| Producto of (exp_ari * exp_ari)
| Division of (exp_ari * exp_ari)
| Mod of (exp_ari * exp_ari)
| Potencia of (exp_ari * exp_ari)
| Opuesto of (exp_ari);;
```

(*

Exemplos de definición de expresiones:

```
let exp1 = Const 3.;;
  ( Equivaldría a -> 3.0 )

let exp2 = Suma (Const 4. , Const 2.);;
  ( Equivaldría a -> 4.0 + 2.0 )

let exp3 = Resta (Const 3. , Const 2.);;
  ( Equivaldría a -> 3.0 - 2.0 )

let exp4 = Producto (Const 2. , Const 5.);;
  ( Equivaldría a -> 2.0 * 5.0 )

let exp5 = Division (Const 4. , Const 2.);;
  ( Equivaldría a -> 4.0 / 2.0 )

let exp6 = RestoDiv (Const 2. , Const 3.);;
  ( Equivaldría a -> Resto de 2.0 / 3.0 )

let exp7 = Potencia (Const 1. , Const 9.);;
  ( Equivaldría a -> 1.0 ^ 9.0 )

let exp8 = Opuesto (Const 4.);;
  ( Equivaldría a -> - [4.0] )
```

*)

(* Función encargada de evaluar as expresións aritméticas definidas co tipo anterior. É de tipo 'exp_ari -> float' . *)

```
let rec eval_exp e = match e with
  Const a -> a
  | Suma (a,b) -> (eval_exp a) +. (eval_exp b)
  | Resta (a,b) -> (eval_exp a) -. (eval_exp b)
  | Producto (a,b) -> (eval_exp a) *. (eval_exp b)
  | Division (a,b) -> (eval_exp a) /. (eval_exp b)
  | Mod (a,b) -> mod_float (eval_exp a) (eval_exp b)
  | Potencia (a,b) -> (eval_exp a) ** (eval_exp b)
  | Opuesto (a) -> 0. -. eval_exp a;;
```

(*

Evaluación das distintas expresións aritméticas anteriormente definidas. Ten como tipo '`exp_ari -> float`'.

Puede ser tanto una 'Suma', 'Resta', ... , de Constantes, como una 'Suma', 'Resta', ... , de outras expresións aritméticas (tales como sumas de sumas, multiplicacións de potencias, ...).

Algunos exemplos (referidos aos exemplos de definición de expresións):

```
eval_exp exp1;;
- : float = 3.
```

```
eval_exp exp2;;
- : float = 6.
```

```
eval_exp exp3;;
- : float = 1.
```

```
eval_exp exp4;;
- : float = 10.
```

```
eval_exp exp5;;
- : float = 8.
```

```
eval_exp exp6;;
- : float = 2.
```

```
eval_exp exp7;;
- : float = 1.
```

```
eval_exp exp8;;
- : float = -4.
```

*)

(* Tipo que define o estado da tartaruga. Para elo tense en conta as coordenadas nas que está situado, o número de grados (en radiáns) que está xirado, e se o lapis está levantado ou baixado *)

```
type estado = { punto: (float*float); grados: float; lapiz: bool };;
```

(* Para definir este estado inicial, necesitamos unha función que transforme os grados en radiáns, xa que as funcións do módulo Graphics necesita que estén nesta unidade. *)

```
let grados_a_Radianes gra =
    let pi = 4. *. atan 1. in (gra *. pi) /. 180.;;
```

(* Agora definimos o que será o estado inicial, co punteiro situado no centro da pantalla, e que será ao que recurrirremos cando queiramos limpar a pantalla e comenzar de novo.
*)

```
let estado_inicial = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true };;
let st0 = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true };;
```

(* Para poder acceder as coordenadas do punto por separado, imos implementar dúas funcións 'getters':
*)

```
let getx (x,y) = x;;
let gety (x,y) = y;;
```

(* Esta é a función básica para avanzar o punteiro, que recibe como parámetros as coordenadas da posición na que está o punto, a distancia que quere moverse e o ángulo en radiáns, para saber cánto se quere xirar.
*)

```
let avanzar (x,y) n radians =
    (sin radians) *. n +. x,
    (cos radians) *. n +. y;;
```

(* Tipo que define as distintas instruccións do programa.
*)

```
type programa =
  Fd of exp_ari
  | Bk of exp_ari
  | Rt of exp_ari
  | Lt of exp_ari
  | Cs
  | Pu
  | Pd
  | Rp of (int * programa)
  | Sec of (programa * programa);;
```

(* Agora necesitamos unha función que sirva para executar todas as instruccíons definidas no tipo programa.*)

```
let rec ejecutar_estado = function
  Fd (a) -> let _ = let (x,y) = (avanzar_estado.punto (eval_exp a) estado.grados) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    in { punto = avanzar_estado.punto (eval_exp a) estado.grados; grados =
  estado.grados; lapiz = estado.lapiz}

  |Bk (a) -> let _ = let (x,y) = (avanzar_estado.punto (eval_exp a) (estado.grados +
  grados_a_Radianes 180.)) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    in { punto = avanzar_estado.punto (eval_exp a) (estado.grados +
  grados_a_Radianes 180.); grados= estado.grados; lapiz = estado.lapiz }

  |Rt (a) -> {punto = (getx estado.punto, gety estado.punto); grados= (estado.grados +
  (grados_a_Radianes (eval_exp a))); lapiz = estado.lapiz}

  |Lt (a) -> {punto = (getx estado.punto, gety estado.punto); grados= (estado.grados -
  (grados_a_Radianes (eval_exp a))); lapiz = estado.lapiz}

  |Pu -> {punto = (getx estado.punto, gety estado.punto); grados= estado.grados; lapiz = false}

  |Pd -> {punto = (getx estado.punto, gety estado.punto); grados= estado.grados; lapiz = true}

  |Rp (a,b) -> let rec bucle_estado i =
    if i >= a then
      estado
    else let estado = (ejecutar_estado b) in bucle_estado (i + 1) in bucle_estado 0

  |Sec (a,b) -> ejecutar (ejecutar_estado a) b

  |Cs -> let _ = clear_graph();
    moveto (int_of_float (getx estado_inicial.punto)) (int_of_float (gety
  estado_inicial.punto))
    in estado_inicial;
```

(* E agora, as tres definicións que se nos piden no enunciado da Práctica 7, o p1: programa, p2: programa e p3: programa, que representarán programas Logo.
*)

```
let p1 = Sec(Sec(Sec (Pu, Bk(Const 150.)), Sec(Pd, Rt (Const 30.))), Rp(6,Sec(Sec(Fd(Const  
60.),Rt(Const 60.)),Sec(Fd(Const 60.), Lt (Const 120.)))));
```

```
let p2= (Sec((Lt (Const 90.),Rp(6,Sec(Sec(Rp(3,Sec(Fd(Const 100.),Rt(Const  
120.))),Rp(2,Sec(Sec(Fd(Const 100.),Rt(Const 60.)),Sec(Fd(Const 100.),Rt(Const  
120.))))),Rt(Const 60.)))));
```

```
let p3= (Rp(6,Sec(Lt(Const 60.),Sec(Rp(2,Sec(Sec(Fd(Const 60.),Rt(Const 60.)),Sec(Fd(Const  
60.),Lt(Const 120.)))),Sec(Sec(Fd(Const 180.),Lt(Const 120.)),Rp(2,Sec(Sec(Fd(Const  
60.),Rt(Const 60.)),Sec(Fd(Const 60.),Lt(Const 120.))))))));
```

ELENA
DELAMANZO
FREIJJE

Paradigmas de Programación - Práctica 7

1 logo_v0

Incluya en un archivo “logo_v0.ml” las definiciones de los tipos `exp_ari`, `estado` y `programa` y las funciones `eval_exp: exp_ari -> float` y `ejecutar: estado -> programa -> estado` descritos en el enunciado de la Práctica 6.

Incluya en ese mismo archivo la definición de un valor `st0: estado` que represente adecuadamente el estado inicial referido en el mismo enunciado, si se supone que la ventana gráfica tiene unas dimensiones de 600 x 450 puntos.

Añada en el mismo archivo definiciones para los valores `p1: programa`, `p2: program` y `p3: programa`, que representen adecuadamente los siguientes programas logo:

`p1`

```
pu bk 150
pd rt 30
repeat 6 [fd 60 rt 60 fd 60 lt 120]
```

`p2`

```
lt 90
repeat 6 [
    repeat 3 [fd 100 rt 120 ]
    repeat 2 [fd 100 rt 60 fd 100 rt 120]
    rt 60
]
```

`p3`

```
repeat 6
[lt 60
 repeat 2 [fd 60 rt 60 fd 60 lt 120]
 fd 180 lt 120
 repeat 2 [fd 60 rt 60 fd 60 lt 120]
]
```

2 testing

Compile como un módulo el contenido del archivo "logo_v0.ml" con la orden

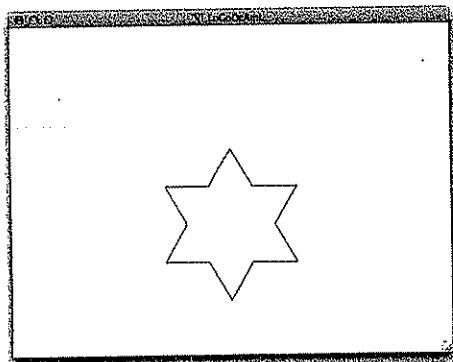
```
ocamlc -c logo_v0.ml
```

Puede comprobar el funcionamiento del módulo Logo_v0 desde el compilador interactivo de ocaml con la siguiente secuencia

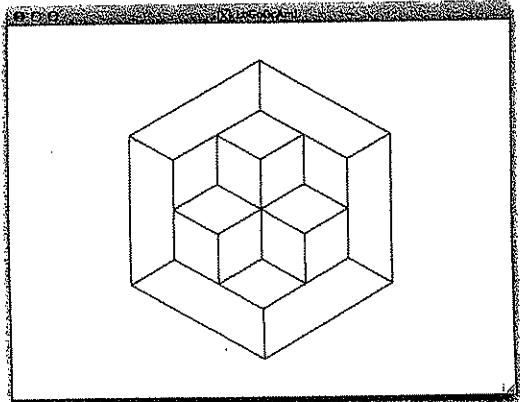
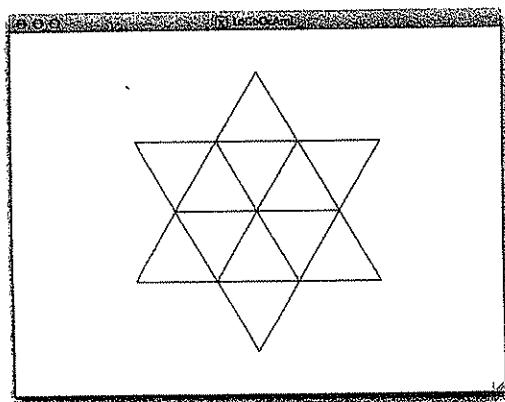
```
#load "graphics.cma";;
#load "logo_v0.cmo";;
open Graphics;;
open Logo_v0;;

open_graph "";
moveto 300 225;;
ejecutar st0 p1;;
```

Debería aparecer una ventana como la siguiente:



Si se ejecutan los programas p2 y p3 de modo análogo deberían aparecer las ventanas



3 Logo

En un archivo "Logo.ml" implemente un módulo "Logo" en el que se completen las definiciones de la Práctica 6 para incluir el uso variables numéricas, expresiones booleanas y las instrucciones `print`, `make`, `while`

Tenga en cuenta que:

- El tipo `estado` ahora deberá reflejar los valores de las variables definidas.
- El tipo `exp_ari` debe incluir ahora la aparición de variables, y la función `eval_exp` debe tener tipo `estado -> exp_ari -> float` (para tener en cuenta el valor de las variables que contenga cada expresión).
- Hay que definir un tipo `exp_bool` que permita representar expresiones booleanas, incluyendo:
 - las constantes `true` y `false`
 - la negación de una expresión booleana
`not b`
 - la conjunción y la disyunción lógicas
`b1 and b2`
`b1 or b2`
 - las comparaciones de expresiones aritméticas
 - `e1 = e2`
 - `e1 < e2`
 - `e1 > e2`
 - `e1 <= e2`
 - `e1 >= e2`
- Hay que definir una función `eval_exp_bool: estado -> exp_bool -> boolean`
- Hay que modificar la definición del tipo `programa` y la función `ejecutar` para incluir las nuevas instrucciones.

(*

Práctica 7. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

```
open Graphics;;
```

```
open_graph "";;
```

```
set_window_title "Logo Ocaml P7";;
```

```
set_color blue;;
```

(* Non fai falla darré unhas dimensións concretas á pantalla, xa que pídesenos unha de
600x450, e é a que aparece por defecto. *)

(* Transladarmos o punto ao que será a posición inicial (coordenadas (300,225)), que é o centro
da pantalla que imos utilizar. *)

```
moveto 300 225;;
```

(* Tipo que define o estado da tartaruga. Para elo tense en conta as coordenadas nas que está
situado, o número de grados (en radiáns) que está xirado, e se o lapis está levantado ou baixado e
unha lista coas variables definidas nese estado e os seus respectivos valores. *)

```
type estado = { mutable punto: (float*float); mutable grados: float; mutable lapiz: bool ; mutable  
var: (string*float) list};;
```

(* Para definir este estado inicial, necesitamos unha función que transforme os grados en
radiáns, xa que as funcións do módulo Graphics necesita que estén nesta unidade.

*)

```
let grados_a_Radianes gra =  
    let pi = 4. *. atan 1. in (gra *. pi) /. 180.;;
```

(* Agora definimos o que será o estado inicial, co punteiro situado no centro da pantalla, e que será ao que recurrirremos cando queiramos limpar a pantalla e comenzar de novo.
*)

```
let estado_inicial = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true; var = [] };;  
let st0 = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true; var = [] };;
```

(* Agora, a maiores, necesitamos dúas funcións que nos servirán para incluír elementos na lista de valores do estado, así como para obter o número decimal correspondente a un string desa lista. *)

```
let asignar_valor estado s f =  
    estado.var ← (s, f):: estado.var ;  
    estado;;
```

```
let consultar_valor estado str = match estado with  
    [] → raise (Failure "No existe variable")  
    | (h :: t) → let rec aux l = match l with  
        [] → raise (Failure "No existe variable")  
        | ((s, f) :: t) → if (s = str) then  
            f  
        else  
            aux t  
    in aux (h :: t);;
```

(* Para poder acceder as coordenadas do punto por separado, imos implementar dúas funcións 'getters': *)

```
let getx (x,y) = x;;  
let gety (x,y) = y;;
```

(* Tipo que define as distintas expresións aritméticas. *)

```
type exp_ari = Const of float  
| Suma of (exp_ari * exp_ari)  
| Resta of (exp_ari * exp_ari)  
| Producto of (exp_ari * exp_ari)  
| Division of (exp_ari * exp_ari)  
| Mod of (exp_ari * exp_ari)  
| Potencia of (exp_ari * exp_ari)  
| Opuesto of (exp_ari)  
| Var of string;;
```

ELENA
DELAMANO
FREIJE

(* Función encargada de evaluar as expresións aritméticas definidas co tipo anterior. É de tipo ' exp_ari → estado → float '. *)

```
let rec eval_exp estado e = match e with
  Const a -> a
  | Suma (a,b) -> (eval_exp estado a) +. (eval_exp estado b)
  | Resta (a,b) -> (eval_exp estado a) -. (eval_exp estado b)
  | Producto (a,b) -> (eval_exp estado a) *. (eval_exp estado b)
  | Division (a,b) -> (eval_exp estado a) /. (eval_exp estado b)
  | Mod (a,b) -> mod_float (eval_exp estado a) (eval_exp estado b)
  | Potencia (a,b) -> (eval_exp estado a) ** (eval_exp estado b)
  | Opuesto (a) -> 0. -. eval_exp estado a
  | Var s -> consultar_valor estado s;;
```

(* Esta é a función básica para avanzar o punteiro, que recibe como parámetros as coordenadas da posición na que está o punto, a distancia que quere moverse e o ángulo en radiáns, para saber cánto se quere xirar. *)

```
let avanzar (x,y) n radians =
  (sin radians) *. n +. x,
  (cos radians) *. n +. y;;
```

(* Tipo que define expresións booleanas. *)

```
type exp_bool = True
| False
| Not of exp_bool
| And of (exp_bool * exp_bool)
| Or of (exp_bool * exp_bool)
| Equal of (exp_ari * exp_ari)
| Minor of (exp_ari * exp_ari)
| Mayor of (exp_ari * exp_ari)
| MinorEqual of (exp_ari * exp_ari)
| MayorEqual of (exp_ari * exp_ari);;
```

(* Función encargada de evaluar as expresións aritméticas definidas co tipo anterior. É de tipo
'estado → exp_bool → float'. *)

```
let rec eval_exp_bool estado e = match e with
  True -> true
  | False -> false
  | Not a -> if ((eval_exp_bool estado a) == true) then
    false
    else
      true
  | And (a,b) -> ((eval_exp_bool estado a) && (eval_exp_bool estado b))
  | Or (a,b) -> ((eval_exp_bool estado a) || (eval_exp_bool estado b))
  | Equal (a,b) -> ((eval_exp_estado a) = (eval_exp_estado b))
  | Minor (a,b) -> ((eval_exp_estado a) < (eval_exp_estado b))
  | Mayor (a,b) -> ((eval_exp_estado a) > (eval_exp_estado b))
  | MinorEqual (a,b) -> ((eval_exp_estado a) <= (eval_exp_estado b))
  | MayorEqual (a,b) -> ((eval_exp_estado a) >= (eval_exp_estado b));
```

(* Tipo que define as distintas instruccións do programa. *)

```
type programa =
  Fd of exp_ari
  | Bk of exp_ari
  | Rt of exp_ari
  | Lt of exp_ari
  | Cs
  | Pu
  | Pd
  | Rp of (exp_ari * programa)
  | Sec of (programa * programa)
  | Make of (string * exp_ari)
  | Print of exp_ari
  | While of (exp_bool * programa) ;;
```

(* Agora necesitamos unha función que sirva para executar todas as instruccións definidas no
tipo programa.*)

```

let rec ejecutar estado = function
  Fd (a) -> let _ = let (x,y) = (avanzar estado.punto (eval_exp estado a) estado.grados) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
  in estado.punto ← ( avanzar estado.punto (eval_exp estado a) estado.grados);
  estado

  | Bk (a) -> let _ = let (x,y) = (avanzar estado.punto (eval_exp estado a) (estado.grados +.
  grados_a_Radianes 180.)) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
  in estado.punto ← avanzar estado.punto (eval_exp estado a) (estado.grados +.
  grados_a_Radianes 180.);
  estado

  | Rt (a) -> estado.grados ← (estado.grados +. (grados_a_Radianes (eval_exp estado a)));
  estado

  | Lt (a) -> estado.grados ← (estado.grados -. (grados_a_Radianes (eval_exp estado a)));
  estado

  | Pu -> estado.lapiz ← false
  estado

  | Pd -> estado.lapiz ← true
  estado

  | Rp (a,b) -> let rec bucle estado i =
    if (i >= (int_of_float (eval_exp estado a))) then
      estado
    else let estado = (ejecutar estado b) in bucle estado (i + 1) in bucle estado 0

  | Sec (a,b) -> ejecutar (ejecutar estado a).b
  | Cs -> let _ = clear_graph();
    moveto (int_of_float (getx estado_inicial.punto)) (int_of_float (gety
  estado_inicial.punto));
    estado.punto ← estado_inicial.punto;
    estado.grados ← estado_inicial.grados;
    estado.var ← estado_inicial.var;
    estado.lapiz ← estado_inicial.lapiz;
    in estado_inicial

  | Make (s, e) -> asignar_valor estado s (eval_exp estado e)
  | Print e -> print_endline (string_of_float (eval_exp estado e));
  estado.grados

  | While (b, e) -> if ((eval_exp bool estado b) = true) then
    (ejecutar estado (e);
     ejecutar estado (While(b,e)))
  else
    estado;;

```

ELENA
DELAMANO
FREIJE

Paradigmas de Programación - Práctica 8

Se trata de aprovechar el módulo *Logo*, construido según el apartado 3 de la Práctica 7, para implementar un bucle interactivo que interprete y ejecute programas (o instrucciones) de una pequeña parte del lenguaje Logo.

Para ello se entrega un paquete “logo.zip” que contiene los archivos necesarios para la parte del intérprete que se encargará de realizar el análisis léxico y sintáctico de los programas.

1 Lo básico

En la interfaz del módulo *Abstract_syntax* (“abstract_syntax.mli”), suministrado en el paquete, se definen los tipos de datos *expr_arit*, *expr_bool* e *instrucion*, que servirán para representar programas Logo. En el archivo que contiene la interfaz, se indica, como comentario, la sintaxis concreta con la que se corresponde cada uno de los valores de estos tipos.

En un archivo “main_loop.ml” deben definirse las funciones

```
exp_ari_of : Abstract_syntax.expr_arit -> Logo.exp_ari  
exp_bool_of : Abstract_syntax.expr_bool -> Logo.exp_bool  
program_of : Abstract_syntax.instrucion -> Logo.programa
```

que conviertan los valores de los tipos del módulo *Abstract_syntax* a los correspondientes definidos en el módulo *Logo*. Los valores de tipo *instrucción* que no tengan correspondiente en el tipo *program*, deberán provocar una excepción *Failure* “*Instrucción no implementada*” al aplicarles la función *program_of*.

El módulo *Input_program* suministrado incluye una función *get_inst : unit -> Abstract_syntax.instrucion* que, cada vez que es aplicada, muestra un “prompt” (*>>*) en la salida estándar del programa que la ejecuta, lee una línea de texto de la entrada estándar (que debería ser una instrucción o programa Logo), y devuelve el valor de tipo *instrucion* correspondiente (fruto de los análisis

léxico-sintácticos adecuados). La aplicación de esta función puede provocar los errores asociados a las excepciones *Error_lexico* y *Error_sintactico* (cuando la frase recibida no se puede analizar correctamente) o *End_of_program* (si se termina la entrada).

Debe escribirse en el archivo **main_loop.ml** el código necesario para que su ejecución produzca un bucle interactivo en el que, en cada ciclo, se interprete y ejecute una instrucción o programa Logo. Para ello, debe abrir inicialmente una ventana gráfica de dimensiones adecuadas (utilizando la función correspondiente del módulo *Graphics*), establecer un estado inicial para la “máquina Logo” y entrar en un bucle en el que se recibe un programa proporcionado por la función *get_inst*; se ejecuta dicho programa y se vuelve a comenzar. Naturalmente, la ejecución de cada instrucción o programa debe realizarse en el estado en que haya quedado la máquina después de la ejecución anterior.

La compilación del programa **main_loop.ml**, y su enlace con el código de los módulos que utiliza, puede realizarse con el comando *make* (que utiliza la descripción incluida en el archivo “*Makefile*”). Si no hay errores, esto producirá un archivo ejecutable de nombre “*logo*”.

Puede comprobar el funcionamiento del programa *logo* con los ejemplos incluidos en la carpeta *data*.

Tenga en cuenta, si lo desea, que la función *get_inst* es capaz de leer el contenido de un archivo de texto que contenga un programa Logo, utilizando el comando especial **#load "<nombre de archivo>"**

También reconoce el comando **#quit** como indicador del fin de la entrada (provocando, al recibirla, la excepción *End_of_program*).

Además, la función *get_inst* ignora las líneas que comienzan por //

2 Mejorando el intérprete (opcional)

Opcionalmente, puede mejorar el intérprete de Logo de forma que sea capaz de reconocer otras instrucciones contempladas en el tipo

Abstract_syntax.instrucion, como *setpencolor <e>*, *setpensize <e>*, *st* (mostrar la tortuga) y *ht* (ocultar la tortuga). Estas dos últimas tendrán sentido si se acomete la tarea de dibujar la tortuga en la ventana gráfica. Para ello pueden utilizarse, si se desea, las funciones:

```
Turtle.drawturtle: float * float -> float -> float ->  
bool -> bool -> Graphics.image * int * int
```

```
Turtle.redraw : Graphics.image * int * int -> unit
```

drawturtle (x,y) a ht wt vi pd dibuja una tortuga *ht* alta y *wt* ancha, en el punto (*x,y*), mirando hacia en un ángulo *a*, teniendo en cuenta el estado de visibilidad de la tortuga (*vi*) y si el lapiz está o no bajado (*pd*), y devuelve un valor que contiene la imagen "pisada" por la tortuga. Esta imagen puede restaurarse con la función *Turtle.redraw*, cuando se mueve la tortuga.

3 Para saber más...

El módulo *Mates*, contiene algunas funciones para el redondeo numérico y para la transformación de coordenadas polares en coordenadas cartesianas. (Puede utilizarlas, si lo desea, en sus módulos, pero esto debería tenerlo ya resuelto en el módulo *Logo*).

El archivo “*scanner.mll*” contiene una descripción, “estilo lex”, del léxico admitido por el intérprete (ahí pueden verse todas las “palabras clave” reconocidas). El programa *ocamllex* genera, a partir de esta descripción, un analizador léxico (en el módulo *Scanner*) que servirá para alimentar de “tokens” al analizador sintáctico.

El archivo “*parser.mly*” contiene una descripción, “estilo yacc”, de la gramática del lenguaje que reconocerá el intérprete. El programa *ocamlyacc* genera, a partir de esta descripción, los archivos “*parser.mli*” y “*parser.ml*” con el módulo que se encargará de realizar el análisis sintáctico.

La combinación de estos dos módulos nos permitirá realizar el análisis que

convierte los programas, escritos con la “sintaxis concreta” de Logo, en valores de los tipos definidos en el módulo *Abstract_syntax*, que representan esos mismos programas con una “sintaxis abstracta” .

El uso de los módulos *Scanner* y *Parser*, lo gestiona directamente el módulo *Input_program*, que lee de la entrada estándar.

El módulo *Meta_syntax*, amplia la sintaxis de los programa Logo, descrita en el módulo *Abstract_syntax*, con algunos comandos adicionales que serán reconocidos por el intérprete (*#load*, *#quit*)

Palabras clave reconocidas por el analizador léxico.

#load	#carga
#quit	#sal
cs	bp
st	mt
ht	ot
pu	sl
pd	bl
fd	av
bk	re
rt	gd
lt	gi

make	haz
repeat	repite
while	mientras
true	verdadero
false	falso
not	no
and	y
or	o
print	pr
setpencolor	poncolorlapiz
setpensize	pongrosor

Nombres de colores reconocidos por el analizador léxico.

maroon	marron
orange	naranja
olive	oliva
purple	purpura
fuchsia	fucsia
lime	lima
navy	marino
aqua	agua
teal	turquesa

silver	plata
gray	gris
black	negro
white	blanco
red	rojo
green	verde
blue	azul
yellow	amarillo
cyan	cian

(*

Práctica 8. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

```
open Graphics;;
```

```
(* open_graph "";;
set_window title "Logo Ocaml P8";;
```

```
set_color blue;;
```

Non fai falla darralle unhas dimensíons concretas á pantalla, xa que pídesenos unha de 600x450, e é a que aparece por defecto. *)
(* Transladamos o punto ao que será a posición inicial (coordenadas (300,225)), que é o centro da pantalla que imos utilizar.

```
moveto 300 225;;
```

(* Tipo que define o estado da tartaruga. Para elo tense en conta as coordenadas nas que está situado, o número de grados (en radiáns) que está xirado, e se o lapis está levantado ou baixado e unha lista coas variables definidas nese estado e os seus respectivos valores. *)

```
exception Variable_no_asignada of string;;
```

```
type estado = { mutable punto: (float*float); mutable grados: float; mutable lapiz: bool ; mutable var: (string*float) list};;
```

(* Para definir este estado inicial, necesitamos unha función que transforme os grados en radiáns, xa que as funcións do módulo Graphics necesita que estén nesta unidade.

*)

```
let grados_a_Radianes gra =
  let pi = 4. *. atan 1. in (gra *. pi) /. 180.;;
```

(* Agora definimos o que será o estado inicial, co punteiro situado no centro da pantalla, e que será ao que recurrirremos cando queiramos limpar a pantalla e comenzar de novo.

*)

```
let estado_inicial = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true; var = [] };;
let st0 = {punto = (300., 225.); grados = grados_a_Radianes 0.; lapiz = true; var = [] };;
```

(* Agora, a maiores, necesitamos dúas funcións que nos servirán para incluír elementos na lista de valores do estado, así como para obter o número decimal correspondente a un string desa lista. *)

```
let asignar_valor estado s f =
    estado.var ← (s, f):: estado.var ;
    estado;;
```

```
let consultar_valor estado str = match estado with
    [] → raise (Variable_no_asignada "Variable no asignada")
  | (h :: t) → let rec aux l = match l with
      [] → raise (Variable_no_asignada "Variable no asignada")
    | ((s, f) :: t) → if (s = str) then
        f
      else
        aux t
    in aux (h :: t);;
```

(* Para poder acceder as coordenadas do punto por separado, imos implementar dúas funcións 'getters': *)

```
let getx (x,y) = x;;
let gety (x,y) = y;;
```

(* Tipo que define as distintas expresións aritméticas. *)

```
type exp_ari = Const of float
| Suma of (exp_ari * exp_ari)
| Resta of (exp_ari * exp_ari)
| Producto of (exp_ari * exp_ari)
| Division of (exp_ari * exp_ari)
| Mod of (exp_ari * exp_ari)
| Potencia of (exp_ari * exp_ari)
| Opuesto of (exp_ari)
| Var of string;;
```

(* Función encargada de evaluar as expresións aritméticas definidas co tipo anterior. É de tipo '
exp_ari → estado → float'. *)

```
let rec eval_exp estado e = match e with
  Const a -> a
  | Suma (a,b) -> (eval_exp estado a) +. (eval_exp estado b)
  | Resta (a,b) -> (eval_exp estado a) -. (eval_exp estado b)
  | Producto (a,b) -> (eval_exp estado a) *. (eval_exp estado b)
  | Division (a,b) -> (eval_exp estado a) /. (eval_exp estado b)
  | Mod (a,b) -> mod_float (eval_exp estado a) (eval_exp estado b)
  | Potencia (a,b) -> (eval_exp estado a) ** (eval_exp estado b)
  | Opuesto (a) -> 0. -. eval_exp estado a
  | Var s -> consultar_valor estado s;;
```

(* Esta é a función básica para avanzar o punteiro, que recibe como parámetros as coordenadas da posición na que está o punto, a distancia que quere moverse e o ángulo en radiáns, para saber cánto se quere xirar. *)

```
let avanzar (x,y) n radians =
  (sin radians) *. n +. x,
  (cos radians) *. n +. y;;
```

(* Tipo que define expresións booleanas. *)

```
type exp_bool = True
| False
| Not of exp_bool
| And of (exp_bool * exp_bool)
| Or of (exp_bool * exp_bool)
| Equal of (exp_ari * exp_ari)
| Minor of (exp_ari * exp_ari)
| Mayor of (exp_ari * exp_ari)
| MinorEqual of (exp_ari * exp_ari)
| MayorEqual of (exp_ari * exp_ari);;
```

ELENA
DELAMANO
FREIJUE

(* Función encargada de evaluar as expresións aritméticas definidas co tipo anterior. É de tipo 'estado → exp_bool → float'. *)

```
let rec eval_exp_bool estado e = match e with
  True -> true
| False -> false
| Not a -> if ((eval_exp_bool estado a) == true) then
    false
  else
    true
| And (a,b) -> ((eval_exp_bool estado a) && (eval_exp_bool estado b))
| Or (a,b) -> ((eval_exp_bool estado a) || (eval_exp_bool estado b))
| Equal (a,b) -> ((eval_exp_estado a) = (eval_exp_estado b))
| Minor (a,b) -> ((eval_exp_estado a) < (eval_exp_estado b))
| Mayor (a,b) -> ((eval_exp_estado a) > (eval_exp_estado b))
| MinorEqual (a,b) -> ((eval_exp_estado a) <= (eval_exp_estado b))
| MayorEqual (a,b) -> ((eval_exp_estado a) >= (eval_exp_estado b));;
```

(* Tipo que define as distintas instruccións do programa. *)

```
type programa =
  Fd of exp_ari
| Bk of exp_ari
| Rt of exp_ari
| Lt of exp_ari
| Cs
| Pu
| Pd
| Rp of (exp_ari * programa)
| Sec of (programa * programa)
| Make of (string * exp_ari)
| Print of exp_ari
| While of (exp_bool * programa);;
```

(* Agora necesitamos unha función que sirva para executar todas as instruccións definidas no tipo programa. *)

```

let rec ejecutar_estado = function
  Fd (a) -> let _ = let (x,y) = (avanzar_estado.punto (eval_exp estado a) estado.grados) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
  in estado.punto <- ( avanzar_estado.punto (eval_exp estado a) estado.grados);
  estado

  | Bk (a) -> let _ = let (x,y) = (avanzar_estado.punto (eval_exp estado a) (estado.grados +.
  grados_a_Radianes 180.)) in
    if estado.lapiz then
      lineto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
    else
      moveto (int_of_float (x+.0.5)) (int_of_float (y+.0.5))
  in estado.punto <- avanzar_estado.punto (eval_exp estado a) (estado.grados +.
  grados_a_Radianes 180.);
  estado

  | Rt (a) -> estado.grados <- (estado.grados +. (grados_a_Radianes (eval_exp estado a)));
  estado

  | Lt (a) -> estado.grados <- (estado.grados -. (grados_a_Radianes (eval_exp estado a)));
  estado

  | Pu -> estado.lapiz <- false
  estado
  | Pd -> estado.lapiz <- true
  estado
  | Rp (a,b) -> let rec bucle_estado i =
    if (i >= (int_of_float (eval_exp estado a))) then
      estado
    else let estado = (ejecutar_estado b) in bucle_estado (i + 1) in bucle_estado 0

  | Sec (a,b) -> ejecutar (ejecutar_estado a) b
  | Cs -> let _ = clear_graph();
    moveto (int_of_float (getx estado_inicial.punto)) (int_of_float (gety
  estado_inicial.punto));
    estado.punto <- estado_inicial.punto;
    estado.grados <- estado_inicial.grados;
    estado.var <- estado_inicial.var;
    estado.lapiz <- estado_inicial.lapiz;
    in estado_inicial
  | Make (s, e) -> asignar_valor_estado s (eval_exp estado e)
  | Print e -> print_endline (string_of_float (eval_exp estado e));
    estado.grados
  | While (b, e) -> if ((eval_exp bool_estado b) = true) then
    (ejecutar_estado (e);
     ejecutar_estado (While(b,e)))
  else
    estado;;

```

```

open Graphics;;
open Input_program;;
open Abstract_syntax;;
open Logo;;  

exception Instrucion_no_implementada of string;;  

let rec exp_ari_of e = match e with
  Abstract_syntax.Constante a -> Const a
| Abstract_syntax.Variable a -> Var a
| Abstract_syntax.Opuesto a -> Opuesto ((exp_ari_of a))
| Abstract_syntax.Suma (a,b) -> Suma ((exp_ari_of a), (exp_ari_of b))
| Abstract_syntax.Resta (a,b) -> Resta ((exp_ari_of a), (exp_ari_of b))
| Abstract_syntax.Producto (a,b) -> Producto ((exp_ari_of a), (exp_ari_of b))
| Abstract_syntax.Division (a,b) -> Division ((exp_ari_of a), (exp_ari_of b))
| Abstract_syntax.Potencia (a,b) -> Potencia ((exp_ari_of a), (exp_ari_of b))
| Abstract_syntax.Modulo (a,b) -> Mod ((exp_ari_of a),(exp_ari_of b));;  

let rec exp_bool_of e = match e with
  Abstract_syntax.Veradero -> True
| Abstract_syntax.Falso -> False
| Abstract_syntax.Negacion a -> Not ((exp_bool_of a))
| Abstract_syntax.Conjuncion (a,b) -> And ((exp_bool_of a),(exp_bool_of b))
| Abstract_syntax.Disyuncion (a,b) -> Or ((exp_bool_of a),(exp_bool_of b))
| Abstract_syntax.Igualdad (a,b) -> Equal ((exp_ari_of a),(exp_ari_of b))
| Abstract_syntax.LT (a,b) -> Minor ((exp_ari_of a),(exp_ari_of b))
| Abstract_syntax.GT (a,b) -> Mayor ((exp_ari_of a),(exp_ari_of b))
| Abstract_syntax.LET (a,b) -> MinorEqual ((exp_ari_of a),(exp_ari_of b))
| Abstract_syntax.GET (a,b) -> MayorEqual ((exp_ari_of a),(exp_ari_of b));;

```

ELENA
DELAMANO
FREIJE

```

let rec program_of a = match a with
  Abstract_Syntax.Asignar (s,e) -> Make (s, (exp_ari_of e))
  | Abstract_Syntax.Secuencia (e,r) -> Sec (program_of e, program_of r)
  | Abstract_Syntax.Mientras (b, e) -> While (exp_bool_of b, program_of e)
  | Abstract_Syntax.Repetir (a,e) -> Rp (exp_ari_of a, program_of e)
  | Abstract_Syntax.Subir_lapiz -> Pu
  | Abstract_Syntax.Bajar_lapiz -> Pd
  | Abstract_Syntax.Avanzar (a) -> Fd (exp_ari_of a)
  | Abstract_Syntax.Retroceder (a) -> Bk (exp_ari_of a)
  | Abstract_Syntax.Girar_derecha (a) -> Rt (exp_ari_of a)
  | Abstract_Syntax.Girar_izquierda (a) -> Lt (exp_ari_of a)
  | Abstract_Syntax.Mostrar_tortuga -> raise (Instruccion_no_implementada "Instrucción no implementada.")
  | Abstract_Syntax.Ocultar_tortuga -> raise (Instruccion_no_implementada "Instrucción no implementada.")
  | Abstract_Syntax.Borrar_pantalla -> Cs
  | Abstract_Syntax.Set_pen_color (a) -> raise (Instruccion_no_implementada "Instrucción no implementada.")
  | Abstract_Syntax.Set_pen_size (a) -> raise (Instruccion_no_implementada "Instrucción no implementada.")
  | Abstract_Syntax.Print a -> Print (exp_ari_of a);;

```

```

let rec bucle_interactivo estado =
  try
    (ejecutar estado (program_of (get_inst())));
    bucle_interactivo estado
  with
    End_of_program -> print_endline "Goodbye! See you soon!";
      close_graph()
    | Cannot_open_file s-> print_endline "Cannot open file";
      bucle_interactivo estado
    | Invalid_command_in_file -> print_endline "Invalid command in file";
      bucle_interactivo estado
    | Error_lexico -> print_endline "Error lexico";
      bucle_interactivo estado
    | Error_sintactico -> print_endline "Error sintactico";
      bucle_interactivo estado
    | Variable_no_asignada s -> print_endline "Variable no asignada";
      bucle_interactivo estado
    | Instruccion_no_implementada s -> print_endline "Instruccion no implementada";
      bucle_interactivo estado;;

```

```
open_graph "";;
set_window_title "Logo Ocaml P8";;
```

```
set_color blue;;
moveto 300 225;;
```

```
bucle_interactivo st0;;
```

PARADIGMAS DE PROGRAMACIÓN

Grado en Ingeniería Informática

Curso 2013/14

PRÁCTICA 9 (Vector Dinámico)

OBJETIVO:

Se desea implementar en un fichero "vectordinamico.ml" una clase VectorDinamico para gestionar un vector cuyo tamaño aumenta según se van añadiendo elementos (enteros). Internamente se implementa mediante un array. El vector depende de un parámetro tamBloque que indica en cuánto crece el espacio disponible una vez que hemos agotado el actual. Como ejemplo, si tamBloque=5, y añadimos uno a uno al vector los elementos 10, 20, 30, 40, 50 y 60, daría lugar a la siguiente secuencia de estados:

Vista lógica	Vista interna
[]	[_, _, _, _, _]

Inicialmente el vector está vacío (i.e. contiene 0 elementos), pero tiene ya espacio disponible para tamBloque=5 elementos. Ahora según se van añadiendo elementos, éste se va llenando.

[10]	[10, _, _, _, _]
[10, 20]	[10, 20, _, _, _]
[10, 20, 30]	[10, 20, 30, _, _]
[10, 20, 30, 40]	[10, 20, 30, 40, _]
[10, 20, 30, 40, 50]	[10, 20, 30, 40, 50]

En este punto se ha agotado el espacio del primer bloque, por lo que si queremos insertar un nuevo elemento es preciso ampliar previamente la estructura aumentando el espacio disponible en tamBloque=5 posiciones más.

[10, 20, 30, 40, 50]	[10, 20, 30, 40, 50, 60, _, _, _]
----------------------	-----------------------------------

Si inserto un elemento por en medio de los elementos (p.ej. inserto 22 en la posición 2), deberé hacerle hueco desplazando hacia la derecha los elementos de esa posición en adelante.

[10, 20, 22, 30, 40, 50, 60]	[10, 20, 22, 30, 40, 50, 60, _, _, _]
------------------------------	---------------------------------------

Actualmente el vector contiene 7 elementos con espacio para 10.

ATRIBUTOS Y MÉTODOS DE LA CLASE:

```
(* ---- Constructor -----  
  Al hacer new se pasara como unico parametro el tamano de bloque deseado, inicializandose  
  la estructura devuelta a un vector dinamico vacio de tamano-de-bloque elementos. *)  
  
(* ---- Atributos ----- *)  
  
(* Estructura de almacenamiento de los elementos del vector.  
  buffer: int array *)  
(* Numero de elementos que contiene actualmente el vector.  
  numEltos: int *)  
(* Tamano de bloque.  
  tb: int *)  
  
(* ---- Metodos de modificacion ----- *)  
  
(* Anadir un elemento al final del vector actual.  
  annade: int → unit *)  
  
(* Insertar en la posicion i (1er argumento) del vector actual un elemento dado (2o  
  argumento) desplazando en una posicion a la derecha todos los elementos desde el i-esimo  
  hasta el ultimo. Si la posicion no es valida se lanza una excepcion Invalid_argument "index  
  out of bounds".  
  inserta: int → int → unit *)  
  
(* Sobreescribir la posicion i (1er argumento) con un elemento dado (2o argumento). No  
  cambia el numero total de elementos. Si la posicion no es valida se lanza una excepcion  
  Invalid_argument "index out of bounds".  
  sobreescribe: int → int → unit *)  
  
(* ---- Metodos de consulta ----- *)  
  
(* Devolver la longitud actual del vector, es decir, el numero de elementos almacenados.  
  longitud: unit → int *)  
  
(* Devolver el i-esimo elemento del vector. Si la posicion no es valida se lanza una  
  excepcion Invalid_argument "index out of bounds".  
  elemento: int → int *)  
  
(* Devolver la primera posicion por la izquierda en la que aparece un elemento dado. Si no  
  estuviera en el array, entonces devuelve -1 sin mas.  
  indice: int → int  
  
(* Imprimir el contenido del vector en un string (i.e. sólo imprime la parte que contiene  
  elementos)  
  to_string: unit → string *)  
  
(* Imprimir el contenido de la estructura de almacenamiento del vector en un string (i.e.  
  imprime tanto la parte que contiene elementos como la que no contiene elementos, ésta última  
  indicada con un guión bajo por posición)  
  to_stringBuffer: unit → string *)
```

EJEMPLO DE EJECUCIÓN:

```
# #use "vectordinamico.ml";;
class vectordinamico :
  int ->
  object
    val mutable buffer : int array
    val mutable numEltos : int
    val tb : int
    method annade : int -> unit
    method elemento : int -> int
    method indice : int -> int
    method inserta : int -> int -> unit
    method longitud : unit -> int
    method sobreescribe : int -> int -> unit
    method to_string : unit -> string
    method to_stringBuffer : unit -> string
  end

# let prueba = new vectordinamico 5;;
val prueba : vectordinamico = <obj>
# prueba#longitud();;
- : int = 0
# prueba#to_string();;
- : string = "[]"
# prueba#to_stringBuffer();;
- : string = "[_,_,_,_,_]"
# prueba#annade 10;;
- : unit = ()
# prueba#longitud();;
- : int = 1
# prueba#to_string();;
- : string = "[10]"
# prueba#to_stringBuffer();;
- : string = "[10,_,_,_,_]"
# prueba#annade 30;;
- : unit = ()
# prueba#annade 40;;
- : unit = ()
# prueba#inserta 4 20;;
Exception: Invalid_argument "index out of bounds".
# prueba#inserta 1 20;;
- : unit = ()
# prueba#longitud();;
- : int = 4
# prueba#to_string();;
- : string = "[10,20,30,40]"
# prueba#to_stringBuffer();;
- : string = "[10,20,30,40,_]"
# prueba#annade 50;;
- : unit = ()
# prueba#to_string();;
- : string = "[10,20,30,40,50]"
# prueba#to_stringBuffer();;
- : string = "[10,20,30,40,50]"
# prueba#annade 60;;
- : unit = ()
# prueba#to_string();;
- : string = "[10,20,30,40,50,60]"
# prueba#to_stringBuffer();;
- : string = "[10,20,30,40,50,60,_,_,_]"
```

```
# prueba#sobreescrige 1 22;;
- : unit = ()
# prueba#to_string();
- : string = "[10,22,30,40,50,60]"
# prueba#to_stringBuffer();
- : string = "[10,22,30,40,50,60,_,_,_,_]"
# prueba#sobreescrige 100 100;;
Exception: Invalid_argument "index out of bounds".
# prueba#elemento 7;;
Exception: Invalid_argument "index out of bounds".
# prueba#elemento 1;;
- : int = 22
# prueba#indice 30;;
- : int = 2
# prueba#indice 100;;
- : int = -1
# prueba#indice 0;;
- : int = -1
```

(*

Práctica 9. Paradigmas de Programación.

Alumna: Elena M^a Delamano Freije.

Login: elena.m.delamano.freije

Grupo: 1.1.2, Jueves, de 10:30 a 12:30.

*)

open Array;;

(* --- Constructor -----

Al hacer new, se pasará como único parámetro el tamaño de bloque deseado, inicializándose la estructura devuelta a un vector dinámico vacío de tamaño-de-bloque elementos. *)

```
class vectordinamico (tamBloque:int) =  
    object (this)
```

(* ____ Atributos ____ *)

(* Estructura de almacenamiento de los elementos del vector. *)
val mutable buffer = make tamBloque 0

(* Número de elementos que contiene actualmente el vector. *)
val mutable numEltos = 0

(* Tamaño de bloque. *)
val tb = tamBloque

(* ____ Métodos ____ *)

(* Métodos de modificación:

annade: int -> unit
inserta: int -> int -> unit
sobreescribe: int -> int -> unit

Métodos de consulta:

longitud: unit -> int
elemento: int -> int
indice: int -> int
to_string: unit -> string
to_stringBuffer: unit -> string *)

(* ----- *)

(* Añadir un elemento al final del vector actual. *)

```
method annade (el: int) =  
    if ((numEltos <> 0) && ((numEltos mod tb) = 0)) then  
        let aux = make tb 0 in  
            buffer <- append buffer aux  
    else ();  
    buffer.(numEltos) <- el;  
    numEltos <- numEltos + 1
```

(* ----- *)

(* Insertar en la posición i (1er argumento) del vector actual un elemento dado (2o argumento) desplazando en una posición a la derecha todos los elementos desde el i-ésimo hasta el último. Si la posición no es válida, se lanza una excepción Invalid_argument 'index out of bounds'. *)

```
method inserta pos el =  
    if ((pos < 0) || (pos > numEltos -1)) then  
        raise (Invalid_argument "index out of bounds")  
    else  
        (if ((numEltos mod tb) = 0) then  
            let aux = make tb 0 in  
                buffer <- append buffer aux  
        else());  
  
        for j = numEltos downto (pos+1) do  
            buffer.(j) <- buffer.(j-1)  
        done;  
        buffer.(pos) <- el;  
        numEltos <- numEltos + 1 )
```

(* ----- *)

(* Devolver la longitud actual del vector, es decir, el número de elementos almacenados. *)

```
method longitud () = numEltos
```

(* ----- *)

(* Sobreescribir la posición i (1er argumento) con un elemento dado (2o argumento). No cambia el número total de elementos. Si la posición no es válida, se lanza una excepción Invalid_argument 'index out of bounds'. *)

```
method sobreescribe pos el =
    if ( (pos < 0) || (pos > numEltos -1) ) then
        raise (Invalid_argument "index out of bounds")
    else
        buffer.(pos) <- el
```

(* ----- *)

(* Devolver el i-ésimo elemento del vector. Si la posición no es válida, se lanza una excepción Invalid_argument 'index out of bounds'. *)

```
method elemento pos =
    if ( (pos < 0) || (pos > numEltos -1) ) then
        raise (Invalid_argument "index out of bounds")
    else
        buffer.(pos)
```

(* ----- *)

(* Devolver la primera posición, por la izquierda, en la que aparece un elemento dado. Si no estuviera en el array, entonces devuelve -1 sin más. *)

```
method indice el =
    let i = ref 0 in
        while ( (!i <= (numEltos-1)) && (buffer.(!i) <> el)) do
            i := !i+1
        done;

    if !i = numEltos then
        -1
    else
        !i
```

ELENA
DELAMANO
FREIJE

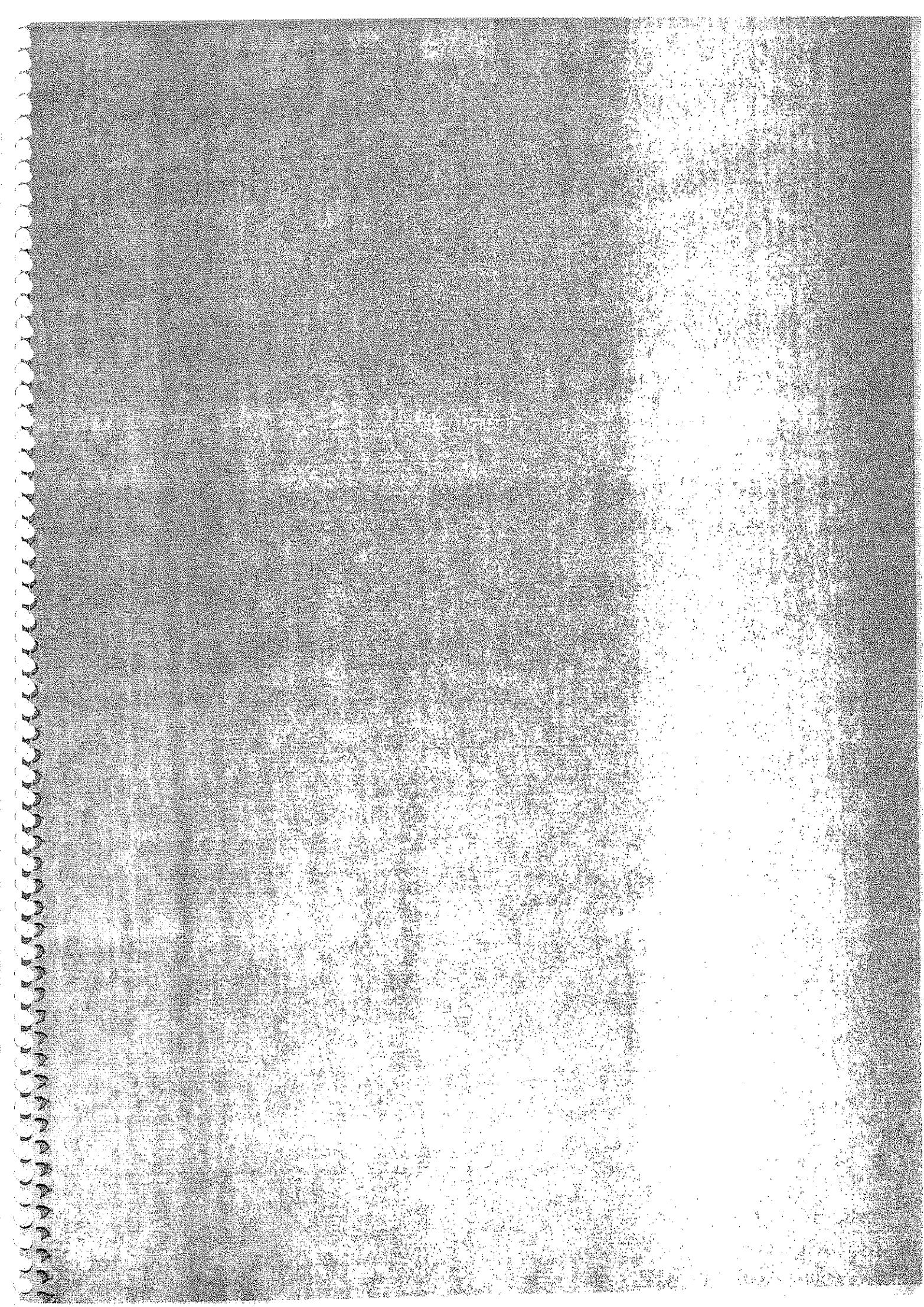
(* ----- *)

(* Imprimir el contenido del vector en un string (i.e. sólo imprime la parte que contiene elementos). *)

```
method to_string () =  
    let rec aux cont =  
        if (cont = numEltos -1) then  
            string_of_int (get buffer cont) ^ aux (cont+1)  
        else  
            if (cont < numEltos) then  
                string_of_int (get buffer cont) ^ "," ^ aux( cont + 1)  
            else "]"  
    in "[" ^ aux 0;  
  
(* ----- *)
```

(* Imprimir el contenido de la estructura de almacenamiento del vector en un string (i.e. imprime tanto la parte que contiene elementos como la que no contiene elementos, ésta última indicada con un guión bajo por posición). *)

```
method to_stringBuffer() =  
    let len = String.length (this#to_string()) in  
    let str= (String.sub (this#to_string()) 0 (len-1)) in  
        (let buffer_str = ref str in  
            if (numEltos > 0) then  
                buffer_str := !buffer_str ^ ","  
            else  
                buffer_str := "[";  
            let buff = length buffer in  
            for i = numEltos to (buff -1) do  
                buffer_str := (!buffer_str ^ "_" ^ ",")  
            done;  
            let len = String.length !buffer_str in  
            (String.sub !buffer_str 0 (len-1)) ^ "]")  
  
end;;
```



Paradigmas de Programación

TGR's

Jueves, 10:30 - 12:30

Para aplicar funciones, simplemente hay que dejar un espacio entre el nombre de la función y al que queremos aplicársela: `char_of_int` espacio.

Las ramas del `IF`, `then` y `else`, en Ocaml son obligatorias. Si no se poner, el programa no compila.

En caso de que no haya "else" se pondrá de la siguiente manera:

```
IF {.....} then {  
  ....  
  } else {  
  } [] ← else → vacío.  
  }
```

La tarea propuesta para este seminario, era la creación de una función “*Upper*”, que tuviera la misma funcionalidad que “*Char.uppercase*”, consistente en la conversión de un carácter tipo letra minúscula a su respectiva mayúscula. A su vez, también había que crear un función “*Lower*” que, en este caso, tuviese la funcionalidad de “*Char.lowercase*”, es decir, la inversa de “*Upper*”.

Para crear esta función, debemos de tener en cuenta el uso de *if – then – else*, de manera que podamos detectar según la tabla ASCII cuándo es mayúscula y cuándo es minúscula. Para explicar mejor todo, en este seminario nos vamos a centrar simplemente en la construcción de “*Upper*”, ya que “*Lower*” se construye de igual manera pero en viceversa alguna frase.

A continuación se muestra el código expuesto por uno de los compañeros, a partir del cual iremos incluyendo o excluyendo cosas hasta hallar un código completo y con un rendimiento adecuado.

```
# let upper = function (n) →
  if int_of_char (n) > 96 && int_of_char (n) < 123 then
    char_of_int (Char.code(n) - 32)
  else
    n;;
```

La posibilidad que nos ha planteado el compañero no está mal, como posible implementación, así a grandes rasgos, sería válida. Pero tiene un problema de eficiencia relacionado con el uso abusivo de “*int_of_char*”. Para solucionar esto podemos hacer un “*let – int*” que coja ese valor, y luego comparamos. Podría ser entonces esta la solución:

```
# let upper = function c →
  let
    n = int_of_char c
  in
    if n > 96 && n < 123 then
      char_of_int (n - 32)
    else
      c
;;
```

ELENA
DELAMANDO
FREIJE

Aunque, si barajamos un poco más la idea, podemos darnos cuenta que ni siquiera hace falta esa obtención de *int*, dado que podemos manejarnos con el carácter directamente de la siguiente manera:

```

# let upper = function c →

    if c >= 'a' && c <= 'z' then
        char_of_int (int_of_char c - 32)
    else
        c
;;

```

Podríamos pensar que la función ya está terminada, pero no hemos tenido presente el resto de símbolos lingüísticos, como la 'ñ', los acentos y la 'ç'. Para arreglar esto ahora, como no queremos que nuestra función sea obtenida de forma visual según la tabla ASCII, implementaremos un bucle “*for*” que recorra toda la tabla y así poder saber a qué caracteres hay que aplicarle la función “*Upper*” y a cuáles no.

```

# for i = 0 to 255 do
let
    c = char_of_int i
in
    if Char.uppercase c <> c then
        print_char c
    else
        0
done;;

```

Con ese trocito del bucle *for*, podemos ver los símbolos que tendremos que cambiar. Ahora tenemos que analizar cuánta distancia hay entre el carácter y su mayúscula, para lo que hacemos un bucle *for* un poco más elaborado:

```

# for i = 0 to 255 do
let
    c = char_of_int i
in
    if Char.uppercase c <> c then
        (print_char c;
         print_char ' ';
         print_int i;
         print_char ' ';
         print_int (int_of_char (Char.uppercase c) - i);
         print_endline "")
    else
        0
done;;

```

Después de ejecutar eso, podemos ver que todos están a una distancia de 32 posiciones, en la tabla ASCII, de su mayúscula, por lo que ahora ya podemos implementar la función definitiva, teniendo en cuenta ahora que tenemos 3 intervalos, no uno sólo como pensábamos al principio:

```
# let upper = function c ->
    if (c>='a' && c<='z') || (c>='\224' && c<='\246') || (c>='\248' && c<='\254') then
        char_of_int (int_of_char c - 32)
    else
        c
;;
```

Pero este resultado es demasiado imperativo. Podemos intentar utilizar la recursividad:

```
# let rec primeros = function n ->
    if n > 0 then
        primeros (n - 1) @ [n]
    else
        []
;;
```

También podemos tratar de hacer el for con una Lista. Esta tendría que empezar en 0.

```
# let rec primeros = function n ->
    if n > 0 then
        primeros (n - 1) @ [n - 1]
    else
        []
;;
```

Para nuestra función necesitamos del 0 al 255, o sea que lo incluimos en una lista "l1".

```
# let l1 = primeros 255;;
```

```
# let l2 = List.map char_of_int l1;;
```

Ahora tendremos una lista de los caracteres, no de los números de la tabla ASCII.

Lo último que queda, es filtrarlos. Haremos lo mismo que con el for anterior, solamente que ahora lo implementaremos con "List.filter":

```
# let l3 = List.filter ( function c → Char.uppercase c <> c ) l2;;
```

Ahora nos quedan simplemente, en la l3, los caracteres a los que le podemos aplicar el uppercase.

```
# let equipo = [ (1, "Casillas") ; (6, "Iniesta") ; (9, "Torres") ; (5, "Puyol") ];;  
  
val equipo : (int * string) list =  
  [(1, "Casillas"); (6, "Iniesta"); (9, "Torres"); (5, "Puyol")]  
  
# List.assoc 9 equipo;;  
- : string = "Torres"
```

Vemos que podemos relacionar los elementos de una lista para seleccionarlos luego con List.assoc mediante la creación de pares de int * string. Es decir, una (int * string) list.

ELENA
DELAMANO
FREIJE

Profesores: Jesús Vilares.

Despacho: 0.20.

Funciones en el módulo List con recursividad no terminal:

- *append (@)*
- *concat*
- *flatter*
- *map*
- *map2*
- *fold_right*
- *split*
- *combine*

Vamos a implementar ahora diferentes funciones, con la intención de que sean recursivas terminales. Empezaremos por la función “*rev*”, de la forma *'a list* → *'a list*.

```
# let rec rev = function
  [] → []
  | h::t → rev t @ [h];;
```

El problema es que, de esta manera, quedan operaciones pendientes (el *@*), por lo que sabemos que no es una función terminal. Intentemos arreglar esto usando constructores o funciones terminales.

```
# let rec rev acum = function
  [] → acum
  | h::t → rev (h::acum) t;;
```

Esta función sí que es terminal, pero habría que llamarla de la siguiente manera: “*rev [] 1*”.

Sería una función de *'a list* → *'a list* → *'a list*, pero este tipo no coincide con el de la función que estamos buscando, por lo que tenemos que hacer un pequeño apañío, y hacemos que una función de orden superior “se lleve el mérito” de la que acabamos de implementar ahora.

Quedaría, definitivamente, de la siguiente manera:

```
# let rev l =
  let rec aux acum = function
    [] → acum
    | h::t → aux (h::acum) t
  in aux [] l;;
```

La diferencia entre @ y :: es muy importante, debemos aprender a diferenciarlas sin problema alguno.

@ es una función de '*a list* → '*a list* → '*a list*', mientras que :: es un constructor de '*a* → '*a list* → '*a list*'. Esta última, en realidad, no se puede representar con flechas, porque no se trata de una función. Simplemente las colocamos para que se entienda que es un constructor que devuelve un '*a list*' a partir de un elemento '*a*' y una '*a list*'.

En ocasiones como:

```
[[ 1 ; 2 ] ; [ 3 ; 4 ]]
```

nos podemos confundir. En este caso, esto es una (*int list*) *list*, es decir, una lista de listas de enteros. Si queremos añadirle [2 ; 5], tenemos que hacerlo con ::, dado que simplemente es un elemento (*int list*).

Pasemos ahora a implementar la función *length*, que es una función que va de '*a list* → *int*'.

```
# let rec length l = match l with
  [] → 0
  | _ :: t → 1 + length ( t );;
```

NO TERMINAL.

ELENA
DELAMANDO
FREIJE

Entonces, en terminal :

```
# let length l =
  let rec aux acum n = match acum with
    [] → n
    | h::t → length t (n+1) ;;
  in aux l 0;
```

El *match* escoge al último. Se podría hacer también como función en vez de con *match – with*. Si reordenamos y ponemos “aux n acum”, cogería el acum como último, se podría quitar y sólo haría falta pasarle el parámetro 0.

```
# let f a b c = match ( a , c ) with
  | __, __ → _____
  | __, __ → _____
  ^
  patrones para a y c.
```

```
# let length lista =
  let rec aux n = function
    [] → n
    | h::t → aux t (n+1)
  in aux 0 lista ;;
```

Ahora sólo tenemos que pasarle el contador, y luego, como se acaban los parámetros de aux, le ponemos el parámetro de la length: la lista.

Función *exists*, que es de ($'a \rightarrow \text{bool}$) $\rightarrow 'a \text{ list}$

Nos indica que recibe una lista de elementos de tipo $'a$, y va comprobando si se cumple la condición ($'a \rightarrow \text{bool}$) en cada uno de sus elementos.

```
exists p [ a1, a2, ..., an ] =  
  ( p a1)  
  | ( p a2)  
  | ( p a3)  
  .  
  .  
  | ( p an)
```

Con OR's, si alguno de ellos lo cumple, ya se nos muestra como cálculo equivalente.

```
# let rec exists p = function  
  [] → false  
  | h::t → if p h then  
            true  
            else  
            exists p t;;
```

Este es un código de una función terminal. No quedan operaciones pendientes.
La llamada a la función *exists* es lo último que se ejecuta en el código.

Si ahora planteamos la solución:

```
# let rec exists p = function  
  [] → false  
  | h::t → ( p h ) || ( exists p t );;
```

También es terminal, debido a que $\langle b_1 \rangle \parallel \langle b_2 \rangle$ es equivalente a

*if $\langle b_1 \rangle$ then true
else $\langle b_2 \rangle$.*

Por lo que el código es equivalente a:

```
if ( p h ) then true  
else ( exists p t );;
```

Por lo tanto, es un if que tiene un comportamiento terminal.
Esto nos da a entender que tenemos que recordar que tanto `&&` como `||` son funciones en cortocircuito.

Por último, implementaremos la función *Filter*, que va de
 $('a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$.

Lo que hace esta función es mantener en orden los elementos de la lista que cumplen la condición.

```
# let filter p l =
let rec aux acum l = function
  [] → rev acum
  | h::t → if (p h) then
    aux (h :: acum) t
  else
    aux acum t
in aux [] l;;
```

Cuando se llama a la función *p*, va a buscarse la última definición de ésta, que en este caso se encuentra en la cabecera de *Filter*, por lo que no es necesario ponerla en la cabecera de *aux*.

Profesores: Jesús Vilares.

Despacho: 0.20

En este TGR se llevarán a cabo las funciones propuestas en las horas de teoría. Estas son:

- La serie de Fibonacci.
- terminal.
- valores muy grandes.
- ejecutable.
- robusto (que detecte errores. Comprobación de entradas).

- El factorial, con las mismas caract que el anterior.
- MergeSort (ordenación por fusión).
- QuickSort.

FIBONACCI

La definición de la serie de Fibonacci es la siguiente:

$$fib(n) = fib(n-1) + fib(n-2)$$

$$\left. \begin{array}{l} fib(0) = 0 \\ fib(1) = 1 \end{array} \right\} \text{- Casos base}$$

La implementación básica, sin tener en cuenta ninguno de los puntos que se piden tratar, sería:

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

```
let fib n = match n with
  0 → 0
  | _ → let rec aux c acum1 acum2
        if (c = n) then
          acum2
        else
          aux (c+1) acum2 (acum1 +/ acum2)

  in aux 1 (num_of_int 0) (num_of_int 1);;
```

↑ Esta función no funcionaría. Daría error de compilación, ya que en la primera rama del pattern matching, la salida es un int, mientras que en la segunda rama, la salida es un num. Para que esto compile sin problemas, simplemente hay que cambiar la primera rama, de manera que:

$0 \rightarrow (\text{num_of_int } 0)$

El operador “ $+/$ ” sirve para sumar dos elementos del tipo “num”. Es un operador infijo. Si lo queremos utilizar como prefijo, tenemos que emplear el “addnum”.

Debemos llamar al Módulo, incluirlo con “**open Num**” ;;

Si no, tenemos que utilizar siempre el nombre del módulo: “**Num.num_of_int**”.

Antes de **open**, hay que cargar la librería → # load “**nums.cma**”.

Ahora que ya tenemos eso implementado, tenemos que hacer la comprobación de las entradas. No se le pueden pasar números negativos a la función:

```
let fib n = match n with
  0 → 0
  | _ → if (n < 0) then
    raise (Failure "Número negativo")
  else
    let rec aux c acum1 acum2
      if (c = n) then
        acum2
      else
        aux (c+1) acum2 (acum1 +/ acum2)
    in aux 1 (num_of_int 0) (num_of_int 1) ;;
```

Aunque, quizás, sea mejor (más autoexplicativo) pasarle esta comprobación en la cabecera de la función, de manera que:

```
let fib n = if (n < 0) then
  raise (Failure "Número negativo")
else
  match n with
  0 → 0
  | _ → let rec aux c acum1 acum2
    if (c = n) then
      acum2
    else
      aux (c+1) acum2 (acum1 +/ acum2)
  in aux 1 (num_of_int 0) (num_of_int 1) ;;
```

En vez de ponerle el “*raise (Failure “Número negativo”)*”, se podía haber puesto directamente la excepción de “*Invalid_Argument*”.

Para que muestre el valor del Fibonacci cuando se le pasan valores altos vamos a necesitar la función “*string_of_num*”, porque llegado a un tope, la función simplemente nos dirá que sí, que ha calculado el valor y que lo tiene almacenado, pero no nos lo mostrará en pantalla. Es mejor implementar esta salida por pantalla en una función aparte en vez de intentar insertarla en la que estamos implementando.

Ahora llegamos al punto en el que necesitamos hacerlo ejecutable:

Sys.argv.(1) → El primer parámetro que se le ha pasado. Es un String, por lo que tenemos que convertirlo a un entero con la función “*int_of_string*”. Si se le pasa una letra, la función devuelve una excepción. Por lo que vamos a intentar capturarla con *try*.

```
try
  let n = int_of_string Sys.argv.(1)
with _ → _ → _ → raise Invalid_Argument ("");
```

Tenemos que pasar el valor a String y usar un printf.

Fichero “fib.ml” → ocalmc → ocalmc -o fib fib.ml
ocamlopt

Al compilar, no va a cargar el *load*. Por lo que tendríamos que poner:

ocamlc → ocalmc -o fib nums.cma fib.ml

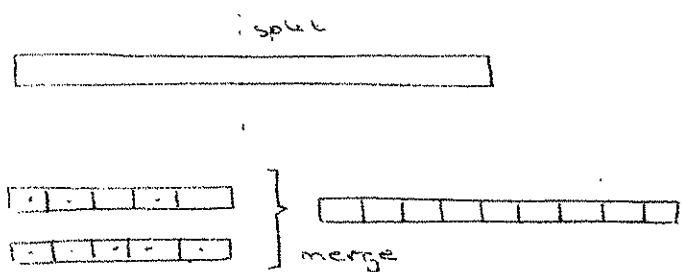
```
let n = try
  int_of_string Sys.argv.(1)
  with _ → raise ...
    ↑ Casos de excepciones
in fib n;;
```

Pueden tratarse por separado o ‘_’

FACTORIAL

Se realiza de la misma manera que la serie de Fibonacci, por lo que lo obviaremos en este seminario.

MERGE SORT



Implementación No-Terminal

ELENA
DELAMANDE
FREIJE

```
let rec split = function
  [] → [], []
  | h :: t → let t1 t2 = split t in
    h :: t1, t2;;
```

Consumo de pila lineal ($O(n)$), desbordamiento con listas de grandes dimensiones.

```
let rec merge l1, l2 = match l1, l2 with
  [], l | l, [] → l
  | h1 :: t1, h2 :: t2 → if (h1 <= h2) then h1 :: merge t1 l2
    else h2 :: merge l1 t2;;
```

Con la función “*merge*” nos encontramos en la misma situación que en “*split*”,
 $O(n)$, desbordamiento de pila.

```
let rec m_sort l = match l with
  [] | [_] → l
  | _ → let l1, l2 = split l in merge (m_sort l1) (m_sort l2);;
```

Esta función no es terminal tampoco ($O(\log_2 n)$).

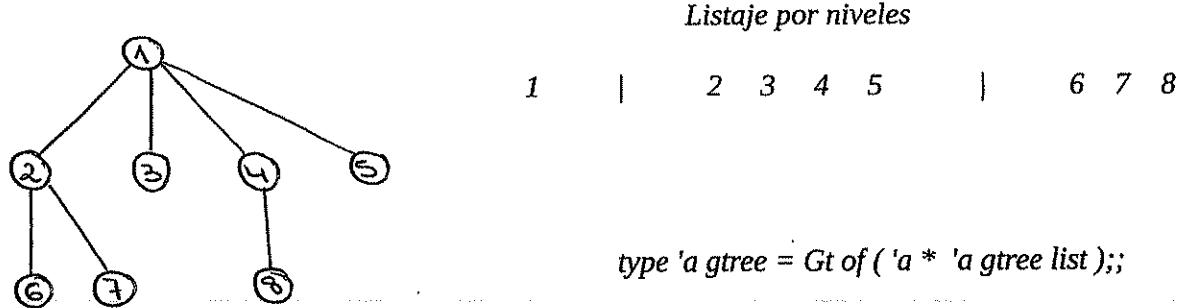
Realmente, no vale la pena pasar la función “*m_sort*” a terminal, ya que consume poca pila, ya que va deshaciendo todo lo que hace.

Lo que sí debemos hacer es pasar las otras dos funciones a terminal, que esas sí que consumen mucho espacio de la pila, pudiendo llegar a causar desbordamiento.

En este TGR se llevarán a cabo tres tareas que se han planteado en las horas de teoría, que son:

- Recorrido en anchura de un árbol general, es decir, con cualquier número de hijos.
- La función de colocar las n reinas.
- Realizar los métodos de ordenación de forma imperativa, y comparar a nivel de complicación con la funcional, al igual que en cuanto a tiempo.

1. Recorrido en anchura de un árbol de tipo 'a gtree.



```
let anchura_gtree_0 t =
    let rec aux = function
        [] → []
        +[Gt (r, l)] → r :: aux l
        | Gt (r, l) :: resto → r :: aux (resto @ l)
    in aux [t];;
```

El segundo caso no es necesario, ya que lo cubre la última comprobación.
La recursividad, aquí no está representada con la pila, si no que con una cola. Esto es debido a que ahora necesitamos que los nodos del siguiente nivel a procesar se pongan a la cola para ser listados.

Ahora vamos a intentar pasar la función a recursividad terminal. Comenzaremos intentando quitarle el “*r :: aux*” con un acumulador.

```

let anchura_gtree_1 arbol =
    let rec aux accum1 = function
        [] → List.rev (accum1)
        | Gt (r, l) :: resto → aux (r :: accum1) (resto @ l)
    in aux [] [arbol];

```

Pero esta función sigue sin ser recursiva terminal. Ahora nos tenemos que encargar de quitarle el “@”. Eso lo arreglamos con otro acumulador más.

```

let anchura_gtree_2 arbol =
    let rec aux accum1 = function
        [] → List.rev (accum1)
        | Gt (r, l) :: resto → aux (r :: l) (List.rev_append
                                            (List.rev t) l)
    in aux [] [arbol];

```

Ahora sí que la función está terminal. Si analizamos los tiempos conseguidos con cada una de las funciones aplicadas a un árbol de 16000 nodos, podemos encontrarnos con:

- *anchura_gtree_0* → 11,19 unidades de tiempo.
- *anchura_gtree_1* → 6,14 unidades de tiempo.
- *anchura_gtree_2* → 10,24 unidades de tiempo.

En este último caso, el tiempo sube debido a las inversiones sucesivas que se realizan con el “List.rev_append”.

Estas funciones tienen, aproximadamente, complejidad cuadrática. Pero ahora queremos tener complejidad lineal. Para ello utilizaremos acumuladores.

```

let anchura_gtree_4 (Gt (r, l)) =
    let rec aux = function
        (ac, [], []) → Lista.rev ac
        | (ac, Gt (r, l) :: t, resto) → aux (r :: ac, t,
                                              List.rev_append (l resto))
        | (ac, [], resto) → aux (ac, List.rev resto, [])
    in aux ([r], l []);

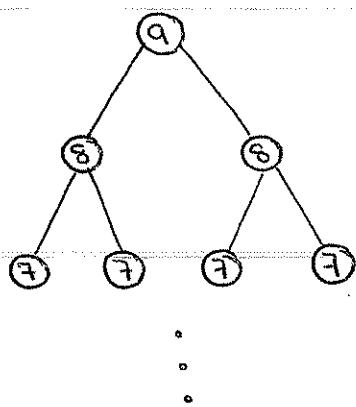
```

Esta función recibirá una terna, compuesta por el resultado parcial * nodos por procesar de nivel actual * nodos por procesar del siguiente nivel.

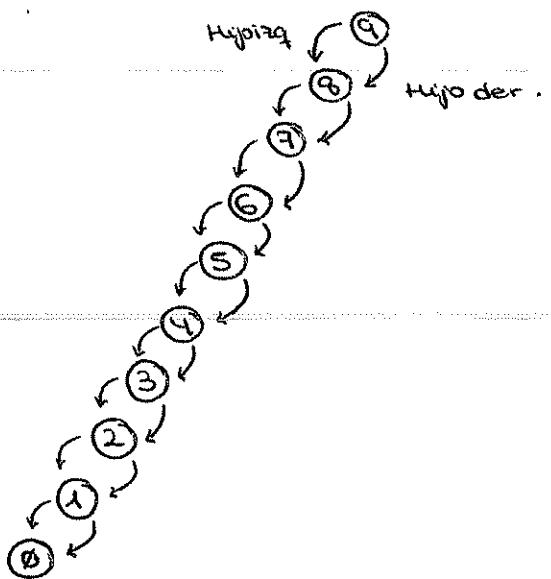
Ahora, la complejidad mejora, y el tiempo pasa a ser de "0,03" unidades de tiempo. Esta mejoría en el tiempo es debida a que ahora simplemente se le da la vuelta ("List.rev_append") una vez por nivel, no siempre, como antes.

Para crear ese árbol de 16000 nodos sin esfuerzo ninguno ni un gran gasto de memoria, utilizaremos la función:

```
let rec gtree_perfect n =
  if (n <= 0) then
    Gt ( 0, [] )
  else
    let p = gtree_perfect (n - 1)
    in Gt (n, [p ; p]);;
```



Pero en
memoria al tratarse
de un lenguaje
funcional



ELENA
DELAMANO
FREIJE

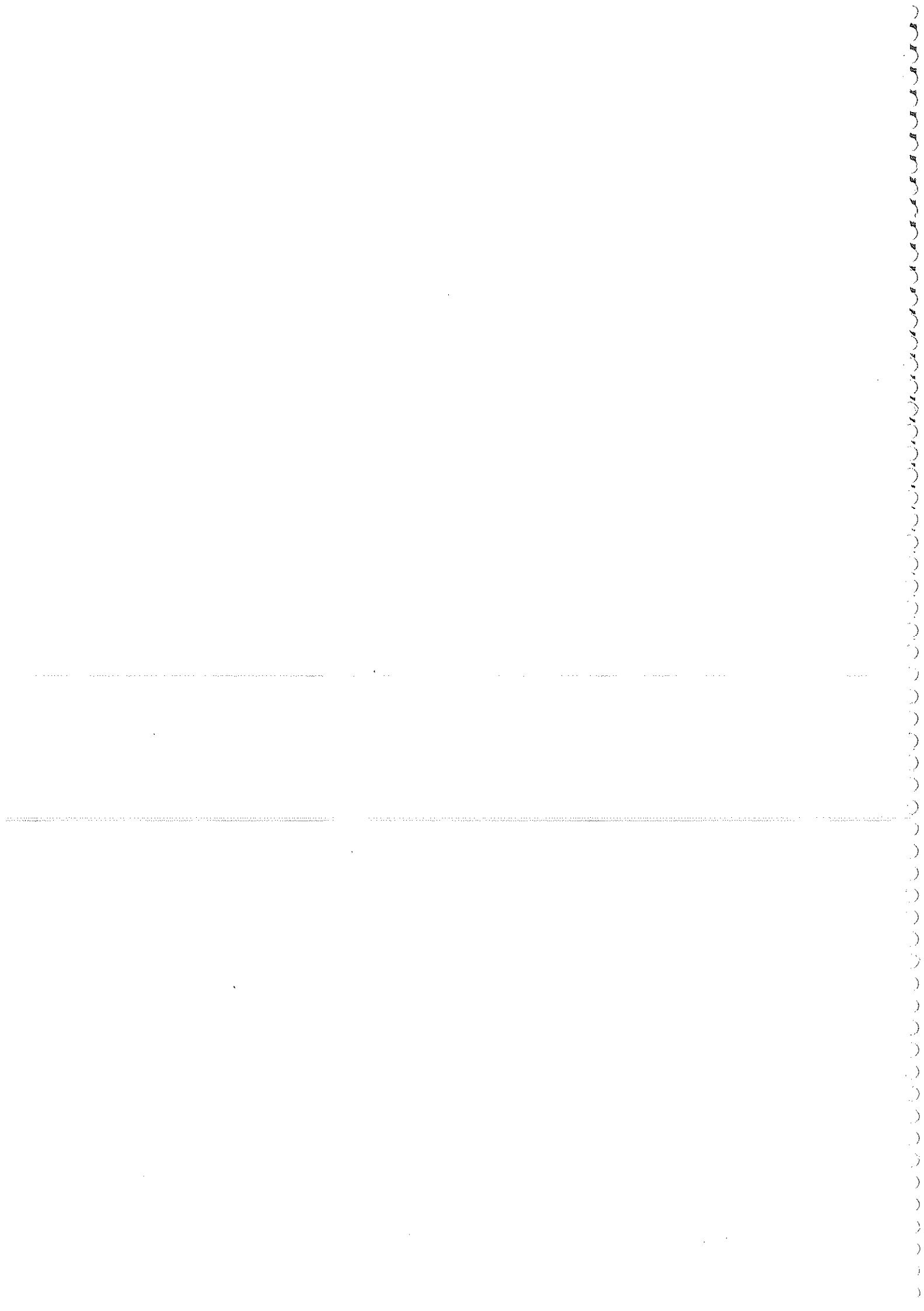
1023 nodos compactos en 10.

2. Función de las N reinas.

Exception No_Solucion;;

```
let come (x1, y1) (x2, y2) =  
  x1 = x2 || y1 = y2 || abs (x2 - x1) = abs (y2 - y1);;
```

```
let reinas n =  
  let rec completa l i j =  
    if (i > n) then  
      l  
    else if (j > n) then  
      raise No_Solucion  
    else  
      if List.exists (come (i, j)) l then  
        completa l i (j+1)  
      else try  
        completa ((i,j)::l) (i+1) 1  
      with No_Solucion → completa l i (j+1)  
  in completa [] 1 1;;
```



```

(** 
@Author Manoel - TutorialesNET
@Archivo fb_tree.ml
**)
type 'a fb_tree = Hoja of 'a | Arbol of 'a * 'a fb_tree * 'a fb_tree;;
let rec string_of_tree f t = match t with
| Hoja x -> "(" ^ f x ^ ")"
| Arbol(x,l,r) -> "(" ^ (f x) ^ " " ^ (string_of_tree f l) ^ " " ^ (string_of_tree f r) ^ ")";
let single x = Hoja x;;
let comp x b = match x, b with
| a, (l,r) -> Arbol(a, l, r);;
let root t = match t with
| Hoja x -> x
| Arbol(x,_) -> x;;
exception Branches
let branches t = match t with
| Hoja _ -> raise Branches
| Arbol(_,l,r) -> l,r;;
(** 
@Author Manoel - TutorialesNET
@Archivo fb_tree_plus.ml
**)
open Fb_tree;;
(* Función Max: max -- Privada *)
let max a b = if a < b then b else a;;
(* Función Is single: is_single *)
let is_single t =
try let _ = branches t in false
with Branches -> true;;
(* Función Left branch: l_branch *)
let l_branch t = fst (branches t);;
(* Función Right branch: r_branch *)
let r_branch t = snd (branches t);;
(* Función Size: size *)
let rec size t =
if is_single t
then 1
else 1 + size (l_branch t) + size (r_branch t);;
(* Función Height: height *)
let rec height t =
if is_single t
then 1
else 1 + max (height (l_branch t)) (height (r_branch t));;
(* Función Preorder: preorder *)
let rec preorder t =
if is_single t
then [root t]
else [root t] @ preorder (l_branch t) @ preorder (r_branch t);;
(* Función Postorder: postorder *)
let rec postorder t =
if is_single t
then [root t]

```

```

        else postorder (l_branch t) @ postorder (r_branch t) @ [root t];;

(* Función Inorder: inorder *)
let rec inorder t =
    if is_single t
    then [root t]
    else inorder (l_branch t) @ [root t] @ inorder (r_branch t);;
(* Función Leafs: leafs *)
let rec leafs t =
    if is_single t
    then [root t]
    else leafs (l_branch t) @ leafs (r_branch t);;
(* Función Mirror: mirror *)
let rec mirror t =
    if is_single t
    then single (root t)
    else let lb = l_branch t and rb = r_branch t in
          comp (root t) (mirror rb, mirror lb);;
(* Función Tree map: treemap *)
let rec treemap f t =
    if is_single t
    then single (f (root t))
    else let lb = l_branch t and rb = r_branch t in
          comp (f (root t)) (treemap f lb, treemap f rb);;
(* Función Is perfect: is_perfect *)
let rec is_perfect t =
    if is_single t
    then true
    else
        if (height (l_branch t) = height (r_branch t)) &&
           (is_perfect (l_branch t) = is_perfect (r_branch t))
        then true
        else false;;
(* Función Is complet: is_complet *)
let rec is_complet t =
    if is_single t
    then true
    else if ((height(l_branch t) = height(r_branch t)) ||
              (height(l_branch t) = (height(r_branch t)+1))) &&
          ((List.length(leafs(l_branch t))) >= (List.length(leafs(r_branch t)))) &&
          ((is_complet(l_branch t)) = (is_complet(r_branch t)))
    then true
    else false;;
(**

@Author Manoel - TutorialesNET
@Archivo lista.ml
**)

(* Función Head: hd *)
let hd = function
    h::_ -> h

```

```
| _ -> aux r (l-1) ((Random.int r)::list)
in aux r [];;
(* _____ *)
```

```
(* s_sort - es terminal*)
let s_sort = function
  [] -> []
  | h::t ->
    let rec aux res m = function
      ([],[]) -> m::res
      | (h::t,[]) -> aux (m::res) h ([]),t)
      | (l,h::t) -> if h <= m
        then aux res m (h::l,t)
        else aux res h (m::l,t)
    in aux [] h ([]),t);;
```

(** Conclusiones sobre s_sort

[vector aleatorio] -> tiempos aumentan exponencialmente
[vector ascendente] -> tiempos aumentan exponencialmente (los tiempos son peores que cuando al entrada es aleatoria)
[vector descendente] -> tiempos aumentan exponencialmente

**)

```
(* insert - No es terminal*)
let rec insert x = function
  [] -> [x]
  | h::t -> if x <= h then x::h::t
              else h::insert x t;;
```

```
(* i_sort - No es terminal*)
let rec i_sort = function
  [] -> []
  | h::t -> insert h (i_sort t);;
```

(** Conclusiones sobre i_sort

[vector aleatorio] -> estamos en el caso medio, tiempos cuadráticos
[vector ascendente] -> estamos en el mejor caso, tiempos lineales (el caso mejor entre los 2 métodos de ordenación)
[vector descendente] -> estamos en el peor caso, tiempos cuadráticos

**)

(** Datos obtenidos de tests

[Listas no ordenadas]
[-] ENTRADA: 1000, doble de diferencia, en tiempos
let aleatoria = rlist 1000 1000;;
crono s_sort aleatoria;;

```

- : float = 0.0560040000000299187
# crono i_sort aleatoria;;
- : float = 0.024000999999983847

[-] ENTRADA: 10000; casi cuádruple, en tiempos
[-] Si aumentamos la entrada, s_sort empeora
[-] respecto a i_sort.
# let aleatoria = rlist 100 10000;;
# crono s_sort aleatoria;;
- : float = 5.060316
# crono i_sort aleatoria;;
- : float = 1.61610100000001466

```

[Listas ordenadas ascendenteamente]

```

[-] i_sort simplemente recorre la lista
[-] sin realizar ninguna inserción
# let ascendente = s_sort (rlist 100 10000);;
# crono s_sort ascendente;;
- : float = 4.94430899999997564
# crono i_sort ascendente;;
- : float = 0.
# crono i_sort ascendente;;
- : float = 0.0040000000001909939

```

[Listas ordenadas descendenteamente]

```

[-] En caso de tener una lista ordenada
[-] descendente (o sea, insertar siempre en
[-] la primera posición).
# let i = s_sort_desc (rlist 100 10000);;
# crono i_sort i;;
- : float = 3.32020799999997962
# crono s_sort i;;
- : float = 4.86030299999998761

```

**)

(* >> Conclusiones sobre s_sort (terminal) e i_sort (no terminal

s_sort es recursiva terminal pero, sin embargo, i_sort no lo es.
insert tampoco es recursiva terminal

Sí, supone la limitación de la pila. En ella se van cargando las operaciones recursivas y si excedemos el límite obtendremos un error de "Stack Overflow" (desbordamiento de la pila). *)

```

(* t_insert - Terminal *)
let t_insert x l =
  let rec aux x l1 l2 = match l1,l2 with
    [] ,[] -> [x]
  | [],l2 -> List.rev_append l2 [x]

```

```

method longitud() = numEltos

method incNumEltos() = numEltos <- numEltos + 1

method annade e =
    if (self#longitud() < 0) && (((self#longitud()) mod tb) = 0)
    then let t = make tb 0 in
        buffer <- append buffer t
    else ();
    buffer.(self#longitud()) <- e;
    self#incNumEltos()

method inserta i e =
    if (i < 0) || (i > self#longitud() - 1)
    then raise(Invalid_argument "index out of bounds")
    else begin
        if self#longitud() mod tb = 0
            then let t = make tb 0 in
                buffer <- append buffer t
            else ();
        for j=self#longitud() downto (i+1) do
            buffer.(j) <- buffer.(j-1)
        done;
        buffer.(i) <- e
    end;
    self#incNumEltos()

method sobreescribe i e =
    if (i < 0) || (i > self#longitud() - 1)
    then raise(Invalid_argument "index out of bounds")
    else (buffer).(i) <- e

method elemento i =
    if (i < 0) || (i > self#longitud() - 1)
    then raise(Invalid_argument "index out of bounds")
    else buffer.(i)

method indice e =
    let i = ref 0 in
        let tam = self#longitud() in
            while (!i <= (tam-1)) && (buffer.(!i) <> e) do
                incr i
            done;
            if !i = self#longitud()
            then -1
            else !i

method to_string () =
    let buffer_str = ref "[" in
        let tam = self#longitud() in
            for i=0 to (tam-1) do

```

```

| h1::t1,_ -> if x <= h1
then List.rev_append (List.rev_append
(x::h1::t1) l2) []
else aux x t1 (h1::l2)
in aux x l [];

```

(* t_i_sort - Terminal - NO es más rápida que la versión no terminal*)

```

let t_i_sort l =
  let rec aux l temp = match l, temp with
    [],_ -> temp
    | h::t,temp -> aux t (t_insert h temp)
  in aux l [];

```

(* >> Ventajas e inconvenientes de t_i_sort con respecto a i_sort

t_i_sort es terminal, por tanto no requiere almacenar los distintos cambios de contexto que se producen en las distintas llamadas recursivas hechas por i_sort. Es por ello que nos preveemos de un desbordamiento de pila. Sin embargo, la implementación de t_i_sort que yo he hecho es más lenta que la de i_sort propuesta y por ello es una desventaja.

```

# let lista = rList_t 10001 10000;;
# crono i_sort lista;;
- : float = 1.9161199999999982
# crono t_i_sort lista;;
- : float = 4.58028600000000097
# (crono t_i_sort lista) < (crono i_sort lista);;
- : bool = false
      *)

```

```

(** @Author Manoel - TutorialesNET
@Archivo vectordinamico.ml
**)

```

```
open Array;;
```

```

class vectordinamico tamBloque =
  object (self)

    (* __ Atributos __ *)
    val mutable buffer = make tamBloque 0
    val mutable numEltos = 0
    val tb = tamBloque

    (* __ Métodos __ *)
    method private incBuffer v =
      let t = make tb 0 in
        buffer <- append v t

```

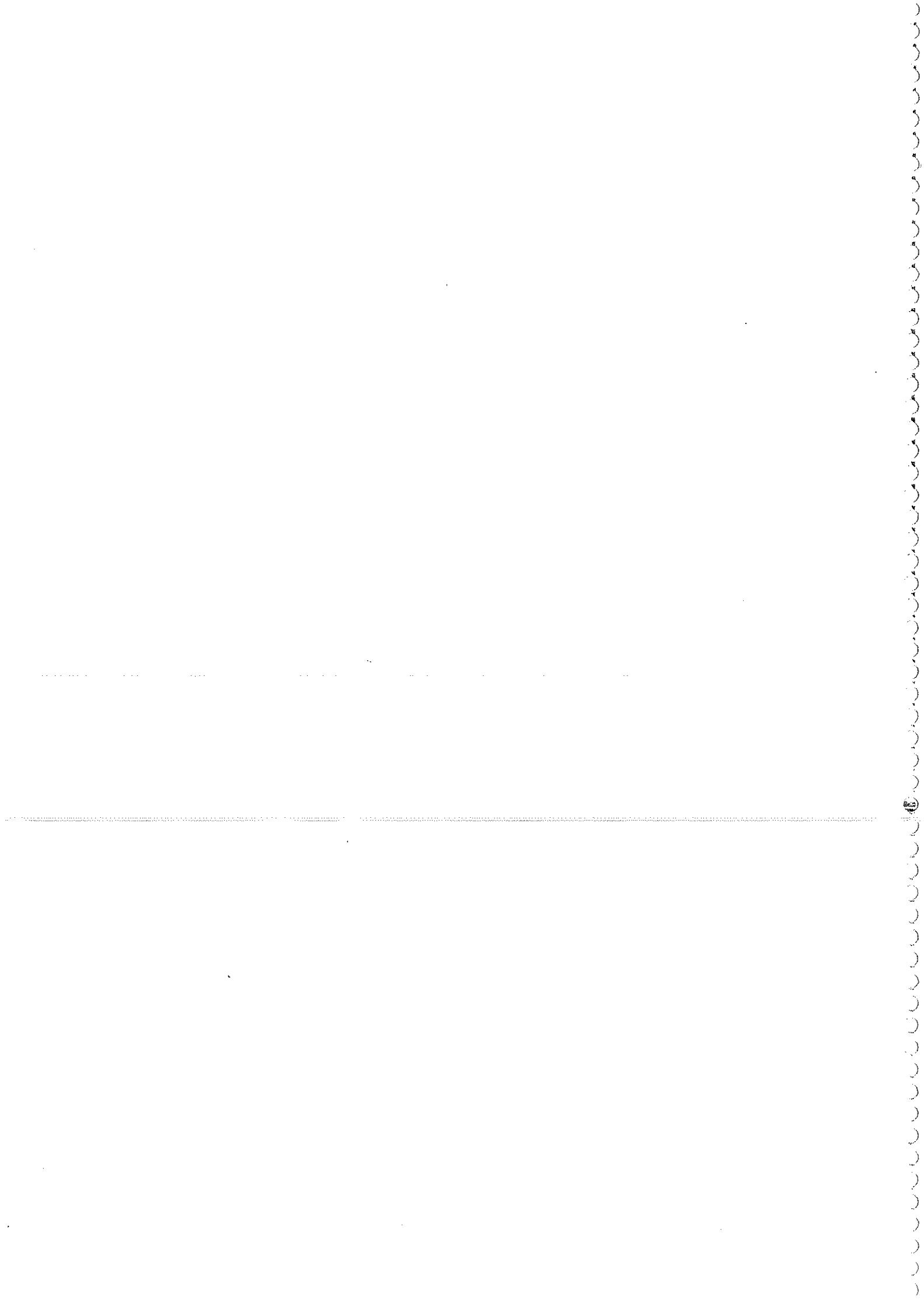
```

                                buffer_str := (!buffer_str ^
string_of_int buffer.(i) ^ ",")

done;
if numEltos = 0
then !buffer_str ^ "]"
else let len = String.length !buffer_str in
      (String.sub !buffer_str 0 (len-1))^ "]";

method to_stringBuffer () =
let len = String.length(self#to_string()).in
let str = (String.sub (self#to_string()) 0 (len-1)) in begin
let buffer_str = ref str in
if numEltos > 0
then buffer_str := !buffer_str ^ ","
else buffer_str := "[";
let buff = length buffer in
for i=numEltos to (buff - 1) do
  buffer_str :=
done;
let len = String.length !buffer_str in
(String.sub !buffer_str 0 (len-
1))^ "]";
end;
end;;

```



```

(*-----)
----- FUNCION FROMTO RECURSIVA
----- *)
let rec fromto m n = if m>n then []
                      else m::fromto (m+1) n;;
;

(*-----)
----- FUNCION fromto_t RECURSIVA TERMINAL
----- *)
let fromto_t m n =
    let rec aux x acum=
        if x<m then acum
        else aux(x-1) (x::acum)
    in aux n []
;;
;

(*-----)
----- FUNCION lmax
----- *)
let rec lmax = function
    h::[]->h
  | h::t-> if h>=lmax t then h
            else lmax t
  | _->raise(Invalid_argument "lmax")
;;
;

(*-----)
----- FUNCION lmax_c corregida
----- *)
let rec lmax_c = function
    h::[]->h
  | h::t-> max h (lmax_c t)
  | _->raise(Invalid_argument "lmax")
;;
;

(*-----)
----- FUNCION lmax_t RECURSIVIDAD TERMINAL
----- *)
let lmax_t l=
    let rec aux acum= function
        []->acum
      | h::h2::t->if (h2>h) then aux h2 t
                  else aux h t
      | _->raise(Invalid_argument "lmax_t")

```

```

        in aux (List.hd(l)) l
;;
(*-----
-----*)
FUNCTION sufaxes
-----*)

let rec sufaxes l = match l with
    []->[]
  | _::t->l::sufaxes t
;;
(*-----*)

(*-----
-----*)
FUNCTION prefixes
-----*)

let rec prefixes=function
    []->[]
  | h::t->[h]::List.map(function l->h::l) (prefixes t)
;;
(*-----*)

(*-----
-----*)
FUNCTION segments
-----*)

let segments l=List.concat(List.map prefixes (sufaxes l))
;;
(*-----*)

(*-----
-----*)
FUNCTION is_sufix
-----*)
let is_sufix l1 l2=List.mem l1 (sufaxes l2)
;;
(*-----*)

(*-----
-----*)
FUNCTION is_prefix
-----*)
let is_prefix l1 l2=List.mem l1 (prefixes l2)
;;
(*-----*)

(*-----
-----*)
FUNCTION is_segment
-----*)
let is_segment l1 l2 =List.mem l1 (segments l2);;

```

```
(*-----  
FUNCTION is_sublist  
-----*)  
let rec is_sublist l1 l2=match l1,l2 with  
    [],_ ->true  
  | h1::t1, h2::t2-> if h1=h2 then is_sublist t1 t2  
    else is_sublist l1 t2  
  |_ ->false  
;;  
(*-----  
FUNCTION sublists  
-----*)  
let rec sublists =function  
    [] -> [[]]  
  | x::l -> let aux = sublists l  
    in aux @ (List.map (fun l -> x::l) aux)  
;;
```

```

-----*)
-----*)

let rec rev=function
[]->[]
| h::t->append(rev t)[h];;

(*-----*)

FUNCION rev_append
-----*)
-----*)

let rec rev_append l1 l2 =match l1 with
[]->l2
| (h::t)->rev_append t (h::l2);;

(*-----*)

FUNCION concat
-----*)
-----*)

let rec concat = function
[]->[]
| h::t->append h(concat t);;

(*-----*)

FUNCION flatten
-----*)
-----*)

let flatten = concat;;;

(*-----*)

FUNCION map
-----*)
-----*)

let rec map f l = match l with
[]->[]
| h::t->f(h)::map f t;;;

(*-----*)

FUNCION map2
-----*)
-----*)

let rec map2 op l1 l2=
    if(length l1 !=length l2) then
        raise(Invalid_argument "map2")
    else
        if(length l1==0 || length l2==0) then
            []
        else

```

c) (1 pto.) Redefina la función ffac utilizando sólo recursividad terminal:

Hecho
180

```
let ffac n =  
  
    let rec aux cont = function  
        0 → cont  
        1 → cont  
        l n → aux (cont * n) (n-2)  
  
    in aux 1 n;;
```

3. La siguiente definición sirve para representar árboles binarios:

```
type arbb = Vacio | Nodo of arbb * arbb;;
```

a) (1 pto.) Defina una función altura : arbb -> int que, para cada árbol binario, devuelva su altura. Llamamos "altura" de un árbol a la distancia de la raíz a la hoja más lejana (contada como el número de nodos que hay que recorrer para llegar hasta ella). (La altura del árbol vacío es 0 y la de un árbol que sólo tenga raíz es 1).

```
let rec altura = function  
    Vacio → 0  
    | Nodo (l, r) → 1 + max (altura l) (altura r);;
```

b) Defina una función tamanno : arbb -> int que, para cada árbol binario, devuelva su tamaño. Llamamos "tamaño" de un árbol al número de nodos que contiene. (El tamaño del árbol vacío es 0 y el de un árbol que sólo tenga raíz es 1).

```
let rec tamanno = function  
    Vacio → 0  
    | Nodo (i, d) → 1 + (tamanno i) + (tamanno d);;
```

PROGRAMACIÓN DECLARATIVA

5 de septiembre de 2002

NOMBRE:

DNI:

I.I. I.T.I.G

Programación funcional

1. (2 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
# let x = 0;;
```

```
val x : int = 0
```

```
# (let x = x + 1 in 2 * x), 2 * x;;
```

-: int * int = (2,0)

```
# x;;
```

-: int = 0

```
# let f x = x, x in f x;;
```

-: int * int = (0,0)

```
# let doble f = function x -> f (f x);;
```

val doble : ('a -> 'a) -> 'a -> 'a = <fun>

2. Dada la siguiente definición:

```
# let rec ffac = function
  0 -> 1 | 1 -> 1
  | n -> n * ffac (n-2);;
```

- a) (0.5 ptos.) Indique su tipo:

val facc : int -> int = <fun>

- b) (0.5 ptos.) Indique el tipo y el valor de la siguiente expresión:

```
# ffac 6;;
```

-: int = 48

2. (2 ptos.) Escriba una definición recursiva terminal de la siguiente función f : int \rightarrow int

```
let rec f x = if x >= 4 then 3 * f(x-1) - 2 * f(x-3)
               else x;;
```

3. (2 ptos.) Escriba una definición recursiva terminal de la función
lista2arbol: 'a list \rightarrow 'a arbol:

```
let rec lista2arbol = function
    []  $\rightarrow$  arbol_vacio
    | h::t  $\rightarrow$  insert h (lista2arbol t);;
```

donde arbol_vacio: 'a arbol y insert: 'a \rightarrow 'a arbol \rightarrow 'a arbol.
(Puede suponerse que la función insert está definida de modo terminal).

let lista2arbol l =

let rec aux acum lista =

[] \rightarrow acum

| h::t \rightarrow aux (insert h acum) t

in aux [] (arbol_vacio) l;

PROGRAMACIÓN DECLARATIVA
16 de diciembre de 2002

NOMBRE: _____ DNI: _____

Ingeniería Técnica en Informática de Gestión Ingeniería Informática

1. (3 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
# let f x = x + 1, x - 1;;
val f : int → int * int = <fun>

# let x = f 0;;
val x : int * int = (1, -1)

# let y, x = x and z = x;;
val y : int = 1
val x : int = -1
val z : int * int = (1, -1)
# let x y = y::[] in x y;;
val x y : int list = [1]

# let mas f g = function (x,y) -> f x + g y;;
val mas : ('a → int) → ('b → int) → ('a * 'b) → int = <fun>

# let x = let x = [1;2;3] in List.tl x;;
val x : int list = [2;3]
```

```

# let div (x,y) = (/) x y;;
val div : int * int → int = <fun>

# let div =
  val div : _ = ... ;;

# let max (p,ord) = if ord p then snd p else fst p;;
val max : ('a * 'a) * ('a * 'a → bool) → 'a = <fun>

# let max =
  val max : _ = ... ;;

```

ELENA
DELAMANO
FREIJE

3. (1.5 ptos.) Un valor de tipo `(float * float) list` puede representar un camino poligonal en el plano (sería la lista ordenada de las coordenadas de los vértices del camino). Defina una función `distancia`: `(float * float) list → float` que calcule la longitud de cada uno de estos caminos. Recuerde que la distancia entre dos puntos, (x_1, y_1) y (x_2, y_2) , viene dada por la fórmula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

`let distancia`

PROGRAMACIÓN DECLARATIVA
11 de diciembre de 2001

NOMBRE: _____ DNI: _____

Programación funcional

1. (2.5 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
# let x y = y + 1, y - 1;;
val x : int → int * int = <fun>

# let y = x 0;;
val y : int * int = (1, -1)

# let x,y = y and y = 0;;
Error: Variable y is bound several times in this matching.

# let rec fold2 op1 op2 e = function
  h :: t -> op1 h (fold2 op2 op1 e t)
| [] -> e;;
val fold2 : ('a → 'b → 'b) → ('a × 'b × 'b) → 'b → 'a list → 'b = <fun>

# fold2 (+) (* ) 0 [1;2;3;4], fold2 (+) (-) 1 [1;2;3;4];;
-: int * int = (7, -3)
```

2. (1.5 ptos.) Indique el tipo de las funciones definidas e continuación y redefinalas en modo "curry" de la forma más breve que pueda.

```
# let op = function p -> fst p (snd p);;
val op : ('a → 'b) * ('a → 'b) = <fun>

# let op =
val op :
```

```

let lsuma l =
  let rec aux cont l = match l with
    [] → cont
    | h::t → aux (cont +. h) t
  in aux 0. l ;;

```

```

let lproducto l =
  let rec aux cont l = match l with
    [] → cont
    | h::t → aux (cont *. h) t
  in aux 1. l ;;

```

ELENA
DELAMANO
FREIJE

3. a. Indique el tipo de las funciones `filtro` y `no_pertenece_a`, definidas a continuación.
 b. Utilizando las funciones `filtro` y `no_pertenece_a`, escriba una definición lo más breve posible de la función `diferencia`: `'a list -> 'a list -> 'a list`, de forma que `diferencia l1 l2` sea la lista de los elementos de `l1` que no están en `l2`.

```

let rec filtro f =
  function [] -> []
    | h::t -> if f h then h::filtro f t
                else filtro f t;;

val filtro : ('a → bool) → 'a list → 'a list = fun

let rec no_pertenece_a =
  function [] -> (function _ -> true)
    | h::t -> (function x -> x <> h & no_pertenece_a t x);;

val no_pertenece_a : 'a list → 'a → bool = fun

let diferencia l1 l2 = match l1, l2 with
  ([], []) → []
  | ([], t) → t
  | ((l, []), t) → l
  | ((l, r), t) → filtro (no_pertenece_a t) l;;

```

```

let rec f = function n -> n * f (n-2) | 1 -> 1 | 0 -> 0;;
val f : int → int = <fun>

f 4 + f 5;;
stack overflow during evaluation (looping recursion?)

let f = function 0 -> 0 | 1 -> 1 | n -> n * f (n-2);;
val f : int → int = <fun>

f 4 + f 5;;
stack overflow during evaluation (looping recursion?)

let rec ap op = function [] -> []
| cab::col -> op cab :: ap op col;;
val op: ('a → 'b) → 'a list → 'b list = <fun>

let pri (a,_) = a;;
val pri: ('a * 'b) → 'a = <fun>

ap pri;;
-: ('a * 'b) list → 'a list = <fun>

ap pri [([],[],[])];;
-: 'a list list = [<fun>]

```

ELENA
DELAMANO
FREIJE

2. Defina las funciones `ldoble`, `lcuadrado` y `lcubo` de forma que aplicadas a una lista de flotantes den, respectivamente, la lista de sus dobles, de sus cuadrados y de sus cubos. Y defina las funciones `lsuma` y `lproducto` de forma que aplicadas a una lista de flotantes den, respectivamente, la suma y el producto de todos sus elementos.

```

let rec ldouble l = match l with
[] → []
| h::t → h *. 2.0 :: ldouble t;;

```

```

let rec lcuadrado l = match l with
[] → []
| h::t → h ** 2.0 :: lcuadrado t;;

```

```

let rec lcubo l = match l with
[] → []
| h::t → h ** 3.0 :: lcubo t;;

```

let f11 (x: SubclaseR1) => metodo1();

let f12 (x: ~~#~~SubclaseR1) => ~~#~~metodo2();;

• Cuál aceptaría una instancia de ClaseP?

Class [a, b] ClaseT =

Object (this)

val mutable l1 = []

val mutable l2 = []

method get_l1 = l1

se \equiv l1 pm1 = l1 \leftarrow pm1

add_l1 pm1 = l1 \leftarrow pm1 :: l1

reset_l1 () = l1 = []

||

mismo para l2

Class [c, d] SubclaseT (pc1: 'd) =

Object (this)

inherit [int + 'c, 'd] ClaseT as Super

initializer this # add_l2 pc1

let st1 = new SubclaseT ("bono");

st1 # set_lists();

st1 # add_l1 (5, "1"); st1 # add_l2 "guerda";

st1 # get_lists();

st1 # reset_l1(); st1 # reset_l2();

st1 # get_lists();

st1 # set_l1 [5, 0.0]; st1 # set_l2 ("");

st1 # get_lists();

PROGRAMACIÓN DECLARATIVA
19 de diciembre de 2000

APELLIDOS: _____

NOMBRE: _____ DNI: _____

Programación Funcional

1. Las siguientes frases corresponden al código introducido en una sesión de trabajo con el toplevel de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
let x = [1;2;3] in 4::x;;
-: int list = [4;1;2;3]

List.hd x;;
Error: Unbound value x.

let x = let x = [1;2;3] in List.tl x;;
val x : int list = [2;3]

let y::x = x;;
val y : int = 2
val x : int list = [3]

let (y,x) = (x,y::[]) in y@x;;
-: int list = [3;2]

let x y = y::[] in x y;;
-: int list = [2]

(fraction f -> f (f 2.0)) (function f -> f ** f);;
-: float = 256.

let rec f n = if n > 1 then n * f (n-2) else n;;
val `f: int → int = <fun>
4 + f 5;;
-: int = 15.
```

Indicar si son o no correctas

Class clase01 (n:int)(c:char) =

Object

val atr1=n

val atr2=c

val atr3=[1;atr2;atr2;atr2]

method met1()=atr1

method met2(p1:n;p2:c)=Attr2, where p1=p2

method met3()=[1;atr2;atr2;atr2]

end;

Class clase03 (n:int) ((c:char)=

Object

val atr1=n

val atr2=c

val mutable atr3=[1;1]

initializar atr3 <--> Attr3, make atr1 atr2

method met1()=atr1

method met2 p1 (p2:n;c:char)=Attr3, make p1 p2

method met3()=[1;atr2;attr2;attr2]

end;

Class clase02 =

object (this)

val mutable atr1=[1;2;3;4;5]

method metodo1()=atr1

method metodo2()=list.hd (this) # metodo1()

.. metodo3()=list.tl (this) # metodo1()

end;

Class subclase01 pcd =

object (this)

inherit clase02 as super,

val mutable atr1=pcd

method metodo1()=super.metodo1() # Attr1

end;



```

| h::t-> (not(mem h t)) && (is_legal_move h (hd t)) &&
(are_chained_moves t)
| _-> false;;
```

```

let is_knight_tour m n l =
  (length l = m * n) &&
  (for_all (is_in_board m n) l) &&
  (are_chained_moves l);;
```

```

(*******)
(*knight_tour : int -> int -> (int * int) -> (int * int) list*)
(*******)
```

```

let notmem l e =
  not (mem e l);;
```

```

let legal_moves m n (x,y) visited =
  let
    all_moves = [x+1,y+2;x+2,y+1;x-1,y+2;x-2,y+1;x+1,y-1;x+2,y-2;x-1,y-
2;x-2,y-2]
  in
    filter (notmem visited) (filter(is_in_board m n) all_moves);;
```

```

exception No_knight_tour;;
```

```

let knight_tour m n (x,y) =
  if (m<0) || (n<0) || (not(is_in_board m n (x,y))) then raise
  (Invalid_argument "knight_tour")
  else
    let
      num_cells = m*n
    in
      let rec aux i visited moves = match (i,visited,moves) with
        (0,_,_)-> raise No_knight_tour
        |(i,vs,_) when i = num_cells -> rev vs
        |(i,_::vs,[)::ms)-> aux(i-1) vs ms (*back tracking*)
        |i,vs,(c::cs)::ms)->
          aux (i+1) (c::vs) ((legal_moves m n c vs)::cs::ms)
      in
        aux 1 [(x,y)] [(legal_moves m n (x,y) [])];;
```

```

(*Knight_tour*)

let valida n m(x,y) = x >= 1 && x <= n && y <= n;;

let candidatas m n ls (x,y) =
  let rec candidatas lc = function
    []-> lc
    | h::t->
      if (valida m n h) && not (List.mem h ls) then candidatas(h::lc)t
      else candidatas lc t
  in candidatas[] [x+1,y+2;x+2,y+1;x-1,y+2;x-2,y+1;x+1,y-2;x+2,y-1;x-1,y-2;x-2,y-2];;

let is_knight_tour m n l =
  if (m<1) then raise(Invalid_argument "m")
  else if(n<1) then raise (Invalid_argument "n")
  else if m*n != List.length l then false
  else if not (valida m n (List.hd l)) then false
  else let rec is_knight_tour l ls = match l with
    h::t->
      if not(List.mem h(candidatas m n ls(List.hd ls))) then false
      else is_knight_tour t(h::ls)
    | []-> true
  in is_knight_tour (List.tl l) (List.hd l);;

let knight_tour m n c =
  if (m<1) then raise(Invalid_argument "m")
  else if(n<1) then raise (Invalid_argument "n")
  else if not(valida m n c) then raise (Invalid_argument "(x,y)")
  else let rec knight_tour ls lp = match lp with
    hp::tp->
      if ls != [] && hp=(List.hd ls) then knight_tour (List.tl ls)
    tp
      in if lc = [] then
          if (m*n)= List.length (hp::ls) then List.rev (hp::ls)
          else knight_tour ls tp
          else knight_tour (hp::ls) (lc @ lp)
        | []-> raise (Failure "knight_tour")
      in knight_tour [] [c];;

(*caballo*)

open List;

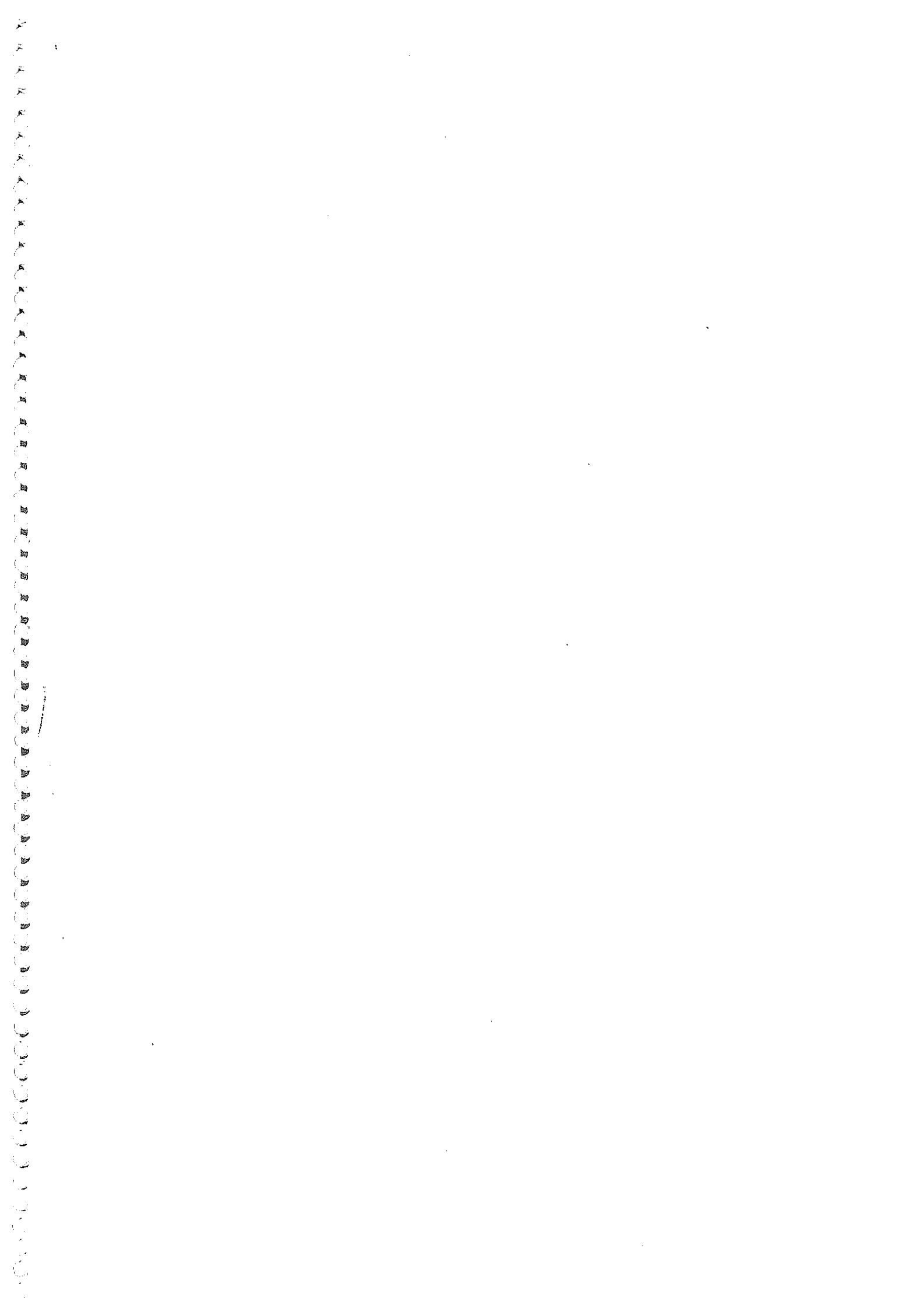
(*****)
(*is_knight_tour : int -> int -> (int * int) list -> bool *)
(*****)

let is_in_board m n (x,y)
  x>=1 && x <= m && y>=1 && y<=n;;

let is_legal_move (x1,y1) (x2,y2) =
  ((abs (x1,x2)=2 && abs(y1,y2) =1)|| (abs (x1,x2)=1 && abs(y1,y2) =2));;

let rec are_chained_moves = function
  h::[]-> true

```



```
- : float -> int = <fun>*)

ceil;;

(*Tipo de la funcion
- : float -> float = <fun>*)

floor;;

(*Tipo de la funcion
- : float -> float = <fun>*)

Char.code;;

(*Tipo de la funcion
- : char -> int = <fun>*)

String.length;;

(*Tipo de la funcion
- : string -> int = <fun>*)

fst;;

(*Tipo de la funcion
- : 'a * 'b -> 'a = <fun>*)

snd;;

(*Tipo de la funcion
- : 'a * 'b -> 'b = <fun>*)

function x->2*x;;
(*Tipo de la funcion
- : int -> int = <fun>*)

(function x->2*x) (2+1);;

(*Valor de la funcion
- : int = 6*)

function (x,y) ->x;;
(*Tipo de la funcion
- : 'a * 'b -> 'a = <fun>*)

let f=function x->2*x;;
(*Tipo de la funcion
val f : int -> int = <fun>*)

f (2+1);;

(*Valor de la funcion
- : int = 6*)

f 2+1;;
```

```
(*Valor de la funcion
- : int = 5*)
```

```

(*nuevo valor para x
val x : int = 5*)

z;;
(*z aun conserva su valor
- : int = 3*)

let y=5 in x+y;;
(*Operacion con variables locales, sus valores vuelven al estado
original al final
- : int = 10*)

x+y;;
(*Operacion con sus valores correctos
- : int = 7*)

let p=2,5;;
(*Declara un par con un nombre
val p : int * int = (2, 5)*)

snd p, fst p;;
(*Devuelve valores segundo y primero
- : int * int = (5, 2)*)

p;;
(*p sigue conservando su valor
- : int * int = (2, 5)*)

let p=0,1 in snd p, fst p;;
(*Operacion en el par localmente, los valores se restauraran
- : int * int = (1, 0)*)

p;;
(*p sigue conservando su valor
- : int * int = (2, 5)*)

let x,y=p;;
(*asignando el par p,el primero a x y el segundo a y
val x : int = 2
val y : int = 5*)

let z=x+y;;
(*Valor para z, sumando x e y
val z : int = 7*)

let x,y=p,x;;
(*Asignacion,x ahora sera un par e y tomara el valor de x antes de su
nueva asignacion
val x : int * int = (2, 5)

```

```
val y : int = 2*)

let x = let x,y=2,3 in x*x +y;;

(*Asignacion de x con otra asignacion que luego se restaurara
val x : int = 7*)

x+y;;

(*x tiene su nuevo valor, e y conserva el suyo
- : int = 9*)

z;;

(*z sigue conservando su valor
- : int = 7*)

let x =x+y in let y=x*y in x+y+z;;

(*Asignacion de x con otra asignacion que luego se restaurara
- : int = 34*)

x+y+z;;

(*Operacion que da un resultado
- : int = 16*)

int_of_float;;
```

```
(*Tipo de la funcion
- : float -> int = <fun>*)

float_of_int;;
```

```
(*Tipo de la funcion
- : int -> float = <fun>*)

int_of_char;;
```

```
(*Tipo de la funcion
- : char -> int = <fun>*)

char_of_int;;
```

```
(*Tipo de la funcion
- : int -> char = <fun>*)

abs;;
```

```
(*Tipo de la funcion
- : int -> int = <fun>*)

sqrt;;
```

```
(*Tipo de la funcion
- : float -> float = <fun>*)

truncate;;
```

```
(*Tipo de la funcion
```

```
(*Error, los resultados han de ser del mismo tipo
Error: This expression has type string but an expression was expected
of type
    int*)

(if 3<5 then 8 else 10)+4;;

(*Se evalua la expresion y luego se le suma 4
- : int = 12*)

let pi=3.14;;

(*La variable de nombre pi recibira ese valor
val pi : float = 3.14*)

sin (pi/. 2.);;

(*Realiza la funcion seno sobre la variable pi que definimos antes
- : float = 0.99999968293183461*)

let x =1;;
(*La variable x llevara el valor 1
val x : int = 1*)

let y =2;;
(*La variable y llevara el valor 2
val y : int = 2*)

x-y;;
(*Operacion con las variables de mismo tipo
- : int = -1*)

let x=y in x-y;;
(*Primero declara las variables localmente, despues vuelven a tomar su
valor
- : int = 0*)

x-y;;
(*
- : int = -1*)

z;;
(*Error, z no se ha definido previamente
Error: Unbound value z*)

let z=x+y;;
(*Nuevo valor para z
val z : int = 3*)

z;;
(*z ya tiene valor
- : int = 3*)

let x=5;;
```

```

(*Comparacion entre cadenas
- : bool = false*)

2,5;;

(*Declara un tipo par entre dos ints
- : int * int = (2, 5)*)

"holo", "adios";;

(*Declara un tipo para entre dos strings
- : string * string = ("holo", "adios")*)

0, 0.0;;

(*Declara un tipo par entre un int y un float
- : int * float = (0, 0.)*)

fst ('a',0);;

(*Devolvera el primer elemento del par
- : char = 'a'*)

snd (false, true);;

(*Devolvera el segundo elemento del par
- : bool = true*)

(1,2,3);
(*
- : int * int * int = (1, 2, 3)*)

(1,2),3;; 

(*Declara un tipo par con su primer elemento otro par
- : (int * int) * int = ((1, 2), 3)*)

fst ((1,2),3);;

(*Devuelve el primer elemento del par, que es otro par
- : int * int = (1, 2)*)

(),abs;; 

(*Declara un par en donde el primer elemento es unit, y el segundo una
funcion que va de int en int
- : unit * (int -> int) = ((), <fun>)*)

if 3=4 then 0 else 4;; 

(*Posibles resultados al evaluar una expresion logica
- : int = 4*)

if 3=4 then "0" else "4";;

(*Posibles resultados al evaluar una expresion logica
- : string = "4"*) 

if 3=4 then 0 else "4";;

```

```
int_of_string "1999" +1;;
(*Devolvera el int correspondiente a la operacion
- : int = 2000*)

"\064\065";;

(*Declaracion de string con codigos de caracteres
- : string = "@A"*)

string_of_int 010;;
(*Devuelve el int transformado en un string
- : string = "10"*)

not true;;
(*Negacion del valor logico true
- : bool = false*)

true && false;;
(*Operacion and logica entre true y false
- : bool = false*)

true || false;;
(*Operacion or logica entre true y false
- : bool = true*)

true or false;;
(*Operacion or logica entre true y false
- : bool = true*)

true and false;;
(*Operacion and logica entre true y false
Error: Syntax error no existe el valor and*)

(1<2) =false;;
(*Definicion logica que es false
- : bool = false*)

"1" <"2";;

(*Compara sus valores en ascii
- : bool = true*)

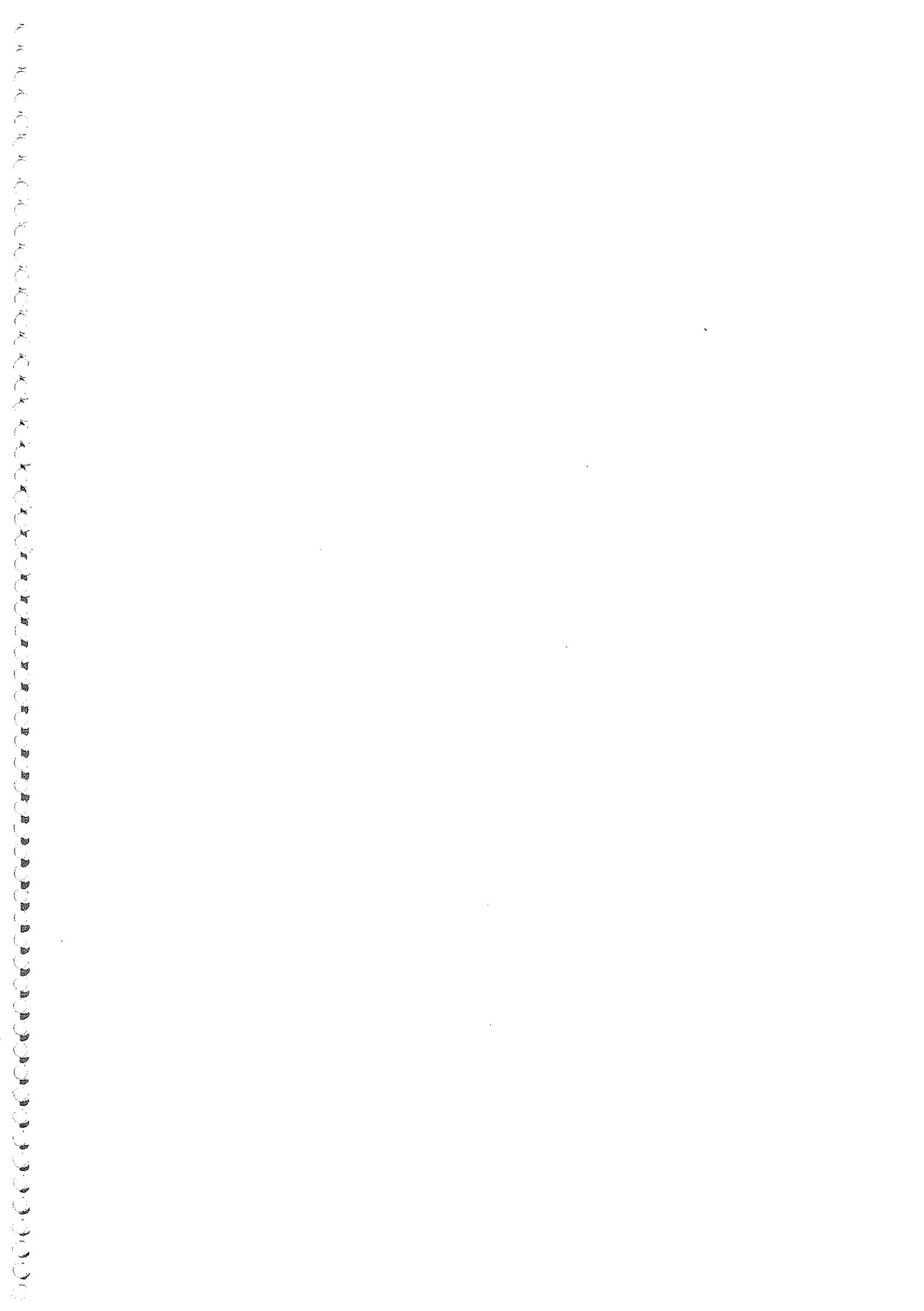
2<12;;
(*Comparacion logica entre dos ints
- : bool = true*)

"2" <"12";;

(*Comparacion entre cadenas
- : bool = false*)

"uno" < "dos";;
```

```
();;  
(*Deberia devolver el mÃ³dulo Unit  
- : unit = ()*)  
  
2+5*3;;  
(*DevolverÃ¡; el resultado de evaluar la expresiÃ³n  
- : int = 17*)  
  
1.0;;  
(*Devuelve el tipo de evaluar la expresiÃ³n, sera un float  
- : float = 1.*)  
  
1.0*2;;  
(*Devolvera error porque el operador '*' opera solo con int, no con un  
float  
La operaciÃ³n correcta serÃ¡: 1.0*.2.0;;  
  
Error: This expression has type float but an expression was expected  
of type  
    int*)  
  
2-2.0;;  
(*Devolvera error porque el operando '-' no puede operar un int y un  
float  
La expresiÃ³n correcta podrÃ¡a ser: 2-2;;  
  
Error: This expression has type float but an expression was expected  
of type  
    int*)  
  
3.0+2.0;;  
(*Devolvera error porque el operando '+' no puede operar sobre dos  
floats  
se utilizarÃ¡a en su lugar el operador '+.'  
  
Error: This expression has type float but an expression was expected  
of type  
    int*)  
  
3.0+.2.0;;  
(*Ahora el resultado serÃ¡; correcto porque el operador '+.' opera  
sobre floats  
- : float = 5.*)  
  
5/3;;  
(*Devuelve el resultado de evaluar la expresiÃ³n  
- : int = 1*)  
  
5 mod 3;;  
(*Devuelve el el resto de dividir 5 entre 3  
- : int = 2*)
```



```

let rec remove n l =match l with
  []->[]
  |h::t->if (n=h) then t
    else
      h::(remove n t);;

(*-----
-----*)
FUNCTION remove_all
-----*)

let rec remove_all n l = match l with
[]->[]
| h::t->if (n=h)then
  remove_all n t
else
  h::(remove_all n t);;

(*-----
-----*)
FUNCTION ldif
-----*)

let rec ldif l1 l2 = match l1,l2 with
  ([],_)->raise(Failure"ldif")
  |(l1,[])->l1
  |(h1::t1,h2::t2)->ldif(remove_all h2 l1) t2;; 

(*-----
-----*)
FUNCTION lprod
-----*)

let rec lprod l1 l2 = match l1 with
  []->[]
  |h::t->(map(function x->(h,x))l2)@(lprod t l2);;

(*-----
-----*)
FUNCTION mem
-----*)

let rec mem e l = match l with
  [] -> false
  | h::t -> if (e = h)
    then true
    else mem e t;;

```

```

(*-----
-----)
FUNCTION filter
-----*)

let rec filter p =function
[]->[]
|(h::t)->if p h then [h]
else
filter p t;;
```



```

(*-----
-----)
FUNCTION find_all
-----*)

let find_all=filter;;
```



```

(*-----
-----)
FUNCTION partition
-----*)

let rec partition p=function
[]->([],[])
|(h::t)->if p h then ([h],[])
else
([],filter p t);;
```



```

(*-----
-----)
FUNCTION split
-----*)

let rec split = function
[]->([],[])
|(x,y)::t->let(lx,ly)= split t in (x::lx,y::ly);;
```



```

(*-----
-----)
FUNCTION combine
-----*)

let rec combine l1 l2 = match(l1,l2) with
([], [])->[]
|(a1:::t1,a2:::t2)->(a1,a2):::combine t1 t2
|(_, _)->raise(Invalid_argument "combine");;
```



```

(*-----
-----)
FUNCTION remove
-----*)

```

```
(op (hd l1) (hd l2)) :: map2 op (tl l1) (tl l2);;
```

```
(*-----  
-----  
FUNCION fold_left  
-----  
-----*)
```

```
let rec fold_left f accum l = match l with  
[] -> accum  
| h :: t -> fold_left f (f accum h) t;;
```

```
(*-----  
-----  
FUNCION fold_right  
-----  
-----*)
```

```
let rec fold_right f l accum = match l with  
[] -> accum  
| h :: t -> f h (fold_right f t accum);;
```

```
(*-----  
-----  
FUNCION find  
-----  
-----*)
```

```
let rec find e l = match l with  
[] -> raise(Not_found)  
| (h :: t) -> if (e h) then  
    h  
else  
    find e t;;
```

```
(*-----  
-----  
FUNCION for_all  
-----  
-----*)
```

```
let rec for_all p = function  
[] -> true  
| h :: t -> (p h) && (for_all p t);;
```

```
(*-----  
-----  
FUNCION exists  
-----  
-----*)
```

```
let rec exists p = function  
[] -> false  
| h :: t -> if p h then true  
else  
exists p t;;
```

PROGRAMACIÓN DECLARATIVA
11 de febrero de 2003

NOMBRE: _____ DNI: _____

I.I. I.T.I.G.

1. (4 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el "toplevel" de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

let x,y = -2.5, 2.5;;

val x : float = -2.5
val y : float = 2.5

ELENA
DELAMANDE
FREIJE

let dup f x = f (fst x), f (snd x);;

val dup : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>.

dup (+);;

- : int * int -> (int -> int) * (int -> int) = <fun>

let p = dup floor (y,x);;

val p : float * float = 2, -3

let p = let x,y = p in y,x;;

val p : float * float = -3, 2

let x = x > y and y = x;;

val x : bool = false
val y : float = -2.5

let rec map2 f1 f2 = function
 [] -> []
 | h::t -> f1 h :: map2 f2 f1 t;;

val map2 : ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list = <fun>

let rec f = function x -> x * x and g x = f (x - 1) + x
in map2 f g [1;2;3;4;5];;

- : int list = [1; 3; 9; 13; 25]

1. (1 pto.) Defina una función "imprime_inversa: int list -> unit" que "visualice" por la salida estándar los elementos de una lista de enteros en orden inverso y uno por línea. Así, por ejemplo, al evaluar la expresión "imprime_inversa [1; 2; 3]" debería aparecer por la salida estándar:

3
2
1

```
let rec imprime_inversa = function
  [] -> ()
  | h::t -> imprime_inversa t;
              print_endline(string_of_int h);;
```

3. (2 ptos.) Observe la siguiente definición de la función fold_right del módulo List de caml y realice una nueva definición que sea recursiva terminal.

```
let rec fold_right f l e = match l with
  [] -> e
  | h::t -> f h (fold_right f t e);;
```

```
let fold_right f l e =
  let rec aux = function
    ([] , r) -> r
    | (h::t, r) -> aux (t, f h r)
  in aux (List.rev l, e);;
```

4. (3 ptos.) Considere la siguiente definición del tipo de dato 'a tree que podría servir para representar en ocaml cierto tipo de árboles binarios:

```
type 'a tree = Leaf of 'a | Node of ('a tree * 'a * 'a tree);;
```

Llamaremos "caminos de un árbol" a cada uno de los recorridos descendentes desde la raíz a cada una de las hojas.

Si tenemos un árbol con valores numéricos asociados a los nodos, diremos que el "peso" de un camino es la suma de los valores de todos los nodos que lo componen.

Así, por ejemplo, en el siguiente árbol el peso máximo de todos sus caminos es 15.

a) Defina una función "peso_maximo: float tree -> float" que devuelva el valor del camino (o los caminos) de peso máximo de un árbol.

```
let rec peso_maximo = function
  Leaf p -> p
  | Node (i, c, d) -> c +. max (peso_maximo i) (peso_maximo
d);;
```

b) Defina una función "caminos: 'a tree -> 'a list list" que devuelva todos los caminos de un árbol de izquierda a derecha, de forma que, por ejemplo, para el árbol del dibujo dé la lista [[3;11]; [3;4;-2;10]; [3;4;-2;9]; [3;4;8]].

```
let rec caminos = function
  | Leaf p -> [[p]]
  | Node (i, c, d) ->
    let f l = c :: l in
    List.map f (caminos i) @ List.map f (caminos d);;
```

c) Un camino dentro de un árbol, puede indicarse enumerando las ramas que hay que escoger en cada nodo para seguirlo desde la raíz hasta la hoja. Si elegimos la letra 'I' para referirnos a las ramas izquierdas y 'D' para las derechas, las listas ['D'; 'I'; 'I'] y ['D'; 'D'] describen ambas caminos de peso máximo en el árbol del dibujo.

Defina una función "camino_maximo: float tree -> char list" que describa un camino de peso máximo en cada árbol dado.

```
let camino_maximo a =
  let rec aux = function
    | Leaf a -> a, []
    | Node (i, n, d) -> let (p1, c1) = aux i
                           and (p2, c2) = aux d
                           in if p1 > p2 then n +. p1, 'I'::c1
                               else n +. p2, 'D'::c2
  in snd (aux a);;
```

NOTA: Las definiciones de los ejercicios 2, 3 y 4 deben realizarse de la forma más sencilla posible.

PROGRAMACIÓN DECLARATIVA
11 de febrero de 2003

NOMBRE: _____

DNI: _____

I.I. I.T.I.G.

1. (4 ptos.) Las siguientes frases corresponden al código introducido en una sesión de trabajo con el "toplevel" de ocaml. Indique, después de cada una de ellas, cuál sería la respuesta que daría el compilador.

```
# let x,y = -2.5, 2.5;;
```

```
val x : float = -2.5
val y : float = 2.5
```

```
# let dup f x = f (fst x), f (snd x);;
```

```
val dup : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>.
```

```
# dup (+);;
```

```
- : int * int -> (int -> int) * (int -> int) = <fun>
```

let p = dup floor (y,x);; (floor devuelve el menor float mas cercano al que le pasamos)

```
val p : float * float = 2, -3
```

```
# let p = let x,y = p in y,x;;
```

```
val p : float * float = -3, 2
```

ELENA
DELAMANO
FREIJE

```
# let x = x > y and y = x;;
```

```
val x : bool = false
val y : float = -2.5
```

```
# let rec map2 f1 f2 = function
  [] -> []
  | h::t -> f1 h :: map2 f2 f1 t;;
```

```
val map2 : ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# let rec f = function x -> x * x and g x = f (x - 1) + x
in map2 f g [1;2;3;4;5];;
```

```
- : int list = [1; 3; 9; 13; 25]
```

2. (1 pto.) Defina una función “`imprime_inversa: int list -> unit`” que “visualice” por la salida estándar los elementos de una lista de enteros en orden inverso y uno por línea.
Así, por ejemplo, al evaluar la expresión “`imprime_inversa [1;2;3]`” debería aparecer por la salida estándar:

3
2
1

```
let rec imprime_inversa = function
  [] -> ()
  | h::t -> imprime_inversa t;
              print_endline(string_of_int h);;
```

3. (2 ptos.) Observe la siguiente definición de la función `fold_right` del módulo `List` de caml y realice una nueva definición que sea recursiva terminal.

```
let rec fold_right f l e = match l with
  [] -> e
  | h::t -> f h (fold_right f t e);;
```

```
let fold_right f l e =
  let rec aux = function
    ([] , r) -> r
    | (h::t, r) -> aux (t, f h r)
  in aux (List.rev l, e);;
```

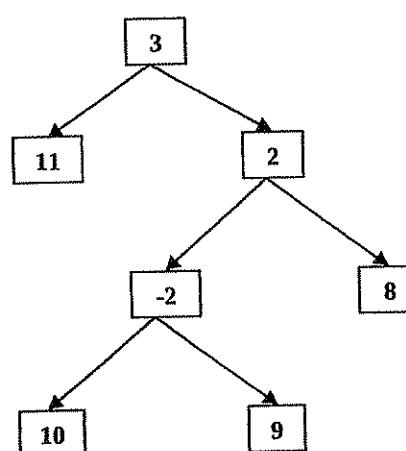
4. (3 ptos.) Considere la siguiente definición del tipo de dato '`a tree`' que podría servir para representar en ocamí cierto tipo de árboles binarios:

```
type 'a tree = Leaf of 'a | Node of ('a tree * 'a * 'a tree);;
```

Llamaremos "caminos de un árbol" a cada uno de los recorridos descendentes desde la raíz a cada una de las hojas.

Si tenemos un árbol con valores numéricos asociados a los nodos, diremos que el "peso" de un camino es la suma de los valores de todos los nodos que lo componen.

Así, por ejemplo, en el siguiente árbol el **peso máximo** de todos sus caminos es 15.



- a) Defina una función "`peso_maximo: float tree -> float`" que devuelva el valor del camino (o los caminos) de peso máximo de un árbol.

```
let rec peso_maximo = function
  Leaf p -> p
  | Node (i, c, d) -> c +. max (peso_maximo i) (peso_maximo d);;
```

- b) Defina una función “caminos: ‘a tree -> ‘a list list” que devuelva todos los caminos de un árbol de izquierda a derecha, de forma que, por ejemplo, para el árbol del dibujo dé la lista `[[3;11];[3;4;-2;10];[3;4;-2;9];[3;4;8]]`.

```
let rec caminos = function
  Leaf p -> [[p]]
  | Node (i, c, d) ->
    let f l = c :: l in
    List.map f (caminos i) @ List.map f (caminos d);;
```

ELENA
DELAMANO
FREIIE

- c) Un camino dentro de un árbol, puede indicarse enumerando las ramas que hay que escoger en cada nodo para seguirlo desde la raíz hasta la hoja. Si elegimos la letra ‘I’ para referirnos a las ramas izquierdas y ‘D’ para las derechas, las listas `['D'; 'I'; 'I']` y `['D'; 'D']` describen ambas caminos de peso máximo en el árbol del dibujo.

Defina una función “camino_maximo: float tree -> char list” que describa un camino de peso máximo en cada árbol dado.

```
let camino_maximo a =
  let rec aux = function
    Leaf a -> a, []
    | Node (i, n, d) -> let (p1, c1) = aux i
                           and (p2, c2) = aux d
                           in if p1 > p2 then n +. p1, 'I'::c1
                             else n +. p2, 'D'::c2
  in snd (aux a);;
```

NOTA: Las definiciones de los ejercicios 2, 3 y 4 deben realizarse de la forma más sencilla posible.

PROGRAMACIÓN DECLARATIVA

6 DE SEPTIEMBRE DE 2003

1. (5 ptos.) Escriba el resultado de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

```
let x, y = 1,5;;
val x : int = 1
val y : int = 5
```

```
let x = let y = x < y in y;;
val x : bool = true
```

```
let z = if x then y + 1 else y - 1;;
val z : int = 6
```

```
let x y = y + 1 in x;;
- : int -> int = <fun>
```

```
x;;
- : bool = true
```

```
let x y = y + 1;;
val x : int -> int = <fun>
```

```
x y,x z;;
- : int * int = 6, 7
```

```
let f = function f -> snd f, fst f;;
val f : 'a * 'b -> 'b * 'a = <fun>
```

```
let x = f (y,z);;
val x : int * int = 6, 5
```

```
f x, x;;
- : (int * int) * (int * int) = (5, 6), (6, 5)
```

```
let f p = f (f p) in f (1,2);;
- : int * int = 1, 2
```

```
f (1,2);;
- : int * int = 2, 1
```

```
let dos x y = x (x y y) y;;
val dos : ('a -> 'a -> 'a) -> 'a -> 'a = <fun>
```

```
dos (+) 2, dos (*) 2;;
- : int * int = 6, 8
```

```
function x -> function y -> function z -> z y x;;
```

```
- : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c = <fun>
```

2. (2'5 ptos.) Considere las siguientes definiciones en ocaml:

```
let mappar f (x,y) = f x, f y;;
let rec split f = function
  [] -> [], []
  | h::t -> let t1, t2 = split f t
    in if f h then h::t1, t2
      else t1, h::t2;;
let split f l = mappar List.rev (split f l);;
```

a. Indique el tipo de cada una de las funciones definidas con ese código.

```
val mappar : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>
val split : ('a -> bool) -> 'a list -> 'a list * 'a list =
<fun>
val split : ('a -> bool) -> 'a list -> 'a list * 'a list =
<fun>
```

b. Escriba una definición terminal para la última definición de "split" (sin usar el código anterior).

```
let split f l =
  let rec aux = function
    ([] , ll, l2) -> ll, l2
    | (h::t, ll, l2) -> if f h then aux (t, h::ll, l2)
      else aux (t, ll, h::l2)
  in aux (l, [], []);;
```

3. (2'5 ptos.) Indique el tipo de cada una de las funciones definidas en el siguiente fragmento de código ocaml y, luego, simplifíquelo. Es decir, reescribalo de la forma más breve posible, de modo que todas las definiciones resulten totalmente equivalentes a las dadas. (Puede utilizar cualquier valor predefinido por el compilador de ocaml).

```
let rec mx x y = if x > y = true then x else y
and my x y = if x > y then true else false;;

let rec rollo f n l =
  if l = [] then n
  else let nn = f n (List.hd l) and nl = List.tl l
    in rollo f nn nl;;

val mx : 'a -> 'a -> 'a = <fun>
val my : 'a -> 'a -> bool = <fun>
val rollo : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

let mx = max and my = (>);;
```

```
let rollo = List.fold_left;;
```

PROGRAMACIÓN DECLARATIVA

30 DE ENERO DE 2004
PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

I.I.

I.T.I.G.

1. (2,5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

let no f x = not (f x);;

val no : ('a → bool) → 'a → bool = <fun>

let par x = x mod 2 = 0 in no par;;

- : int → bool = <fun>

let rec rep n f x = if n > 0 then rep (n-1) f (f x) else x;;

val rep : int → ('a → 'a) → 'a → 'a = <fun>

rep 3 (function x -> x * x) 2, rep 4 (function x -> 2 * x) 1;;

- : int * int = (256, 16)

(let par x y = function z -> x z, y z in par ((+) 2) ((/) 2)) 3;;

- : int * int = (5, 0)

2. (2 puntos) Dada la siguiente definición del tipo de dato '*a arbol*', que sirve para representar cierto tipo de árboles binarios

type 'a arbol = Vacio | Nodo of ('a * 'a arbol * 'a arbol);;

- a. defina una función *cont*: '*a* -> '*a arbol* -> int, que devuelva el número de nodos de un árbol que están etiquetados con un valor determinado.

```
let rec cont x = function
  Vacio → 0
  | Nodo (r, Vacio, Vacio) → if r = x then 1 else 0
  | Nodo (r, i, Vacio) → if (r = x) then 1 + cont x i
    else cont x i
  | Nodo (r, Vacio, d) → if (r = x) then 1 + cont x d
    else cont x d
  | Nodo (r, i, d) → if r = x then i + cont x i +
    cont x d else cont x i + cont x d;;
```

b. defina una función `subst`: '`a -> 'a -> 'a arbol -> 'a arbol`', de forma que `subst x y` sea una función que, al aplicarla a un árbol, devuelva un árbol igual al original salvo los nodos que tuviesen valor `x`, que tendrán valor `y`.

```

let rec subst x y = function
  | Vacio =>
    | Nodo (r, Vacio, Vacio) => if r=x then Nodo (y, Vacio, Vacio)
    | Nodo (r, Vacio, Vacio) else Nodo (r, Vacio, Vacio)
    | Nodo (r, i, Vacio) => if r=x then Nodo (y, subst x y i, Vacio)
    | Nodo (r, i, Vacio) else Nodo (r, subst x y i, Vacio)
    | Nodo (r, Vacio, a) => if r=x then Nodo (y, Vacio, subst x y a)
    | Nodo (r, Vacio, a) else Nodo (r, Vacio, subst x y a)
    | Nodo (r, i, a) => if r=x then Nodo (y, subst x y i, subst x y a)
    | Nodo (r, i, a) else Nodo (r, subst x y i, subst x y a);
  
```

3. (1 punto) Defina una función `L_ordenada`: '`'a -> 'a -> bool` -> '`'a list -> bool`', de forma que si `f` es una relación de orden en el tipo `'a` (esto es, una función que dice si dos elementos de este tipo están ordenados), `L_ordenada f` sea la función que dice si una lista está ordenada según el orden `f`.

Práctica 3 → sorted-by.

ELENA
DELAMAND
FREIJJE

4. (1 punto) Defina utilizando exclusivamente recursividad terminal una función `L_max`: '`'a list -> 'a` que devuelva de cada lista el mayor de sus elementos.

```

let L_max = function
  [] => raise (Failure "L_max")
  [h;t] => let rec aux x = function
    [] => x
    [h;t] => if x > h then aux x t
    else aux h t
  in aux h t;;
  
```

PROGRAMACIÓN DECLARATIVA

30 DE ENERO DE 2004
PROGRAMACIÓN FUNCIONAL

NOMBRE: Idem que anterior.

I.I.

I.T.I.G.

1. (2,5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

let no f x = not (f x);;

val no: ($\alpha \rightarrow \text{bool}$) $\rightarrow \alpha \rightarrow \text{bool} = <\text{fun}>$

let par x = x mod 2 = 0 in no par;;

val no: $\neg : \text{int} \rightarrow \text{bool} = <\text{fun}>$

let rec rep n f x = if n > 0 then rep (n-1) f (f x) else x;;

val rep: int \rightarrow ($\alpha \rightarrow \alpha$) \rightarrow 'a \rightarrow 'a = < fun >

rep 3 (function x -> x * x) 2, rep 4 (function x -> 2 * x) 1;;

$\neg : \text{int}^* \rightarrow (\text{int} \rightarrow \text{int}) = (256, <\text{fun}>)$

(let par x y = function z -> x z, y z in par ((+) 2) ((//) 2)) 3;;

$\neg : \text{int}^* \times \text{int}^* = (5, 0)$

2. (2 puntos) Dada la siguiente definición del tipo de dato ' α arbol', que sirve para representar cierto tipo de árboles binarios

type ' α arbol = Vacio | Nodo of (' α * ' α arbol * ' α arbol);;

a. defina una función $cont: \alpha \rightarrow \alpha \text{ arbol} \rightarrow \text{int}$, que devuelva el número de nodos de un árbol que están etiquetados con un valor determinado.

Página anterior



b. defina una función $\text{subst}: 'a \rightarrow 'a \rightarrow 'a \text{ arbol} \rightarrow 'a \text{ arbol}$, de forma que $\text{subst } x \ y$ sea una función que, al aplicarla a un árbol, devuelve un árbol igual al original salvo los nodos que tuviesen valor x , que tendrán valor y .

3. (1 punto) Defina una función $\text{l_ordenada}: ('a \rightarrow 'a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow \text{bool}$, de forma que si f es una relación de orden en el tipo $'a$ (esto es, una función que dice si dos elementos de este tipo están ordenados), $\text{l_ordenada } f$ sea la función que dice si una lista está ordenada según el orden f .

4. (1 punto) Defina utilizando exclusivamente recursividad terminal una función $\text{l_max}: 'a \text{ list} \rightarrow 'a$ que devuelva de cada lista el mayor de sus elementos.

```
let l-max l =
  let rec aux acum lista =
    [ ]
    | h :: t => if h >= acum then aux(h) t
      else aux(acum) t
    in o. e;
```

PROGRAMACIÓN DECLARATIVA

8 DE SEPTIEMBRE DE 2004

PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

I.I.

I.T.I.G

1. (3,5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

```
let apa x f = f x;;
```

```
val apa : 'a -> ('a -> 'b) -> 'b = <fun>
```

```
List.map (apa 2) [(function x -> x * x); succ; (+)1; (-) 1];;
```

```
- : int list = [4; 3; 3; -1] (hace 1-2 =-1) // Aplica apa 2 a todos los elementos
```

```
let apa_rep n x f =
  let rec aux x = function 0 -> x
    | n -> aux (apa x f) (n-1)
  in aux x (abs n);;
```

ELENA
DELAMANO
FREIJE

```
val apa_rep : int -> 'a -> ('a -> 'a) -> 'a = <fun>
```

```
apa_rep (-2) "x" (function x -> x ^ x);;
```

```
- : string = "xxxx"
```

```
let fop op f g = function y -> op (f y) (g y);;
```

```
val fop : ('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> 'd -> 'c = <fun>
```

```
let suma = fop (+);;
```

```
val suma : ('a -> int) -> ('a -> int) -> 'a -> int = <fun>
```

```
let f = let f1 x = x * x in
      let f2 x = f1 x * x in
      suma f1 f2
in f 2;;
```

```
- : int = 12
```

2. (1 punto) Redefina la función `f` de modo que sólo se utilice recursividad terminal (pero debe dar siempre el mismo resultado que la original).

```
let rec f orden =
  function [] -> raise (Failure "f") | [x] -> x
    | h::t -> let m = f orden t in
      if orden h m then h else m;;
```

```
let f orden =
  function [] -> raise (Failure "f")
  | h::t -> let rec aux x = function [] -> x
    | h::t -> if orden x h then aux x t
      else aux h t
  in aux h t;;
```

3. (2 puntos) Una relación (de equivalencia, de orden, etc...) en un conjunto A puede representarse como una función de $(A \times A) \rightarrow \text{bool}$, que indica para cada pareja de elementos de A si están o no relacionados. Dada una función cualquiera $f : A \rightarrow B$, puede hablarse de la relación de equivalencia que induce sobre el conjunto A como aquella en la que son equivalentes los elementos que tienen la misma imagen.

a. Defina en ocaml una función `rel_eq : ('a -> 'b) -> 'a * 'a -> bool`, que para cualquier función devuelva la relación de equivalencia inducida por ella en el sentido señalado.

```
let rel_eq f (x,y) = f x = f y;;
```

b. Defina en ocaml una función
`clases_eq : ('a * 'a -> bool) -> 'a list -> 'a list list`,
de modo que, dada una relación de equivalencia r sobre un conjunto (tipo de dato) A y dada una lista de elementos de A , "divida" los elementos de la lista en clases de equivalencia inducidas por la relación r .
(El orden no sería relevante, aunque sí el número de apariciones de cada elemento; así, por ejemplo,

```
clases_eq (function (x,y) -> x mod 2 = y mod 2)
[5;7;9;10;30;0;4;1;5;10]
```

podría ser la lista

```
[[10; 30; 0; 4; 10]; [5; 7; 9; 1; 5]]).
```

```
let rec clases_eq r =
  let rec anadir x = function
    [] -> [[x]]
    | (h::t)::clases -> if r (x, h) then (x::h::t)::clases
                           else (h::t):: anadir x clases
  in function [] -> []
        | h::t -> anadir h (clases_eq r t);;
```

ELENA
DELAMANO
FREIJE

PROGRAMACIÓN DECLARATIVA

8 DE SEPTIEMBRE DE 2004
PROGRAMACIÓN FUNCIONAL

NOMBRE: _____

I.I.

I.T.I.G

1. (3,5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el "toplevel" de ocaml:

let apa x f = f x;;

val apa : 'a -> ('a -> 'b) -> 'b = <fun>

✓

List.map (apa 2) [(function x -> x * x); succ; (+)1; (-) 1];;

- : int list = [4; 3; 3; -1]

✓

let apa_rep n x f =
 let rec aux x = function 0 -> x
 | n -> aux (apa x f) (n-1)
 in aux x (abs n);;

val apa_rep : int -> 'a -> ('a -> 'a) -> 'a = <fun>

✓

apa_rep (-2) "x" (function x -> x ^ x);;

- : string = "xxxx"

✓

let fop op f g = function y -> op (f y) (g y);;

val fop : ('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> 'd -> 'c = <fun> ✗

let suma = fop (+);;

val suma : ('a -> int) -> ('a -> int) -> 'a -> int = <fun> ✗

let f = let f1 x = x * x in
 let f2 x = f1 x * x in
 suma f1 f2
in f 2;;

- : int = 12

2. (1 punto) Redefina la función f de modo que sólo se utilice recursividad terminal (pero debe dar siempre el mismo resultado que la original).

```
let rec f orden =
    function [] -> raise (Failure "f") | [x] -> x
    | h::t -> let m = f orden t in
        if orden h m then h else m;;
```

```
let f orden =
    function [] -> raise (Failure "f")
    | h::t -> let rec aux x = function [] -> x
        | h::t -> if orden x h then aux x t
        else aux h t
    in aux h t;;
```

3. (2 puntos) Una relación (de equivalencia, de orden, etc...) en un conjunto A puede representarse como una función de $(A \times A) \rightarrow \text{bool}$, que indica para cada pareja de elementos de A si están o no relacionados. Dada una función cualquiera $f : A \rightarrow B$, puede hablarse de la relación de equivalencia que induce sobre el conjunto A como aquella en la que son equivalentes los elementos que tienen la misma imagen.

a. Definá en ocaml una función `rel_eq : ('a -> 'b) -> 'a * 'a -> bool`, que para cualquier función devuelva la relación de equivalencia inducida por ella en el sentido señalado.

```
let rel_eq f (x,y) = f x = f y;;
```

ELENA
DELAMANO
FREIJES

b. Defina en ocaml una función

`clases_eq : ('a * 'a -> bool) -> 'a list -> 'a list list,`
de modo que, dada una relación de equivalencia r sobre un conjunto (tipo de dato) A y dada una lista de elementos de A , "divida" los elementos de la lista en clases de equivalencia inducidas por la relación r .

(El orden no sería relevante, aunque sí el número de apariciones de cada elemento; así, por ejemplo,

```
clases_eq (function (x,y) -> x mod 2 = y mod 2)
[5;7;9;10;30;0;4;1;5;10]
```

podría ser la lista

```
[[10; 30; 0; 4; 10]; [5; 7; 9; 1; 5]]).
```

```
let rec clases_eq r =
  let rec anadir x = function
    [] -> [[x]]
    | (h::t)::clases -> if r (x, h) then (x::h::t)::clases
                           else (h::t):: anadir x clases
  in function [] -> []
        | h::t -> anadir h (clases_eq r t)::
```

PROGRAMACIÓN DECLARATIVA
3 DE FEBRERO DE 2005

NOMBRE: _____

I.I.

I.T.I.G.

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el toplevel de ocaml:

ELENA
DELAMANO
FREIJAS

```
let x = let x = 0 in x, x+1, x+2;;
val x: int * int * int = (0,1,2)
```

```
let par = let f x = 2 * x in let g x = f x * x in (f,g);;
-: val par: (int → int) * (int → int) = (<fun>, <fun>).
```

```
let doble p x = fst p x, snd p x;;
val doble: ('a → 'b) * ('a → 'c) → 'a → 'b * 'c = <fun>
```

```
let rap = doble par in rap 2;;
-: unit → int = (4,8)
```

```
let rec g l x = match l with [] -> x | h::t -> g t (h x);;
val g: ('a → 'a) list → 'a → 'a = <fun>
```

```
let rec suces gen x0=
  function 0 -> [] | n -> x0 :: suces gen (gen x0) (n-1);;
val suces: ('a → 'a) → 'a → int → 'a list = <fun>
```

```
let lista = suces ((+) 1) 0 5;;
val lista : int list = [0;1;2;3;4]
```

```
g (List.map (+) lista) 10;;
-: int = 20
```

2. (2 puntos) Realice una nueva definición para la función *sumpro* definida a continuación, de forma que sólo se utilice recursividad terminal.

```
let rec sumpro n =
  if n < 1 then (0,1)
  else let (s,p) = sumpro (n-1) in
    (s+n, p*n);;
```

```
let sumpro n =
  let rec aux (x,i,j) =
    if (x < 1) then
      (i,j)
    else aux (x-1, i+x, j*x)
  in aux (n,0,1);}
```

3. (2 puntos) Defina una función *aBase*: *int -> int -> int list* tal que *aBase b n* devuelva la lista de enteros correspondiente a los dígitos de la representación en base *b* del número *n* (de forma que la cabeza de la lista corresponda al dígito menos significativo, y una función *deBase*: *int -> int list -> int* tal que *deBase b l* devuelva el número entero cuya representación en base *b* corresponde a la lista *l* (es decir, de forma que *deBase b* sea la función inversa a *aBase b*). El parámetro *b* que indica la base será siempre un número entero estrictamente mayor que 1, el número *n* será siempre un entero positivo y los elementos de la lista *l* serán siempre enteros no negativos estrictamente menores que *b*.

Por tanto, estas definiciones deberán comportarse de forma que las siguientes expresiones de tipo *bool* tengan todas valor *true* en *ocaml*:

*aBase 10 1234 = [4;3;2;1]
aBase 100 1234 = [34; 12]
aBase 2 11 = [1;1;0;1]
aBase 3 11 = [2;0;1]
aBase 16 200 = [8;12]*

*deBase 10 [4;3;2;1] = 1234
deBase 100 [34; 12] = 1234
deBase 2 [1;1;0;1] = 11
deBase 3 [2;0;1] = 11
deBase 16 [8;12] = 200*

y *deBase b (aBase b n) = n*, siempre que *b* y *n* cumplan los requisitos arriba mencionados.

```
let rec aBase a b = if b < a then [b]
  else [b mod a] @ aBase a (b/a)
```

```

let rec debase b e = match l with
  () → raise (Failure "debase")
  | h :: t → h
  | h :: t → h + b + debase h t;;

```

4. (2 puntos) Dada la siguiente definición en ocamí para los tipos de datos 'a arbolgen (que sirven para representar árboles con nodos etiquetados con valores de tipo 'a, en los que de cada nodo puede colgar cualquier número de ramas)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

de modo que, por ejemplo, el valor ag1 definido por

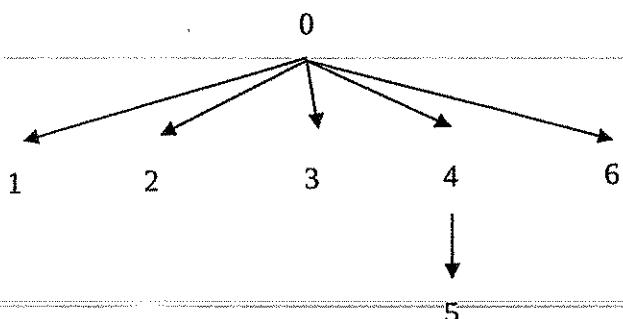
```

let ag1 = Nodo (0, [Nodo (1, []);
  Nodo (2, []);
  Nodo (3, []);
  Nodo (4, [Nodo (5, [])]);
  Nodo (6, [])]);

```

ELENA
DELAMANDO
FREIJE

correspondería al árbol



defina una función hojas : 'a arbolgen -> 'a list
que devuelva la lista de valores asociados a las hojas del árbol, de izquierda a derecha, de forma que $hojas\ ag1 = [1; 2; 3; 5; 6]$

```

let rec hojas = function
  Nodo (r, []) → [r]
  | Nodo (r, h :: t) → if t = [] then
    hojas h
    else hojas h @ hojas (Nodo (r, t));;

```

PROGRAMACIÓN DECLARATIVA
3 DE FEBRERO DE 2005

NOMBRE: _____

I.I.

I.T.I.G

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *top-level de ocaml*:

let x = let x = 0 in x, x+1, x+2;;

val x : int * int * int = (0, 1, 2)

let par = let f x = 2 * x in let g x = f x * x in (f, g);;

val par : (int → int) * (int → int) = (<fun>, <fun>)

let doble p x = fst p x, snd p x;;

val doble : ('a → 'b) * ('a → 'c) → 'a → 'b * 'c = <fun>

let rap = doble par in rap 2;;

- : int * int = (4, 8)

let rec g l x = match l with [] → x | h :: t → g t (h x);;

val g : ('a → 'a) list → 'a → 'a = <fun>

let rec suces gen x0 =
 function 0 → [] | n → x0 :: suces gen (gen x0) (n-1);;

val suces : ('a → 'a) → 'a → int → 'a list = <fun>

let lista = suces ((+) 1) 0 5;;

val lista : int list = [0; 1; 2; 3; 4]

g (List.map (+) lista) 10;;

- : int = 20

2. (2 puntos) Realice una nueva definición para la función *sumpro* definida a continuación, de forma que sólo se utilice recursividad terminal.

```
let rec sumpro n =  
    if n < 1 then (0,1)  
    else let (s,p) = sumpro (n-1) in  
        (s+n, p*n);;
```

```
let sumpro n  
    let rec aux (i,j) x =  
        if x < 1 then sum(i,j)  
        else aux(i+1,j*x) (x-1)  
  
in (0,1) n
```

3. (2 puntos) Defina una función *aBase*: $\text{int} \rightarrow \text{int} \rightarrow \text{int list}$ tal que *aBase b n* devuelva la lista de enteros correspondiente a los dígitos de la representación en base *b* del número *n* (de forma que la cabeza de la lista corresponda al dígito menos significativo, y una función *deBase*: $\text{int} \rightarrow \text{int list} \rightarrow \text{int}$ tal que *deBase b l* devuelva el número entero cuya representación en base *b* corresponde a la lista *l* (es decir, de forma que *deBase b* sea la función inversa a *aBase b*). El parámetro *b* que indica la base será siempre un número entero estrictamente mayor que 1, el número *n* será siempre un entero positivo y los elementos de la lista *l* serán siempre enteros no negativos estrictamente menores que *b*).

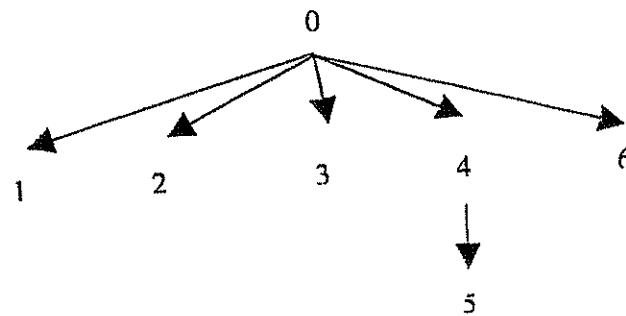
4. (2 puntos) Dada la siguiente definición en *ocaml* para los tipos de datos '*a arbolgen*' (que sirven para representar árboles con nodos etiquetados con valores de tipo '*a*', en los que de cada nodo puede colgar cualquier número de ramas)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

de modo que, por ejemplo, el valor *ag1* definido por

```
let ag1 = Nodo (0, [Nodo (1, []);  
                     Nodo (2, []);  
                     Nodo (3, []);  
                     Nodo (4, [Nodo (5, [])]);  
                     Nodo (6, [])])
```

correspondería al árbol



defina una función *hojas* : '*a arbolgen* -> '*a list*
que devuelva la lista de valores asociados a las hojas del árbol, de izquierda a
derecha, de forma que *hojas ag1* = [1; 2; 3; 5; 6]

Let rec *hojas* = *function*

Nodo (r, t) -> r :: t

| Nodo (r, h::t) -> *hojas h*

| Nodo (r, h::t) -> *hojas h @ hojas (Nodo(r,t))*

PROGRAMACIÓN DECLARATIVA
16 DE DICIEMBRE DE 2005

NOMBRE: _____

I.I.o

I.T.I.G o

1. (5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de *ocaml*:

let id = function x -> x;;

let cte k = función _ -> k;;

(cte 0) "a";;

let rec genlist f = function
 0 -> []
 | n -> (genlist f (n-1)) @ [f n];;

let l1,l2 = let dela n = genlist.id n
 in dela 4, dela 5;;

let l3 = let rep x n = genlist (cte x) n in rep 5 2;;

let rec reduce f e = function [] -> e | h::t -> f h (reduce f e t);;

let sigma = reduce (+) 0;;

let pi = reduce (*) 1;;

let l4 = (List.map sigma [l1; l2; l3]) in (sigma l4, pi l4);;

2. (1 punto) Considere la siguientes definiciones escritas en ocaml

```
type 'a bintree = Empty | Node of ('a * 'a bintree * 'a bintree);;
let rec g = function
  Empty -> 0
  | Node (r,i,d) -> if g d > 2 * g i then r + g d / 2
    else r + g i / 3;;
```

La definición de la función *g* contiene un error de diseño que no afecta al resultado de la función, pero sí gravemente a la eficiencia del cálculo.

Redefina la función *g*, cambiando su definición lo menos posible, de forma que se corrija ese problema de eficiencia.

3. (2 puntos) Escriba una definición alternativa para la función *f*, de modo que sólo se utilice recursividad terminal

```
let rec f = function
  0 -> 0
  | 1 -> 1
  | n -> if n > 0 then 2*f(n-1) - 3*f(n-2)
    else raise (Failure "f");;
```

↑ Fibonacci

4. (2 puntos) Defina una función `criba` : ('a → bool) list → 'a list list, de forma que `criba [p1; p2; ...; pn] l` devuelva una lista de listas cuyo primer elemento sea la lista de elementos de `l` en los que se cumple el predicado `p1`; el segundo, la lista de elementos de `l` que no cumplen `p1`, pero sí `p2`; ...; el penúltimo, la lista de elementos de `l` que cumplen el predicado `pn`, pero ninguno de los anteriores; y el último, la lista de elementos de `l` que no cumplen ninguno de los predicados. En cada una de estas listas los elementos deben conservar entre sí el mismo orden relativo que tenían dentro de `l`.

Así, por ejemplo, ha de verificarse que

```
criba [(function x -> x mod 2 = 0); (function x -> x mod 3 = 0); (<)
0]
[-10;-9;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9;10] =
[[[-10; -8; -6; -4; -2; 0; 2; 4; 6; 8; 10];
 [-9; -3; 3; 9];
 [1; 5; 7];
 [-7; -5; -1]]]
```


PROGRAMACIÓN DECLARATIVA
16 DE DICIEMBRE DE 2005

NOMBRE: _____

I.I.O I.T.I.G.O

1. (5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de *ocaml*:

let id = function x -> x;;

val id : 'a -> 'a = <fun>

let cte k = function _ -> k;;

val cte : 'a * -> 'b -> 'a = <fun>

(cte 0) "a";

- : int = 0

let rec genlist f = function
| 0 -> []
| n -> (genlist f (n-1)) @ [f n];;

val genlist : (int -> 'a) -> int -> 'a list = <fun>

let l1, l2 = let de1a n = genlist id n
in de1a 4, de1a 5;;

val l1 : int list = [1; 2; 3; 4]
val l2 : int list = [1; 2; 3; 4; 5]

let l3 = let rep x n = genlist (cte x) n in rep 5 2;;

val l3 : int list = [5; 5]

let rec reduce f e = function [] -> e | h::t -> f h (reduce f e t);;

val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>

let sigma = reduce (+) 0;;

val sigma : int list -> int = <fun>

let pi = reduce (*) 1;;

val pi : int list -> int = <fun>

let l4 = (List.map sigma [l1; l2; l3]) in (sigma l4, pi l4);;

- : int * int = [25, 1500]

2. (1 punto) Considere la siguientes definiciones escritas en ocaml

```
type 'a bintree = Empty | Node of ('a * 'a bintree * 'a bintree);;

let rec g = function
  Empty -> 0
  | Node (r,i,d) -> if g d > 2 * g i then r + g d / 2
    else r + g i / 3;;
```

La definición de la función *g* contiene un error de diseño que no afecta al resultado de la función, pero sí gravemente a la eficiencia del cálculo.

Redefina la función *g*, cambiando su definición lo menos posible, de forma que se corrija ese problema de eficiencia.

3. (2 puntos) Escriba una definición alternativa para la función *f*, de modo que sólo se utilice recursividad terminal

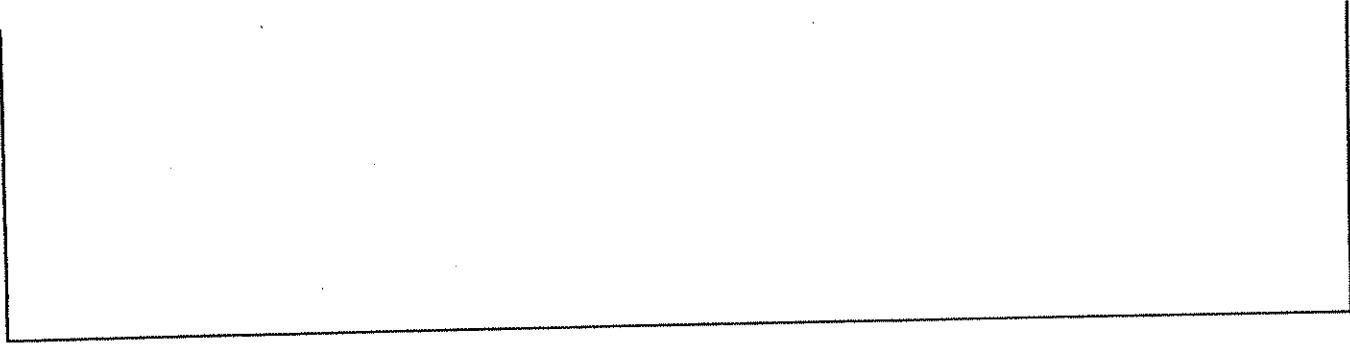
```
let rec f = function
  0 -> 0
  | 1 -> 1
  | n -> if n > 0 then 2*f(n-1) - 3*f(n-2)
    else raise (Failure "f");;
```

= fibonacci

4. (2 puntos) Defina una función ***criba*** : ('a -> bool) list -> 'a list list, de forma que ***criba*** [$p_1; p_2; \dots; p_n$] l devuelva una lista de listas cuyo primer elemento sea la lista de elementos de l en los que se cumple el predicado p_1 ; el segundo, la lista de elementos de l que no cumplen p_1 , pero sí p_2 ; ...; el penúltimo, la lista de elementos de l que cumplen el predicado p_n , pero ninguno de los anteriores; y el último, la lista de elementos de l que no cumplen ninguno de los predicados. En cada una de estas listas los elementos deben conservar entre sí el mismo orden relativo que tenían dentro de l .

Así, por ejemplo, ha de verificarse que

```
criba [(function x -> x mod 2 = 0); (function x -> x mod 3 = 0); (<) 0]
      [-10;-9;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9;10] =
[[[-10; -8; -6; -4; -2; 0; 2; 4; 6; 8; 10];
  [-9; -3; 3; 9];
  [1; 5; 7];
  [-7; -5; -1]]]
```



PROGRAMACIÓN DECLARATIVA

14 DE FEBRERO DE 2006

NOMBRE: _____

I.I.

I.T.I.G

1. (5.5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de *ocaml*:

```
let f, x = (+), 0;;
val f : int -> int -> int = <fun>
val x : int = 0
f x;;
- : int -> int = <fun>
```

```
let y = x + 1, x - 1;;
val y : int * int = (1, -1)
```

```
let a, b = y in if a > b then b else a;;
- : int = -1
```

```
let z = let a, b = y in let z = a - b in z * z;;
val z : int = 4
```

```
(function x -> x) (function s -> s ^ s);;
- : string -> string = <fun>
```

```
let rec itera op = function [] -> () | h::t -> op h; itera op t;;
val itera : ('a -> 'b) -> 'a list -> unit = <fun>
```

```
let rec clist n x = if n < 1 then [] else x :: clist (n-1) x;;
val clist : int -> 'a -> 'a list = <fun>
```

```
clist 3 true;;
- : bool list = [true; true; true]
```

```
let rec powerr n op x = if n <= 1 then x else powerr (n / 2) op (op x x);;
val powerr : int -> ('a -> 'a -> 'a) -> 'a -> 'a = <fun>
```

```
let g = powerr 5 (+) in g 3;;
- : int = 12
```

2. (1.5 puntos) Considere la siguiente definición en ocaml:

```
let rec comp l x = match l with
    [] -> x
    | h::t -> h (comp t x);;
```

¿Cuál es el tipo de la función *comp*?

```
val comp: ('a -> 'a) list -> 'a -> 'a = <fun>
```

Realice una nueva definición para *comp* de forma que sólo se utilice recursividad terminal.

```
let comp l x =
  let rec aux res = function
    [] -> res
    | h::t -> aux (h res) t
  in aux x (List.rev l);;
```

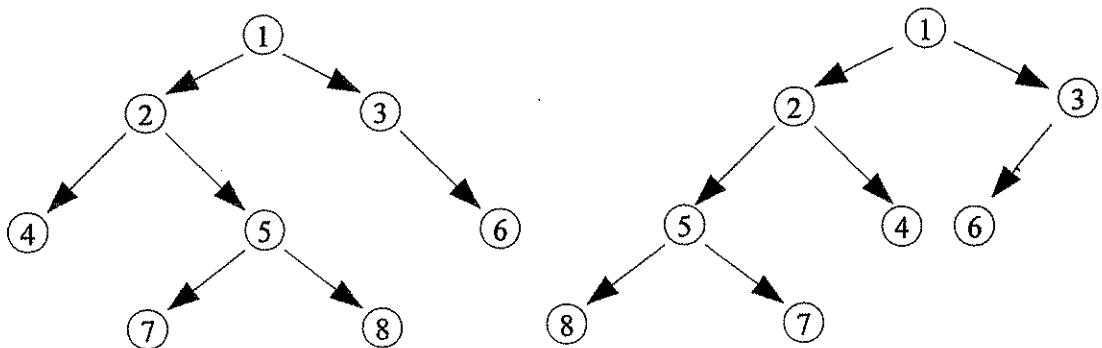
3. (1.5 puntos) La función *max_en* está definida de forma que el valor de *max_en* x / corresponde a la función de la lista / que alcanza el mayor valor en el punto x. Redefínala para optimizar su eficiencia.

```
let rec max_en x = function
  [] -> raise (Failure "max_en")
  | [f] -> f
  | f::l -> if f x > (max_en x l) x then f
               else max_en x l;;
```

```
let max_en x = function
  [] -> raise (Failure "max_en") | [f] -> f
  | f::l -> let rec aux fmax vmax = function
              [] -> fmax
              | h::t -> let hx = h x in
                           if hx > vmax then aux h hx t
                           else aux fmax vmax t
  in aux f (f x) l;;
```

4. (1.5 puntos) Diremos que un árbol binario es un "giro" de otro árbol binario si el primero puede obtenerse del segundo intercambiando las ramas de cualesquiera de sus nodos.

Así, por ejemplo, los dos árboles siguientes son el uno "giro" del otro, pues el segundo se puede obtener a partir del primero intercambiando las ramas de los nodos "2", "3", y "5".



Utilizando, para representar los árboles binarios, el tipo de dato '*a bintree*' definido a continuación, implemente en ocaml una función *giro*: '*a bintree* -> '*a bintree* -> *bool* que indique si dos árboles son el uno giro del otro.

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

```
let rec giro a1 a2 = match (a1,a2) with  
    Empty, Empty -> true  
  | Node (i1,r1,d1), Node (i2,r2,d2) ->  
      r1 = r2 &&  
      ((giro i1 i2 && giro d1 d2) ||  
       (giro i1 d2 && giro d1 i2))  
  | _ -> false;;
```

PROGRAMACIÓN DECLARATIVA
14 DE FEBRERO DE 2006

NOMBRE: _____

I.T.I.G

1. (5.5 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el compilador de *ocaml*:

```
let f, x = (+), 0;;
val f : int -> int = <fun>
val x : int = 0
f x;;
- : int -> int = <fun>
```

```
let y = x + 1, x - 1;;
val y : int * int = (1, -1)
```

ELENA
DELAMANDO
FREIJE

```
let a, b = y in if a > b then b else a;;
- : int = -1
```

```
let z = let a, b = y in let z = a - b in z * z;;
val z : int = 4
```

```
(function x -> x) (function s -> s ^ s);;
- : string -> string = <fun>
```

```
let rec itera op = function [] -> () | h::t -> op h; itera op t;;
val itera : ('a -> 'b) -> 'a list -> unit = <fun>
```

```
let rec clist n x = if n < 1 then [] else x :: clist (n-1) x;;
val clist : int -> 'a -> 'a list = <fun>
```

```
clist 3 true;;
- : bool list = [true; true; true]
```

```
let rec powerr n op x = if n <= 1 then x else powerr (n / 2) op (op x x);;
val powerr : int -> ('a -> 'a -> 'a) -> 'a -> 'a = <fun>
```

```
let g = powerr 5 (+) in g 3;;
- : int = 12
```

← (7.5 puntos) Considere la siguiente definición en ocaml:

```
let rec comp l x = match l with
    [] -> x
    | h::t -> h (comp t x);;
```

¿Cuál es el tipo de la función *comp*?

```
val comp: ('a -> 'a) list -> 'a -> 'a = <fun>
```

Realice una nueva definición para *comp* de forma que sólo se utilice recursividad terminal.

```
let comp l x =
  let rec aux res = function
    [] -> res
    | h::t -> aux (h res) t
  in aux x (List.rev l);;
```

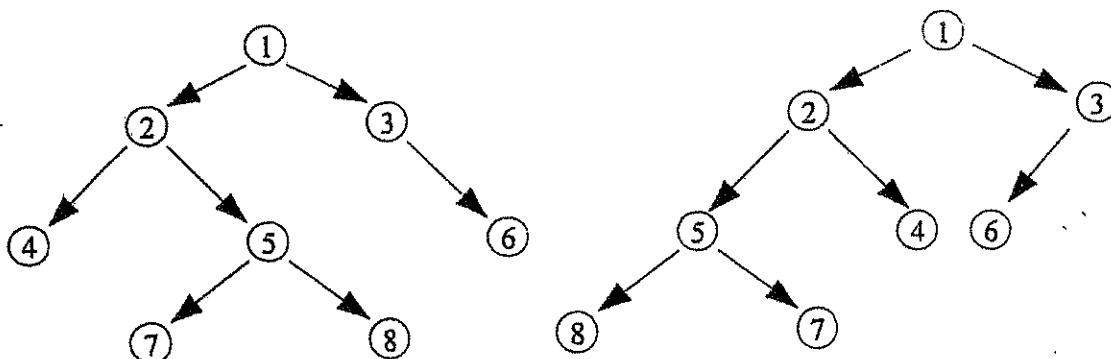
3. (1.5 puntos) La función *max_en* está definida de forma que el valor de *max_en x* / corresponde a la función de la lista *l* que alcanza el mayor valor en el punto *x*. Redefinala para optimizar su eficiencia.

```
let rec max_en x = function
  [] -> raise (Failure "max_en")
  | [f] -> f
  | f::l -> if f x > (max_en x l). x then f
    else max_en x l;;
```

```
let max_en x = function
  [] -> raise (Failure "max_en") | [f] -> f
  | f::l -> let rec aux fmax vmax = function
    [] -> fmax
    | h::t -> let hx = h x in
      if hx > vmax then aux h hx t
      else aux fmax vmax t
  in aux f (f x) l;;
```

4. (1.5 puntos) Diremos que un árbol binario es un "giro" de otro árbol binario si el primero puede obtenerse del segundo intercambiando las ramas de cualesquiera de sus nodos.

Así, por ejemplo, los dos árboles siguientes son el uno "giro" del otro, pues el segundo se puede obtener a partir del primero intercambiando las ramas de los nodos "2", "3", y "5".



Utilizando, para representar los árboles binarios, el tipo de dato `'a bintree` definido a continuación, implemente en ocaml una función `giro: 'a bintree -> 'a bintree -> bool` que indique si dos árboles son el uno giro del otro.

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

```
let rec giro a1 a2 = match (a1,a2) with  
  Empty, Empty -> true  
  | Node (i1,r1,d1), Node (i2,r2,d2) ->  
    r1 = r2 &&  
    ((giro i1 i2 && giro d1 d2) ||  
     (giro i1 d2 && giro d1 i2))  
  | _ -> false;;
```

PROGRAMACIÓN DECLARATIVA
13 DE SEPTIEMBRE DE 2006

NOMBRE: _____

I.I.

I.T.I.G

1. (6 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *top level* de *ocaml*:

let x, y = 2, 5;;

- val x : int = 2
- val y : int = 5

let f y = x + y;;

- val f : int → int = <fun>

f (let f = function x -> x * x in f x);;

- : int = 6

let v x f = f x;;

- val v: 'a → ('a → 'b) → 'b = <fun>

let cero x = v 0 x and dos x = v 2 x;;

- val cero : (int → 'a) → 'a = <fun>
- val dos : (int → 'a) → 'a = <fun>

```
let g = cero (-) in g 1;;
```

```
- : int = -1
```

```
let h = dos (+);;
```

```
val h : int → int = <fun>
```

```
dos h;;
```

```
- : int = 4
```

ELENA
DELAMANDO
FREITJE

```
g 0, h 0;;
```

Error. g no está definida
Unbound value g

```
let doble = let vacia = "" in function  
    vacia → vacia  
    | s → s ^ s;;
```

```
val doble : string → string = <fun>
```

```
doble "hola";;
```

```
- : string = "hola"
```

```
let rec ap = function  
    [] → 0 | h :: t → h (ap t);;
```

```
val ap : (int → int) list → int = <fun>
```

2. (2 puntos) Realice una nueva definición para la función f , de forma que sólo se utilice recursividad terminal.

```
let rec f = function
[] -> 0
| h::t -> h + 2 * f t;;
```

let $f l =$

~~let rec aux = function~~

~~([], x) -> x~~

~~| (h::t, x) -> aux(t, h+2 * x)~~

~~in aux(list.rev l, 0);;~~

let $f l =$

~~let rec aux accu resto = function~~

~~[] -> accu~~

~~| h::t -> aux(h+2 * accu, resto)~~

~~in 0 l ii~~

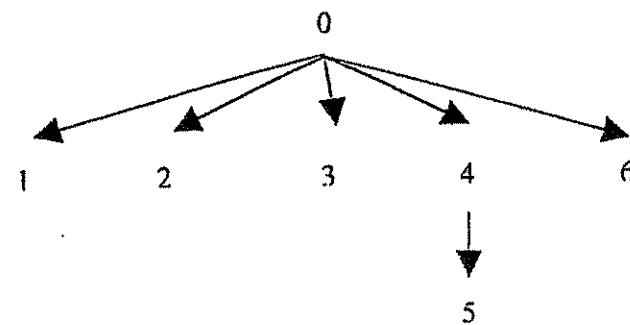
3. (2 puntos). Considere la siguiente definición en ocaml para el tipo de dato 'a arbolgen (que sirve para representar árboles con nodos etiquetados con valores de tipo 'a, en los que de cada nodo puede colgar cualquier número de ramas)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

de modo que, por ejemplo, el valor *ag1* definido por

```
let ag1 = Nodo (0, [Nodo (1, []); Nodo (2, []);  
                     Nodo (3, []); Nodo (4, [Nodo (5, [])]);  
                     Nodo (6, [])])
```

correspondería al árbol



Defina una función *nnodos* : '*a arbolgen* -> *int*
que devuelva el número de nodos de un árbol, de forma que, por ejemplo,
nnodos ag1 = 7

PROGRAMACIÓN DECLARATIVA

13 DE SEPTIEMBRE DE 2006

1. (6 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *toplevel* de *ocaml*:

```
let x, y = 2, 5;;           val x : int = 2
                           val y : int = 5
let f y = x + y;;          val f : int -> int = <fun>
f (let f = function x -> x * x in f x);; ; int = 6
let v x f = f x;;          val v : 'a -> ('a -> 'a) -> 'b = <fun>
let cero x = v 0 x and dos x = v 2 x;;
let g = cero (-) in g 1;;
let h = dos (+);;
dos h;;
g 0, h 0;;
let doble = let vacia = "" in function
vacia -> vacia
| s -> s ^ s;;
doble "hola";;
let rec ap = function
[] -> 0 | h :: t -> h (ap t);;
```

2. (2 puntos) Realice una nueva definición para la función *f*, de forma que sólo se utilice recursividad terminal.

```
let rec f = function
[] -> 0
```

EXAMEN ENERO 2015

let $x = \langle 1, 2 \rangle; \langle 3, 4 \rangle \rangle;;$
val $x = (\text{int} \times \text{int}) \text{list} = \langle (1, 2); (3, 4) \rangle;$
let $x :: y = x \text{ in } x :: x :: y;;$
 $(\text{int} \times \text{int}) \text{list} = \langle (1, 2), (1, 2); (3, 4) \rangle$

let $(x, y) :: _ = \text{list. tl } x;;$
val $x : \text{int} = 3$
val $y : \text{int} = 4$
 $x + (\text{let } x = \underline{x} + y \text{ in } x + y);;$
 $3 + (\text{let } x = 3 + 4 \text{ in } 3 + 4) = 7$

(function $x \rightarrow x, x$) "holá";
string * string = ("holá", "holá")

let rec pad $x = \text{function}$
 $\quad [] \rightarrow []$
 $\quad | h :: t \rightarrow (x, h) :: \text{pad } x t;;$
 $\quad | a \rightarrow | b \rightarrow (a * b) \text{ list} = \langle \text{from} \rangle$

pad $\ell [2; 3; 4];;$
 $\left\{ \begin{array}{l} L = \text{empty} \\ A = (\text{int} \times \text{int}) \text{list} = \langle (1, 2); (1, 3); (1, 4) \rangle \end{array} \right.$

let rec lprod tl tl = match tl with

[],] → []

| h::tl → prod h tl @ lprod tl tl;;

vale lprod : 'a list → 'b list → ('a * 'b) list = <fun>

lprod [1;2] ['a';'b';'c'];;

[(1,'a');(1,'b');(1,'c');(2,'a');(2,'b');(2,'c')]

2. Redefina la función prod con la función List.map, sin utilizar directamente recursividad.

let prod

3. Indique el tipo de la función f:

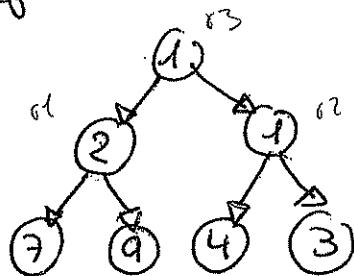
let rec f = function x::y::tl → f(y::tl) & & x <= y
| _ → true;;

f: 'a list → bool = <fun>

Es definición recursiva termina: si no lo fueran definirla de nuevo

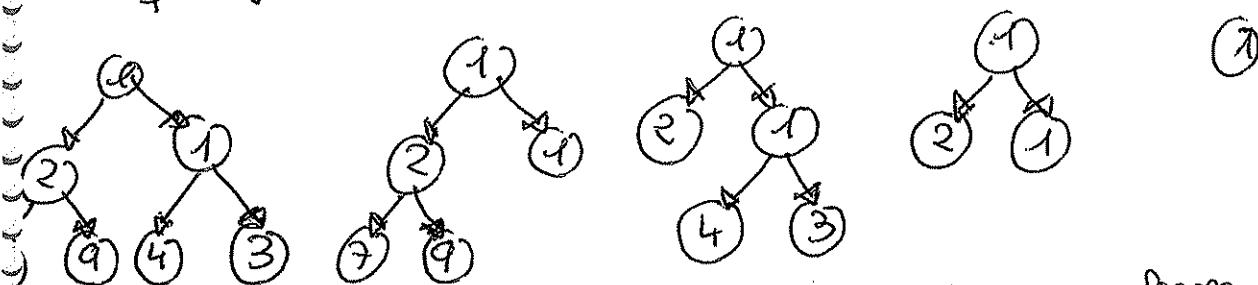
4. type 'a tree = L of 'a | T of 'a * 'a tree * 'a tree

a) Representación



Let $t = T(1, T(2, L(7), L(9)), T(3, L(4), L(8)))$

b) Llamarímos segmentos raíz de un árbol a todos aquellos árboles que pueden obtenerse a partir de este "podándolo", ejemplo:



Defina es_segrz : 'a tree \rightarrow 'a tree \rightarrow bool', de forma que ~~es_segrz~~
 $\text{es_segrz } t_1 \text{ } t_2$ Indique si t_1 es o no un segmento raíz de t_2

t_2

let rec $\text{es_segrz } t_1 \text{ } t_2 = \text{match } t_1, t_2 \text{ with}$

$| (a), T(_, _, _) \rightarrow \text{if } a = \text{root } t_2 \text{ then true else false}$

$| T(a1, i1, d1), T(i2, d2) \rightarrow \text{es_segrz } a1 \text{ } T(i2, d2)$

Indique si las siguientes clases son correctas:

```
class claseB pc1 = object
    val atributo1 = pc1
    method metodo1 () = String.length(atributo1)
    method metodo2 pm1 = atributo1 ^ pm1
```

end;;

es correcta

```
class subclassB1 pc1 pc2 = object
    inherit claseB pc1
    val atributo2 = pc2
    method metodo2 pm1 =
        atributo1 ^ atributo2 ^ pm1
```

end;;

es correcta

```
class subclassB2 pc1 pc2 = object
```

inherit claseB pc1

val atributo2 = pc2

method metodo2 pm1 pm2 =

atributo1 ^ atributo2 ^ pm1 ^ pm2

end;;

Error: El método 2 tiene tipo string → a → 'b

pero se ha escrito string → string

Tipo 'a → 'b no es compatible con el tipo
string

```
| h::t -> h + 2 * f t;;
```

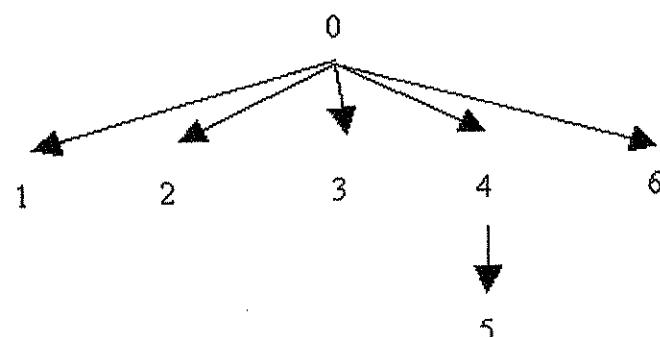
3. (2 puntos) Considere la siguiente definición en *ocaml* para el tipo de dato '*a arbolgen* (que sirve para representar árboles con nodos etiquetados con valores de tipo '*a*, en los que de cada nodo puede colgar cualquier número de ramas)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

de modo que, por ejemplo, el valor *ag1* definido por

```
let ag1 = Nodo (0, [Nodo (1, []); Nodo (2, []);  
Nodo (3, []); Nodo (4, [Nodo (5, [])]);  
Nodo (6, [])])
```

correspondería al árbol



<>Defina una función *nnodos* : '*a arbolgen* -> *int* que devuelva el número de nodos de un árbol, de forma que, por ejemplo,
nnodos ag1 = 7



PROGRAMACIÓN DECLARATIVA

13 DE SEPTIEMBRE DE 2006

1. (6 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *toplevel* de *ocaml*:

```
let x, y = 2, 5;;
  val x : int = 2
  val y : int = 5
let f y = x + y;;
  val f : int -> int = <fun>
f (let f = function x -> x * x in f x);;
  - : int = 6
let v x f = f x;;
  val v : 'a -> ('a -> 'b) -> 'b = <fun>
let cero x = v 0 x and dos x = v 2 x;;
  val cero : (int -> 'a) -> 'a = <fun>
  val dos : (int -> 'a) -> 'a = <fun>
let g = cero (-) in g 1;;
  - : int = -1
let h = dos (+);;
  val h : int -> int = <fun>
dos h;;
  - : int = 4
g 0, h 0;;
Characters 0-1: g 0, h 0;; Unbound value g
let doble = let vacia = "" in function
vacia -> vacia
| s -> s ^ s;;
Characters 16-21:
let doble = let vacia = "" in function
      ^^^^^^
Warning Y: unused variable vacia.
Characters 57-58:
Warning U: this match case is unused. | s -> s ^ s;;
val doble : string -> string = <fun>
```

ELENA
DELAMANO
FREIJJE

```

doble "hola";;
  - : string = "hola"
let rec ap = function
[] -> 0 | h::t -> h (ap t);;
val ap : (int -> int) list -> int = <fun>

```

2. (2 puntos) Realice una nueva definición para la función f , de forma que sólo se utilice recursividad terminal.

```

let rec f = function
[] -> 0
| h::t -> h + 2 * f t;;

```

3. (2 puntos) Considere la siguiente definición en *ocaml* para el tipo de dato '*a arbolgen*' (que sirve para representar árboles con nodos etiquetados con valores de tipo '*a*', en los que de cada nodo puede colgar cualquier número de ramas)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;
```

de modo que, por ejemplo, el valor *ag1* definido por

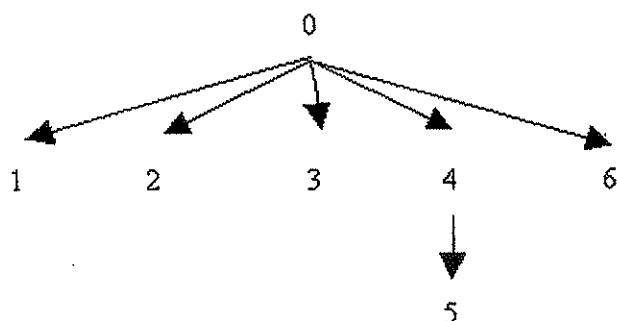
```

let ag1 = Nodo (0, [Nodo (1, []); Nodo (2, []);
Nodo (3, []); Nodo (4, [Nodo (5, [])]);
Nodo (6, [])])

```

correspondería al árbol

<>Defina una función *nnodos* : '*a arbolgen* -> int' que devuelva el número de nodos de un árbol, de forma que, por ejemplo,
nnodos ag1 = 7



(*****
Examen de Sep 06
*****)

(* 1. (6 puntos) Escriba el resultado de la compilaciÃ³n y ejecuciÃ³n de las siguientes frases, con tipos y valores, como lo indicarÃa el toplevel de ocaml: *)

let x, y = 2, 5;;

(*
val x : int = 2
val y : int = 5
*)

let f y = x + y;;

(*
val f : int -> int = <fun>

Es la funcion sumar dos

*)

f (let f = function x -> x * x in f x);;

(*
(let f = function x -> x * x in f x);;

Eso vale 4. Al aplicarle f, da 6

- : int = 6

*)

let v x f = f x;;

(*
val v : 'a -> ('a -> 'b) -> 'b = <fun>
*)

let cero x = v 0 x and dos x = v 2 x;;

(*
val cero : (int -> 'a) -> 'a = <fun>
val dos : (int -> 'a) -> 'a = <fun>
*)

let g = cero (-) in g 1;;

(*
- : int = -1;
*)

let h = dos (+);;

(*
val h : int -> int = <fun>
*)

dos h;;

(*

```

    - : int = 4
*)

g 0, h 0;;
(*
No existe g, esta declarada dentro de un let in
Unbound value g
*)

let doble = let vacia = "" in function
    vacia -> vacia
    | s -> s ^ s;;

(*
vacia no es "", si no, el parametro de la funcion; siempre devolvera el string
pasado
val doble : string -> string = <fun>
*)

doble "hola";;

(*
- : string = "hola"
*)

let rec ap = function
    [] -> 0
    | h::t -> h (ap t);;

(*
val ap : (int -> int) list -> int = <fun>

```

Ejemplo:

```

# ap [(-) 2; (+) 5];;
Equivale a
(-) 2 ap [(+) 5];;
Equivale a
(-) 2 5
que es restarle 5 a 2:
- : int = -3
*)


```

(* 2. (2 puntos) Realice una nueva definiciÃ³n para la funciÃ³n f , de forma que se utilice recursividad terminal.*)

```

let rec f = function
[] -> 0
| h::t -> h + 2 * f t;;
f [3;2;4];;

(*
- : int = 23
3 + 2 * (f [2;4]) = 3 + 2 * (2 + 2*(f [4])) = 3 + 2 * (2 + 2 * (4 + 2*(f [1]))) =
3 + 2 * (2 + 2 * (4 + 2 * 0)) = 23

```

ELENA
DELAMANZO
FREIJE

```
*)  
  
let f l =  
    let rec aux = function  
        [],x -> x  
        | h::t, x -> aux (t, h+2*x)  
    in aux (List.rev l, 0);;  
  
f [3;2;4];;  
  
(*  
- : int = 23  
aux [4;2;3], 0  
aux [2;3], 4+2*0  
aux [3], 2+2*4  
aux [], 3+2*10  
23  
*)
```

(* 3. (2 puntos) Considere la siguiente definiciÃ³n en ocaml para el tipo de dato arbolgen (que sirve para representar Ã©rboles con nodos etiquetados con valores de tipo 'a, en los que de cada nodo puede colgar cualquier nÃºmero de ramas) *)

```
type 'a arbolgen = Nodo of 'a * 'a arbolgen list;;  
  
(* de modo que, por ejemplo, el valor ag1 definido por *)
```

```
let ag1 = Nodo (0, [Nodo (1, []); Nodo (2, []);  
Nodo (3, []); Nodo (4, [Nodo (5, [])]);  
Nodo (6, [])])
```

(* corresponderÃ¡ al Ã©rbol

<>Defina una funciÃ³n nnodos : 'a arbolgen -> int que devuelva el nÃºmero de nodos de un Ã©rbol, de forma que, por ejemplo,
nnodos ag1 = 7 *)

```
(* let nnodos (Nodo (_,l)) =  
    let rec aux = function  
        suma, Nodo (_,[]) -> suma +1  
        | suma, Nodo (_,h::t) -> aux (suma, (Nodo(_,t)))  
    in aux 0, Nodo(_,l);; *)  
  
(* let rec nnodos = function  
    [] -> 0  
    | Nodo (_, h::t) -> 1 + nnodos (Nodo(_, t));;*)
```

(*****
Funciones dadas en clase
*****)

```
let rec input_string_list en =  
    try  
        let h = input_line en in  
        h :: (input_string_list en)
```

```

        with End_of_file -> [];

let rec output_string_list sal = function
    [] -> []
  | h::t -> output_string sal (h^"\n"); output_string_list sal t;;

let copy_textfile en sal =
  let entrada = open_in en and
      salida = open_out sal in
  output_string_list salida (input_string_list entrada);
  close_in entrada;
  close_out salida;;

(* copy_textfile "archivo.ml" "salida"
cat salida *)

let crono f x =
  let t = Sys.time() in f x; Sys.time()-.t;;

let fib n =
  let rec aux (a, f, i) = if i=n then f
    else aux (f, a+f, i+1)
  in aux (0,n,1);;

(*****)

type persona = {nombre:string; edad: int};;

{nombre="Pepe";edad=19};;
(* - : persona = {nombre = "Pepe"; edad = 19} *)
{edad=19; nombre="Maria"};;
(* - : persona = {nombre = "Maria"; edad = 19} *)
{edad=19;nombre="Hola"}.edad;;
(* - : int = 19 *)

let edad p = p.edad;;
let edad {edad = n; nombre = _} = n;;
let envejece p = {nombre = p.nombre; edad = p.edad +1};;
type persona = {mutable nombre:string; mutable edad: int};;
let p = {nombre="Pepe";edad=19};;
p.edad <- 20;;
(* - : unit = () *)

p;;
(* - : persona = {nombre = "Pepe"; edad = 20} *)
let envejece p = p.edad <- p.edad +1;;

let turn_o_matic = let file = open_in "nombre" in
                    let n = input_binary_int file in

```

```

        close_in_file;
        let file = open_out "nombre" in (
            output_binary_int file (n+1);
            close_out file);
        n;;
    (*******)

type var_int = {mutable valor : int};;
let t = {valor = 0};;
let turn_o_matic () = t.valor <- (t.valor+1) ; t.valor;;

(*******)

type 'a ref = {mutable content: 'a};;
let (!) v = v.content;;
let ref v = {content = v};;
let (:=) v x = v.content <- x;;

let t = ref 0;
let turn_o_matic () = t := !t + 1; !t;;
(*******)

let factorial n = let suma = ref 1 in
    for i=2 to n do
        suma := !suma *i
    done;
    !suma;;
let factorial n = let i = ref 2 and
    f = ref 1 in
    while (i<=n) do
        f := !f * !i;
        i := !i + 1;
    done;
    !f;;
(*******)

let rec l_ordenada f = function
    [] -> true
    | _ :: [] -> true
    | h1::h2::t -> if (f h1 h2) then (l_ordenada f (h2::t)) else false;;

```

PROGRAMACIÓN DECLARATIVA

22 de diciembre 2006

NOMBRE: _____

I.I. □

I.T.I.G □

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el toplevel de ocam!

let x, y = "hola", "adiós";;

Val x : string = "hola"

Val y : string = "adiós"

let x y = x ^ y;;

Val x : string \rightarrow string = <fun>

let x = x "pepe" in x;;

: string = "halapepe"

let x::y::z = [1] @ [2] @ [3];;

Val x : int = 1

Val y : int = 2

Val z : int list = [3]

let x::y::z = [1] :: [2] :: [3] :: [];;

Val x : int list = [1]

Val y : int list = [2]

Val z : int list list = [[3]]

let rec m2 l1 l2 = match (l1, l2) with
 ([], []) -> [] | (a :: b, c :: d) -> a c :: m2 b d;;

; ; ; ; ; ; ; ;

```
m2 [abs] [1; -2];;
```

```
|-----|
```

```
(function x -> x (x 2)) (function x -> 2 * x * x);;
```

```
|-----|
```

2. Diremos que una lista es *sub-lista* de otra si puede obtenerse eliminando algunos elementos de esta. Puesto que cada elemento de una lista de n elementos puede ser o no eliminado para obtener una sub-lista, esta tendrá 2^n sub-listas. Así, por ejemplo, la lista [3;5;3] tendría las siguientes sub-listas: [], [3], [5], [5; 3], [3], [3; 3], [3; 5] y [3; 5; 3]. (Nótese que alguna sub-lista aparece más de una vez debido a que puede ser obtenida eliminando distintos elementos de la lista original).

a. (2 puntos) Defina una función *sublists* : 'a list -> 'a list list que dé todas las sub-listas de una lista dada. (Si lo desea puede optar por no incluir repeticiones en la lista de sub-listas; pero en ese caso debe indicarlo explícitamente).

```
let rec sublists = function
```

```
| [] -> [[]]
```

```
| x::l' -> let aux = sublists l'
```

```
    in aux @ (List.map (fun l -> x::l) aux)
```

```
Es sublista?
```

```
let rec is-sublist l1 l2 = match l1, l2 with
```

```
| [], _ -> true
```

```
| h1::t1, h2::t2 -> if h1=h2 then is-sublist t1 t2  
                           else is-sublist l1 t2
```

```
| _ -> false
```

b. (2 puntos) Defina una función *sublista_de* : '*a list* -> '*a list* -> *bool*, tal que *sublista_de l1 l2* indique si *l2* es, o no, *sub-lista de l1*. (Se tendrá en cuenta la eficiencia de la definición).

3. (2 puntos) Defina, utilizando sólo **recursividad terminal**, una función *apariciones*: '*a* -> '*a list* -> *int* que devuelva el número de veces que aparece un valor en una lista.

PROGRAMACIÓN DECLARATIVA

22 de diciembre 2006

NOMBRE: _____

I.I.

I.T.I.G

1. (4 puntos) Escriba el resultado de la compilación y ejecución de las siguientes frases, con tipos y valores, como lo indicaría el *toplevel* de *ocaml*:

let x, y = "hola", "adiós";;

let x y = x ^ y;;

let x = x "pepe" in x;;

let x::y::z = [1] @ [2] @ [3];;

let x::y::z = [1] :: [2] :: [3] :: [];;

let rec m2 l1 l2= match (l1,l2) with
([],[]) -> [] | (a::b, c::d) -> a c :: m2 b d;;

```
m2 [abs] [1; -2];;
```

```
(function x -> x (x 2)) (function x -> 2 * x * x);;
```

```
-: int = 128
```

2. Diremos que una lista es *sub-lista* de otra si puede obtenerse eliminando algunos elementos de esta. Puesto que cada elemento de una lista de n elementos puede ser o no eliminado para obtener una sub-lista, esta tendrá 2^n sub-listas. Así, por ejemplo, la lista [3;5;3] tendría las siguientes sub-listas: [], [3], [5], [5; 3], [3], [3; 3], [3; 5] y [3; 5; 3]. (Nótese que alguna sub-lista aparece más de una vez debido a que puede ser obtenida eliminando distintos elementos de la lista original).

a. (2 puntos) Defina una función *sublistas* : 'a list -> 'a list list que dé todas las sub-listas de una lista dada. (Si lo desea puede optar por no incluir repeticiones en la lista de sub-listas; pero en ese caso debe indicarlo explícitamente).

b. (2 puntos) Defina una función `sublista_de` : `'a list -> 'a list -> bool`, tal que `sublista_de l1 l2` indique si `l2` es, o no, sub-lista de `l1`. (Se tendrá en cuenta la eficiencia de la definición).

3. (2 puntos) Defina, utilizando sólo recursividad terminal, una función `apariciones`: `'a -> 'a list -> int` que devuelva el número de veces que aparece un valor en una lista.

```
let apariciones x l =  
    let rec aux acum hasta = function  
        [] -> acum  
        h::t -> if h = x then aux (acum+1) t  
                  else aux acum t  
    in aux 0 l
```

Examen Programación Declarativa Febrero 2007

Resuelto por Zarovich

```
# let x f = f,f;;
val x : 'a -> 'a * 'a = <fun>

# let a::b = [x 1; x 2] in (a,b);;
- : (int * int) * (int * int) list = ((1, 1), [(2, 2)])

# let doble x y = x (x y);;
val doble : ('a -> 'a) -> 'a -> 'a = <fun>

# let f = doble (function x -> x * x);;
val f : int -> int = <fun>

# let x = f 2 in x + 1;;
- : int = 17

# let h f = function x -> let c::_ = f x in c;;
val h : ('a -> 'b list) -> 'a -> 'b = <fun>

# let s = h List.tl in s [1;2;3];
- : int = 2

# let s l = h List.tl l;;
val s : 'a list -> 'a = <fun>

# let rec num x = function [] -> 0
    | h::t -> (if x = h then 1 else 0) + num t;;
val num : 'a -> 'a list -> int = <fun>

# num "hola";;
- : string list -> int = <fun>

# let rec pre l s = match (l,s) with
    | ([],_) -> false
    | (_,[]) -> true
    | (h1::t1, h2::t2) -> h1 = h2 && pre t1 t2;;
val pre : 'a list -> 'a list -> bool = <fun>

# let l = ['1';'2';'3'] in
    pre l ['1';'2'], pre l (List.tl l);;
- : bool * bool = (true, false)
```

1) escriba el resultado de la compilación tal y como lo haría el Ocaml:

```

let rec p = function 0 -> true |n -> i(n-1)
and i = function 0 -> false |n -> p(n-1);
(* val p : int -> bool = <fun>
   val i : int -> bool = <fun> *)

p 2, i (-2);
(* recursividad infinita *)

let p n = p(abs n) and i n = i(abs n) in p 2, i (-2);
(* -: bool * bool = (true,false) *)

let x::y::z = [1]@[2];
(* val x: int = 1
   val y: int = 2
   val z: int list = [] *)

z::[];
(* -: int list list = [[]] *)

let x,y = y,x;;
(* val x: int = 2
   val y: int = 1 *)

let f z = z + 2 * y;;
(* val f: int -> int = <fun> *)

f x + f y;;
(* -: int = 7 *)

let y = x + y;;
(* val y: int = 3 *)

f x + f y;;
(* -: int = 9 *)

let p = let x,y = x+y,x-y in y,x;;
(* val p: int*int = (-1,5) *)

let p = x+y,y-x in let x,y = p in y,x;;
(* -: int * int = (1,5) *)

```

ELENA
DELAMANO
FREIJE

2) let f (x,y)=
 let x = abs x and y = abs y in
 let
 a = ref (max x y) and
 b = ref (min x y) in
 while !b <> 0 do
 let temp = !a mod !b in a := !b;
 b := temp
 done;
 !a;

(*pasar la función anterior a una sin referencias, de programación imperativa a declarativa. *)

```

let f (x,y) =
  let rec aux = function
    (a,0) -> a
    |(a,b) -> aux(b, a mod b)
  in aux(max (abs x) (abs y),min (abs x) (abs y));

```

```
3) type 'a arb = R of 'a
   |U of 'a * 'a arb
   |B of 'a * 'a arb * 'a arb;;
(* hacer el recorrido en anchura del arbol
  'a arb -> 'a list *)

let anchura arbol =
  let rec aux = function
    [] -> []
    |R(x)::t -> x:: aux t
    |U(x,i)::t -> x:: aux (t @ [i])
    |B(x,i,d)::t -> x:: aux (t@[i;d])
  in aux [arbol];;
```

PROGRAMACIÓN DECLARATIVA

10 DE DECEMBRO DE 2010

APELIDOS:

NOME:

E.I. □

E.T.I.X. □

1. (5 puntos) Escriba o resultado da compilación e execução das seguintes frases, com tipos e valores, como faria o compilador interativo de *ocaml*:

let five _ = 5;;

val five: 'a → int = <fun>

let id x = x and apply x y = x y;;

val apply: ('a → 'b) → 'a → 'b = <fun>

five 0, id 0, (id five) 0, id (five 0), apply five true;;

-: int * int * int * int * bool = (5, 0, 5, 5, true)

let mx3 x y z = max x (max y z);;

val mx3: 'a → 'a → 'a → 'a = <fun>

mx3 1, mx3 1 2, mx3 1 2 3;;

-: (int → int → int) * (int → int) * int = (<fun>, <fun>, 3)

let rec fold x = function [] -> x | (op,y)::t -> fold (op x y) t;;

'a → (('a → 'b → 'a) * 'b) list

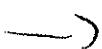
fold 1 [(+, 2; (*), 3; (-), 1; (/), 3)];;

let (|>) x f = f x;;

```
-2 |> abs |> (+) 3 |> function x -> x * x;;
```



```
let x = let x = 1 and y = 2 in x + y, x - y;;
```



2. (1 punto) Reescriba as seguintes definicións sen utilizar definicións locais nin expresións if...then...else...

```
let f x = let (x,y) = x in x;;
```



```
let n x g = if g x then true else false;;
```



3. (3 puntos) Indique o tipo das seguintes funcións:

```
let rec sorted = function
  [] | [] -> true
  | x::y::z -> x <= y && sorted (y::z);;
```

```
let rec merge l1 l2 = match (l1,l2) with
  (l, []) | ([], l) -> l
  | (h1::t1, h2::t2) -> if h1 <= h2 then h1 :: merge t1 l2
                           else h2 :: merge l1 t2;;
```

¿É terminal a recursividade empregada nestas definicións?

Se nalgún caso non é así, realice unha nova definición recursiva terminal para a función correspondente.

Solución adelante →

4. (1 punto) Considere a seguinte definición en ocaml para o tipo de dato ***bitree*** (que serve para representar árbores binarias en ocaml):

```
type bitree = Empty | Node of bitree * bitree;;
```

Dicimos que unha árbore binaria é "perfecta" se ten cheos todos os seus niveis. Isto é, unha árbore binaria perfecta terá 2^i nodos no nivel i (para cada nivel i da árbore). Defina a función ***es_perfecta***: ***bitree -> bool*** que indique se unha árbore é ou non é perfecta.

Solución
más adelante →

Solución del examen de PD de Diciembre 2010

Ejercicio 1: deducir los mensajes de la compilación en el top-level

Una buena estrategia es diferenciar claramente los dos tipos de mensajes que nos puede dar el top-level acerca de una expresión válida. Son:

```
val <nombre> : <tipo_expresion> = <valor_expresion_evaluada>
- : <tipo_expresion> = <valor_expresion_evaluada>
```

¿Cómo sé cuando se muestran unos u otros? Bien, la respuesta es sencilla: después de una definición con let siempre nos encontraremos el primer caso, y el segundo se corresponderá con la evaluación de resultados (y en definitiva solo se diferencian en la palabra “val” y el nombre asignado antes de los dos puntos). Por ejemplo:

```
# let five _ = 5;;
val five : 'a -> int = <fun>
```

Y por la contra:

```
# mx3 1, mx3 1 2, mx3 1 2 3;;
- : (int -> int -> int) * (int -> int) * int = (<fun>, <fun>, 3)
```

En el primer caso estamos tomando el valor 5 y asignandoselo a una función “five” que reciba cualquier cosa “_” (comodín). En la segunda instrucción vemos tres expresiones separadas por comas, es decir, tres miembros de una tupla. En esta, el primer miembro es:

mx3 1

Como se ve en la definición previa, mx3 es una función de tipo 'a -> a' -> 'a -> 'a, que devuelve el valor máximo de 3 valores dados. Como es una función de “Curry”, se puede aplicar a menos parámetros de los esperados, y

se produce como resultado otra función que tendrá un parámetro menos (y que de usarse, tomará como primero el parámetro que le hayamos impuesto). Así, mx3 1 es una función que devolverá el máximo de dos valores y 1; por lo que su tipo es:

int -> int -> int

donde el último "int" se refiere ya a la imagen de la función. El segundo miembro de la tupla se comporta de forma similar. El tercer miembro de la tupla es la función mx3 aplicada a los enteros 1, 2 y 3. Obviamente su máximo es 3. El tipo de la tupla entonces se corresponde con el que ya he puesto arriba:

- : (int -> int -> int) * (int -> int) * int = (<fun>, <fun>, 3),

dónde se ven claramente los miembros de la tupla separados por el símbolo asterisco.

Es obligado hacer una pequeña aclaración: una definición local se evalúa como una expresión, por ejemplo:

let x = 2 in x;;

Produce como resultado:

- : int = 2

porque, realmente, dado el carácter local de la definición, el mensaje principal de la sentencia es lo que venga a continuación de "in", por lo que en este caso se está simplemente invocando al valor representado por la letra "x", que puntualmente (localmente) es el número entero 2.

Por supuesto, una sentencia puede conducir a una excepción, que se denotan de la siguiente manera (no aparece ninguna en este examen):

Exception: <Constructor_excepcion> "<mensaje_o_nombre>".

Por ejemplo:

Exception: Invalid_argument "mi_funcion".

Aunque no todos los constructores de excepciones aceptan un mensaje, como pasa con “Division_by_zero”:

Dado que este ejercicio se puede corregir simplemente usando el top-level, pasaré al siguiente.

Ejercicio 2: reescribir dos funciones sin usar definiciones locales ni sentencias “if...then...else”

Lo más importante para reescribir una función es entender bien el propósito de la original para poder hacer correctamente nuestra versión. Vamos con la primera:

```
# let f x = let (x,y) = x in x;;
```

En esta expresión, lo primero que encontramos es que asociamos a la letra f una función que opera sobre un parámetro x. ¿Qué ocurre en el “let...in”?

Comenzamos por la parte “in”. En ella simplemente se devuelve el valor de la x. En la definición local sin embargo, vemos que encajamos en un par (x,y) el valor de la x.

Por lo tanto, la x ha de ser un par. Si de el par (x,y) devolvemos la x, estamos simplemente definiendo una función f que dado un par, devuelve su primera componente. Podemos reescribirla así:

```
# let f (a, _) = a;;
```

Por supuesto los tipos coinciden y se expresan con valores 'a y 'b, pues no sabemos por qué tipo de elementos estará construido el par:

```
val f : 'a * 'b -> 'a = <fun>
```

De esta forma evitamos el uso de las definiciones locales.

En el segundo caso nos encontramos con esta función:

```
# let n x g = if g x then true else false;;
```

Lo gracioso es que para eliminar la sentencia condicional "if...then...else" de esta expresión habrá que hacer precisamente eso. La explicación es muy sencilla:

Imaginemos una expresión como la siguiente:

```
# 2 = 5;;
- : bool = false
```

Es una comparación. El resultado de una comparación obviamente solo puede ser verdadero o falso. 2 no es 5, por lo que aquí el resultado sería falso. En la instrucción del examen, estamos creando una función n que recibe un parámetro x y otro parámetro g. Por lo que podemos ver, g es otra función ya que luego se aplica a x tras el if. Según esto, g x ha de producir un resultado booleano ya de por sí, por lo que utilizar el if para establecer que es verdadero si al evaluarlo lo fue, o que será falso si al evaluarlo lo fué, es caer en una terrible redundancia. Es mucho más simple así, comparando si ha conducido a un resultado verdadero:

```
# let n x g = (g x) = true;;
```

Y por supuesto su tipo es:

```
val n : 'a -> ('a -> bool) -> bool = <fun>
```

Esto es, dado un valor 'a y una función que se aplica a 'a y retorna valores booleanos, nuestra función "n" devolverá tambien su resultado booleano.

Ejercicio 3: Indicar el tipo de dos funciones

```
# let rec sorted = function
[] | [] -> true
| x::y::z -> x <= y && sorted (y::z);;
```

De nuevo tenemos que entender bien qué nos dicen las funciones. La función "sorted" trabaja aparentemente con una lista y devuelve aparentemente valores booleanos (esto lo podemos apreciar a simple vista en la segunda linea del código). Por lo tanto, parece lógico que su tipo sea:

```
val sorted : 'a list -> bool = <fun>
```

El problema viene en el tipo de la lista. Dado que las operaciones que vemos en la ultima linea (es decir, la operación comparativa "menor o igual que": \leq) se pueden aplicar a varios tipos de datos (por ejemplo, a char o a int) tendremos que decantarnos porque sea una 'a list, ya que a priori no podremos concretar más.

```
# let rec merge l1 l2 = match (l1,l2) with
  (l, []) | ([], l) -> l
  | (h1::t1, h2::t2) -> if h1 <= h2 then h1 :: merge t1 l2
    else h2 :: merge l1 t2;;
```

Ahora tenemos ante nosotros una función que parte de dos listas l1 y l2. En principio, la unica operación que encontramos es la igualdad en la sentencia condicional, por lo que de nuevo no podemos prever su tipo concreto, pero si sabemos que ambas deben almacenar elementos del mismo tipo para que la comparación pueda ser viable. Es decir, las dos listas que son parámetros de la función serán de tipo 'a list. falta por ver el resultado. En la primera regla vemos que se devuelve una de ellas, por lo tanto su tipo de salida ha de ser el mismo que el de entrada:

```
val merge : 'a list -> 'a list > 'a list = <fun>
```

La primera de las funciones es la que verifica si los elementos de una lista están ordenados. Trabaja de forma recursiva: una lista estará ordenada si es vacía, si solo tiene un elemento o si el primero es menor o igual que el segundo y el resto de la lista tambien está ordenada.

La segunda, mezcla dos listas ordenadas en una sola lista ordenada de la siguiente forma: Si una de las dos listas está vacía, el resultado será inequívocamente la otra, pues ya es en sí una lista ordenada. Si ambas contienen elementos, se compondrá una lista que comienza con el menor de ellos y con el resultado de mezclar el resto de esa y la otra completa.

Obviamente esta composición no tendrá lugar mientras no se hayan compuesto los últimos elementos.

¿Podemos redefinirlas de forma recursiva terminal? Por supuesto que podemos. Como siempre, la estrategia es delegar la recursividad en una función interna definida localmente donde coloquemos un parámetro extra en el que llevar el resultado temporalmente. Las dos soluciones que yo he encontrado son:

```
# let sorted l =
let rec sorted maximo li = match li with
  [] -> true
  | h::t -> if maximo <= h
    then sorted h t
    else false
  in sorted (List.hd l) l;;
```

Mi función sorted utiliza un parámetro "máximo" que va almacenando el máximo de una lista y otro "li" que es la lista que falta por comprobar. Comienza tomando como máximo el valor de la cabeza de la lista "l" y la propia lista "l", que es la que yo quiero comprobar. En cada paso, se comprobará si el máximo es menor o igual que el primer elemento de la lista. Si lo es, comprobaremos la cola tomando como máximo el elemento h.

```
# let merge l1 l2 =
let rec merge l1 l2 lf = match (l1, l2) with
  (l, []) | ([], l) -> lf @ l
  | (h1::t1, h2::t2) -> if h1 <= h2
    then merge t1 l2 (lf@[h1])
    else merge l1 t2 (lf@[h2])
  in merge l1 l2 [];;
```

La función merge me ha dado algo más de trabajo. Por supuesto, utilizaré el parámetro "lf" para ir componiendo una "lista final" de la siguiente forma. En el caso de que una de las dos listas sea vacía, compondré lo que haya en la lista final con lo que quede de la otra.

En el caso de que ambas tengan elementos, el menor de las cabezas debe ser compuesto con la lista final y la función merge invocada para lo que reste de esa lista, la otra completa y llevando ya la lista resultante como parámetro "lf", sin tener que dejar pendiente ninguna operación. Obviamente mi "lista final" debe estar vacía al inicio.

Ejercicio 4: Definir la función “es_perfecto” para un árbol binario

Dado un tipo árbol binario “bitree” de la forma:

```
# type bitree = Empty | Node of bitree * bitree;;
```

Un arbol binario será perfecto si tiene 0 o 2 hijos, pero no si solo tiene 1. Tenemos que aprovecharnos de la naturaleza recursiva de un árbol para que juegue en nuestro favor. Mi solución ha sido la siguiente:

En primer lugar, desecharmos los casos base verdaderos: un árbol vacío (Empty) o el nodo que no tenga hijos, es decir Node (Empty,Empty). En ambas situaciones, un arbol es perfecto.

Desechados estos primeros casos, nos centramos en el arbol que tiene solo uno de sus hijos, que nos conduce a un resultado “false”. Por ultimo, para el arbol que tenga dos subárboles hijos (i y d) habrá que evaluar si son árboles perfectos ambos hijos.

```
# let rec es_perfecto = function
  Empty
  | Node (Empty, Empty) -> true
  | Node (Empty, _)
  | Node (_, Empty) -> false
  | Node (i, d) -> es_perfecto i && es_perfecto d;;
```



