Ocaml.pdf

Gonzoo

Paradigmas de Programación

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

1) PRIMERAS INSTRUCCIONES EN OCAML

```
# 1 + 2 * 3;;
- : int = 7
```

Podemos ver que Ocaml indica que el valor que devuelve es de tipo int.
Ocaml interpreta esta expresión de la siguiente forma:
(1 + (2 * 3)) -> (1 + 6) -> 7

Tenemos las siguientes operaciones en Ocaml:

a + b -> suma
a - b -> resta
a * b -> multiplicación
a / b -> divide a y b, devolviendo la parte entera
a mod b -> divide a y b, devolviendo el resto

(*b = 0 Ocaml imprime Exception: Division_by_zero*)
(*cuando b <> 0, x < 0 el resultado es negativo*)
(*cuando b <> 0, x = 0 el resultado es 0*)

Podemos usar también comparaciones booleanas:

a = b -> a es igual a b
a < b -> a menor que b
a > b -> a mayor que b
a >= b -> a mayor o igual que b
a <= b -> a menor o igual que b
a <> b -> a distinto de b

Si intentamos hacer operaciones entre valores de distinto tipo puede resultar en un error, por ejemplo:

```
# 1 + true;;
Error: This expression has type bool but an expression was expected of type
         int
```

El operador + recibe dos valores de tipo int y realiza la suma de los dos valores enteros, al recibir un valor bool da como resultado un error.

Otro tipo que tiene Ocaml es char, y si juntamos varios obtenemos un String

Mediante el operador ^ podemos concatenar strings

```
# let c = 'c' ;;
val c : char = 'c'
```

Construcción if...then...else:

```
# if 100 > 99 then 1 else 0;;
- : int = 1
```

La expresión entre if y then debe tener tipo bool para evaluar que sea true o false; en caso de que sea true, hará lo que ponga después de then, si es false hará lo que está después de else.

2) NOMBRES Y FUNCIONES

Podemos definir expresiones si las vamos a usar más veces

```
let x = 200;;

x * x * x;;
- : int = 8000000
```

Si queremos escribirlo todo en una sola expresión podemos usar let ... = ... in

```
# let a = 200 in a * a * a;;
- : int = 8000000
```

Esta definición de valor podría considerarse volátil, porque una vez se calcule el resultado de la operación, dicho valor no se guarda en memoria.

Podemos también definir una función que realice un cálculo

```
# let cube x = x * x * x;;
val cube : int -> int = <fun>
```

Ocaml nos dice que es una función int -> int, esto se debe a que recibe un int y devuelve un int (el número elevado al cubo).

```
# let negativo n = if n < 0 then true else false;;
val negativo : int -> bool = <fun>
(*Deberíamos añadir paréntesis para comprobar que la función funciona para números negativos*)
```

Esta función es equivalente a:

```
let negativo = x < 0;;

(*CUESTIONES*)

# let mayus c = if (int_of_char c >= 65) && (int_of_char c <= 90) then true else false;;
val mayus : char -> bool = <fun>
(*Función que comprueba si el caracter introducido es una letra mayúscula*)

# let addtoten a b = a + b = 10;;
val addtoten : int -> int -> bool = <fun>
(*Comprueba si la suma de los dos números es igual a 10 y devuelve un bool*)
```

A continuación usaremos rec antes del nombre de la función para indicar que es recursiva y puede llamarse a sí misma en su interior.

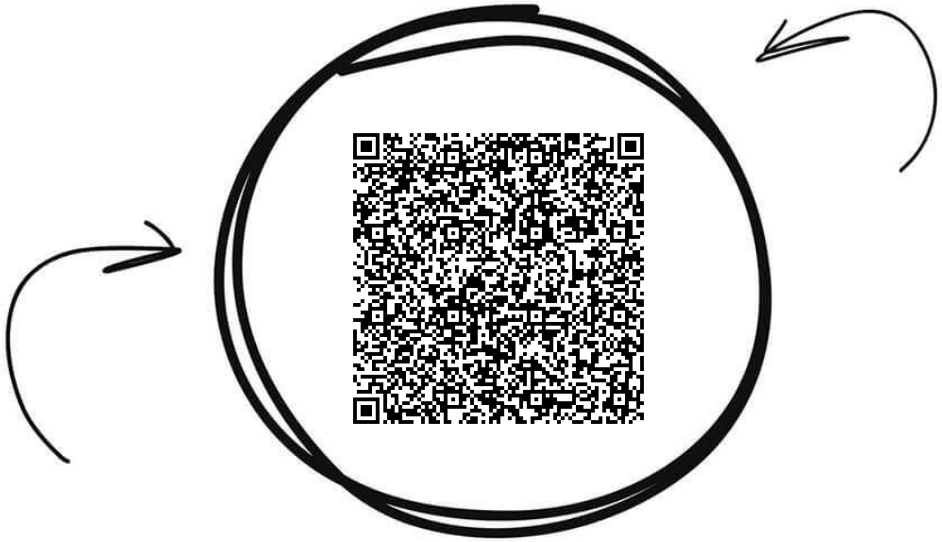
```
let rec factorial a =
 if a < 0 then raise (Invalid_argument "factorial") else
  if (a = 0) || (a = 1) then 1 else a * fact (a-1);;
val factorial : int -> int = <fun>

let rec suma n =
 if n < 0 then raise (Invalid_argument "suma")
 else if n = 0 then 0 else
  if n = 1 then 1 else n + suma (n-1);;
```

PEKIS®
for Foodies

SÓLO UN APUNTE MÁS: CÓMETE UN TACO.

RELLENOS PARA TACOS PEKIS® BRUTALES.

Paradigmas de Programación



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

```
val suma : int -> int = <fun>

let rec power x n =
 if (n = 0) && (x = 0) then raise (Failure "power") else
  if (n = 0) && (x <> 0) then 1 else
   if (n = 1)  then x else x * (power x (n-1));;
val power : int -> int -> int = <fun>

let isconsonant c =
 if ((c = "a ) || (c = 'e') || (c = 'i') || (c = 'o') || (c = 'u')) then false else
  if (int_of_char c >= 97) && (int_of_char c <= 122) then true else false;;
val isconsonant : char -> bool = <fun>
```

3) PATTERN MATCHING

En vez de usar la estructura if...then...else usaremos pattern matching debido a que es más fácil leerlo y permite expandir la función más fácilmente.

```
let rec fact a =
 if a = 1 then 1 else a * fact (a-1);;
```

Con pattern matching sería así:

```
let rec fact a = match a with
 1 -> 1
 | _ -> a * fact (a-1);;

let isvowel c = match c with
 "a  | 'e' | 'i' | 'o' | 'u' -> true
 | _ -> false;;

val isvowel : char -> bool = <fun>

(*Algoritmo de Euclides para obtener el mayor divisor común*)
let rec gcd a b = match b with
 0 -> a
 | _ ->  gcd b (a mod b);;

val gcd : int -> int -> int = <fun>

let not arg = match arg with
 true -> false
 | false -> true;;
val not : bool -> bool = <fun>

let rec suma n = match n with
 0 -> 0
 | _ -> n + suma (n-1);;

let rec power x n = match n with
 0 -> 1
 | 1 -> x
```

```
| _ -> x * power x (n-1);;

let mayus c = match c with
 "a ..'Z' -> true
 | _ -> false;;

let minus c = match c with
 "a ..'z' -> true
 | _ -> false;;
```

4) LISTAS

Todos los elementos de una lista deben tener el mismo tipo.

```
let list = [1;2;3];;
val list : int list = [1;2;3]
```

Llamamos head al primer elemento de una lista, y al resto de la lista sin la cabeza la llamamos tail
Mediante las operaciones List.hd y List.tl podemos acceder a la cabeza y a la cola respectivamente.
Aunque la lista tenga solamente un elemento, por ejemplo [5], su head sería 5 y su tail sería [].

Para añadir un nuevo elemento a la cabeza de una lista usaremos el operador ::

```
0 :: list;;
val list : int list = [0;1;2;3]
```

Si queremos concatenar dos listas usaremos @:

```
[1;2] @ [3;4;5];;
[1;2;3;4;5]

let isemptylist l = match l with
 [] -> true
 | _ -> false;;
val isemptylist : 'a list -> bool = <fun>


let rec length l =
 match l with
  [] -> 0
  | h::t -> 1 + length t;;
val length : 'alist -> int = <fun>
```

Esta funcion tiene tipo 'a list -> int porque no sabemos qué tipo tendrá la lista, y devuelve un entero que es su longitud. También podemos sustituir h por _ porque no lo usamos.

```
let rec sum n =
 match n with
  [] -> 0
  | h::t -> h + sum t;;
val sum : int list -> int = <fun>
```

Estas operaciones no son recursivas terminales, por lo tanto pueden provocar Stack Overflow al llegar a un determinado número de iteraciones; podemos hacerlas recurivas terminales.

```
let rec length_inner l n =
 match l with
  [] -> n
  | h::t -> length_inner t (n+1)
let length l = length_inner l 0;;
```

(*n es el acumulador de la función, la longitud de la lista; luego definimos una función length a la cual se le pasamos una lista y ponemos el acumulador a 0.*)

```
let rec impares l =
 match l with
 [] -> []
 | [a] -> [a]
 | a::_::t -> a::impares t;;
val impares : 'a list -> 'a list = <fun>

let rec pares l =
 match l with
  [] -> []
  | [_] -> []
  | _::a::t -> a::pares t;;
val pares : 'a list -> 'a list = <fun>

let rec append l1 l2 =
 match l1 with
  [] -> l2
  | h::t -> h::append t l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

let rec rev l =
 match l with
 [] -> []
 | h::t -> rev t @ [h];;
val rev : 'a list -> 'a list = <fun>
```

Una implementación de la función de forma recursiva terminal sería:

```
let rec rev_inner a l =
 match l with
  [] -> a
  | h::t -> rev_inner (h::a) t;;
let rev l = rev_inner [] l;;

let palindrome l =
 l = rev l;;
val palindrome : 'a list -> bool = <fun>

let rec take n l =
 if n = 0 then [] else
  match l with
   h::t -> h::take (n - 1) t;;
val take : int -> 'a list -> 'a list = <fun>
```

(*Devuelve una lista con los n primeros elementos de la lista*)

```
let rec drop n l =
 if n = 0 then l else
  match l with
   h::t -> drop (n-1) t;;
val drop : int -> 'a list -> 'a list = <fun>
```

(*Devuelve una lista sin los n primeros elementos de la lista*)

```
let rec count_true l =
 match l with
  [] ->  0
  | true::t -> 1 + count_true t
  | false::t -> count_true t;;
val count_true : bool list -> int = <fun>

let rec drop_last l =
 match l with
  [] -> []
  | [_] -> []
  | h::t -> h::drop_last t;;
val drop_last : 'a list -> 'a list = <fun>

let rec member n l =
 match l with
  [] -> false
  | h::t -> h = n || member n t;;
val member : 'a  -> 'a list -> bool = <fun>

let rec make_set l =
 match l with
 [] -> []
 | h::t -> if member h t then make_set t else h::make_set t;;
val make_set : 'a list -> 'a list = <fun>
```

(*Si el elemento situado en la cabeza está en la cola de la lista entonces no lo pone y pasa al siguiente, si no está lo añade a la lista resultante*)

5) MÉTODOS DE ORDENACIÓN

```
let rec insert x l =
 match l with
  [] -> [x]
  | h::t -> if x <= h then x::h::t
  else h::insert x t;;
val insert : 'a  -> 'a list -> 'a list = <fun>

let rec sort l =
 match l with
  [] -> []
  | h::t -> insert h (sort t);; (*insertamos la cabeza en una lista ordenada ya que insert lo añade en su correspondiente posición*)
```

```
val sort : 'a list -> 'a list = <fun>

let rec merge x y =
 match x, y with
 [], l -> l (*Casos de listas vacías*)
 | l, [] -> l
 | hx :: tx, hy :: ty ->
  if hx < hy
   then hx :: merge tx (hy::ty) (*concatenamos hx con merge de lo que sobra de x y la lista y*)
   else hy :: merge (hx::tx) ty;; (*concatenamos hy con la lista y lo que sobra de y*)
val merge : 'a list -> 'a list -> 'a list = <fun>

(*CUESTIONES*)
5.1)
let rec msort l =
 match l with
  [] -> []
  | [x] -> [x] (*si la lista es vacía o tiene un elemento ya estaría*)
  | _ ->
   let mitad = (length l)/2 in
    let left = take mitad l in
     let right = drop mitad l in
      merge (msort left) (msort right);;
val msort : 'a list -> 'a list = <fun>

5.3)
(*FUNCION QUE ORDENA LA LISTA AL REVÉS (DE MAYOR A MENOR), EN REALIDAD
SOLO HAY QUE CAMBIAR EL <= POR >=*)

let rec reverseinsert x l =
 match l with
  [] -> [x]
  | h::t ->
   if x > h then x::h::t
   else h:: reverseinsert x t;;
val reverseinsert : 'a  -> 'a list -> 'a list = <fun>

let rec inversesort l =
 match l with
  [] -> []
  | [x] -> [x]
  | h::t -> reverseinsert h (inversesort t);;
val inversesort : 'a list -> 'a list = <fun>

5.4)
let is_sorted l =
 if l = sort l then true else false;;
val is_sorted : 'a list -> bool = <fun>

(*Las listas se comparan con los operadores <,>,>=,<= basándose en el primer elemento de ellas, si
ambas tienen el mismo primer elemento, se pasa al siguiente*)
```

5.6)

```
let rec sort l =
 let rec insert x s =
  match s with
   [] -> []
   | h::t ->
    if x <= h
     then x::h::t
     else h::insert x t
  in
   match l with
    [] -> []
    | h::t -> insert h (sort t)
val sort : 'a list -> 'a list = <fun>
```

6) FUNCIONES SOBRE FUNCIONES

```
let rec double l =
 match l with
  [] -> []
  | h::t -> h*2::double t;;
val double : int list -> int list = <fun>

let rec par l =
 match l with
  [] -> []
  | h::t -> (h mod 2 = 0)::par t;;
val par : int list -> bool list = <fun>

let rec map f l =  (*f es la función que queremos que se le aplique a la lista l*)
 match l with
  [] -> []
  | h::t -> f h :: map f t;; (*aplicamos la función a la cabeza de la lista y luego map al resto*)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

(*alfa por beta en alfa list (lista que le pasamos) en beta list (lista que da como resultado)*)


let halve x = x/2;;
(*haremos que a la lista [10;20;30] se le aplique la función halve*)

map halve [10;20;30];;

int list = [5; 10; 15]

let is_even x =
 x mod 2 = 0;;

let evens l =
 map is_even l;;

(*También podemos usar funciones anónimas si sólo vamos a usarlas una única vez*)
```

```
let evens l =
 map (fun x -> x mod 2 = 0);; (*la funcion que le pasamos es la que utiliza map*)

(*otro ejemplo con sort y msort*)

let greater a b =
 a >= b;;
val greater : 'a  -> 'a  -> bool = <fun>

let rec merge cmp x y =
 match x , y with
  [] , l -> l
  | l , [] -> l
  | hx :: tx, hy :: ty ->
   if cmp hx hy
    then hx :: merge cmp tx (hy :: ty)
    else hy :: merge cmp (hx :: tx) ty;;
val merge : ('a  -> 'a  -> bool) -> 'a list -> 'a list -> 'a list = <fun>

let rec msort cmp l =
 match l with
  [] -> []
  | [x] -> [x]
  | _ ->
  let media = (length l/2) in
   let left = take media l in
    let right = drop media l in
     merge cmp (msort cmp left)(msort cmp right);;
val msort : ('a  -> 'a  -> bool) -> 'a list -> 'a list = <fun>

(*le pasamos como función el método de ordenación y nos lo ordena sin necesidad de cambiar la
implementación de msort y sort*)

msort (<=) [5;4;6;2;1];;
[1;2;4;5;6]
msort (>=) [5;4;6;2;1];;
[6;5;4;2;1]

(*CUESTIONES*)
6.1)
let rec calm l =
 match l with
  [] -> []
  | '!'::t -> '.' :: calm t
  | h::t -> h :: calm t;;
val calm : char list -> char list = <fun>

(*USANDO MAP*)

let calm_char x =
 match x with
```

```
  '!' -> '.'
  | _ -> x;;
val calm_char : char -> char = <fun>

let calm l =
 map calm_char l;;
val calm : char list -> char list = <fun>

6.2)
let clip x =
 if x > 10 then 10
  else if x < 1 then 1
   else x;;
val clip : int -> int = <fun>

let cliplist l =
 map clip l;;
val cliplist : int list -> int list = <fun>

6.3)
(*ahora empleando una función anónima en vez de clip*)
let cliplist1 l =
 map (fun x -> if x > 10 then 10 else if x < 1 then 1 else x) l;;

6.4)
let rec apply f times x =
 if times = 0
  then x
 else f (apply f (times - 1) x);;
val apply : ('a  -> 'a ) -> int -> 'a  -> 'a  = <fun>

6.5)
let rec insert f x l =
 match l with
  [] -> [x]
  | h::t -> if f x h
   then x::h::t
   else h::insert f x t;;
val insert : ('a  -> 'a  -> bool) -> 'a  -> 'a list -> 'a list = <fun>

let rec sort f l =
 match l with
  [] -> []
  | h::t -> insert f h (sort f t);;
val sort : ('a  -> 'a  -> bool) -> 'a list -> 'a list = <fun>

6.6)
let rec filter f l =
 match l with
  [] -> []
  | h::t ->
   if f h
```

```
    then h::filter f t
    else filter f t;;
val filter : ('a  -> bool) -> 'a list -> 'a list = <fun>

int list = [7; 14; 3; 5; 9; 11; 2; 0]
filter (fun x -> x mod 2 = 0) lista;;
- : int list = [14; 2; 0]
```

6.7)

```
let rec for_all f l =
 match l with
  [] -> true
  | h::t -> f h && for_all f t;;
val for_all : ('a  -> bool) -> 'a list -> bool = <fun>
ej: for_all (fun x -> x >= 0) lista;;
```

6.8)

```
let rec mapl f l =
 match l with
  [] -> []
  | h::t -> map f h :: mapl f t;;
val mapl : ('a  -> 'b ) -> 'a  list list -> 'b  list list
```

7) EXCEPCIONES

Podemos definir nuestras propias excepciones mediante exception, las excepciones DEBEN empezar por letra mayúscula. EL constructor of puede usarse para introducir el tipo de información que va con una excepción. Cuando estén definidas podemos usar raise para usarlas en nuestras funciones.

```
exception Problem;;
exception NotPrime of int;;

let rec take n l =
 match l with
  [] ->
   if n = 0
    then []
    else raise (Invalid_argument "take")
  | h::t ->
   if n < 0 then raise (Invalid_argument "take") else
    if n = 0 then [] else h:: take (n-1) t;;

let rec drop n l =
 match l with
  [] ->
   if n = 0
    then []
    else raise (Invalid_argument "drop")
  | h::t ->
   if n < 0 then raise (Invalid_argument "drop") else
    if n = 0 then l else drop (n-1) t;;
```

Podemos usar un manejador de excepciones (exception handler) con try...with

```
let safe_divide x y =
 try x / y with
  Division_by_zero -> 0;;   (*En el caso de que x / 0 cambiará la excepción de Division_by_zero por el valor 0*)

let rec last l =
 match l with
  [] -> raise Not_found
  | [x] -> [x]
  | h::t -> last t;;
val last : 'a list -> 'a list = <fun>

(*CUESTIONES*)

7.1)
(*Funciones que calculan el mayor y el menor elemento de una lista, en smallest_inner comprueba que sea positivo también*)

let rec highest_inner current found l =
 match l with
  [] ->
   if found then current else raise Not_found
  | h::t ->
   if h > current
    then highest_inner h true t
    else highest_inner current found t;;
val highest_inner : 'a  -> bool -> 'a list -> 'a  = <fun>

let highest l =
 highest_inner min_int false l;;
val highest : int list -> int = <fun>


let rec smallest_inner current found l =
 match l with
  [] ->
   if found then current else raise Not_found
  | h::t ->
   if h > 0 && h < current
    then smallest_inner h true t
    else smallest_inner current found t;;

let smallest l =
 smallest_inner max_int false l;;

7.2)
let rec smallest_or_zero l =
 try smallest l with Not_found -> 0;;
```

7.3)
exception Complejo;;

```
let rec sqrt_inner x n =
 if x * x > n
  then x - 1
 else sqrt_inner (x+1) n;;
val sqrt_inner : int -> int -> int = <fun>

let sqrt n =
 if n < 0
  then raise Complejo
 else sqrt_inner 1 n;;
val sqrt : int -> int = <fun>
```

(*comprueba que n sea mayor que x^2 empezando x en 1, y aumentando su valor. Si $x^2 > n$ entonces la respuesta es n-1. sqrt 26 = 5*)

7.4)

```
try safe_sqrt n =
 try sqrt n with Complejo -> 0;;
```

8) PARES

No tienen por que tener el mismo tipo
p : int x int

```
let p = (1,4);;

let fst (x,_) = x;;
let snd (_,y) = y;;

val fst : 'a  * 'b  -> 'a
val snd : 'a  * 'b  -> 'a

let censo = [(1,4);(2,2);(3,2);(4,3);(5,1);(6,2)];;
val censo : (int x int) list
```

(*Los paréntesis indican el tipo*)

```
let rec lookup x l =
 match l with
  [] -> raise Not_found
  | (k,v)::t ->
   if k = x then v else lookup x t;;
val lookup : 'a  -> ('a  * 'b ) list -> 'b  = <fun>
```

(*busca un elemento en la (int x int) list mediante una key y devuelve la segunda componente, de tipo 'b *)

```
let rec add k v d =       (*key, value, dictionary*)
 match d with
  [] -> [(k,v)]   (*si el dictionary está vacío lo añade*)
  | (k',v')::t ->
   if k = k' then (k,v) :: t          (*misma key, la reemplaza por el nuevo valor*)
   else (k',v') :: add k v t;;        (*sigue comprobando mediante recursividad*)
```

```
val add : 'a  -> 'b  -> ('a  * 'b ) list -> ('a  * 'b ) list = <fun>

let rec remove k d =
 match d with
  [] -> []
  | (k', v')::t ->
   if k = k' then t          (*elimina la clave*)
   else (k',v'):: remove k t;;      (*sigue buscando la clave a eliminar recursivamente*)
val remove : 'a  -> ('a  * 'b ) list -> ('a  * 'b ) list = <fun>

let rec key_exists k d =
 match d with
  [] -> false
  | (k',v')::t ->
   if k = k' then true
   else key_exists k t;;
val key_exists : 'a  -> ('a  * 'b ) list -> bool = <fun>

(*CUESTIONES*)
8.2)
let rec replace k v d =
 match d with
  [] -> raise Not_found
  | (k',v')::t ->
   if k = k' then (k,v)::t
   else (k',v')::replace k v t;;
val replace : 'a  -> 'b  -> ('a  * 'b ) list -> ('a  * 'b ) list = <fun>


8.3)
let rec length l =
 match l with
  [] -> 0
  | h::t -> 1 + length t;;

let rec dictionary l1 l2 =
 match l1 , l2 with
  [], [] -> []
  | l1, [] -> raise (Invalid_argument "dictionary")
  | [], l2 -> raise (Invalid_argument "dictionary")
  | h::t , h'::t' ->
   if length l1 <> length l2 then raise (Invalid_argument "dictionary")
   else (h,h')::dictionary t t';;
val dictionary : 'a list -> 'b  list -> ('a  * 'b ) list = <fun>

8.4)
let rec split d =
 match d with
  [] -> ([],[])
  | (k,v)::t ->
   match split t with
    (ks,vs) -> (k::ks, v::vs);;
```

```
val split : ('a  * 'b ) list -> ('a  list * 'b  list) = <fun>
```

8.5)

```
let rec dictionary_of_pairs_inner keys_seen l =
 match l with
  [] -> []
  | (k,v)::t ->
   if member k keys_seen
    then dictionary_of_pairs_inner keys_seen t
    else (k,v):: dictionary_of_pairs_inner (k::keys_seen) t
```

(*si la lista está vacía la devuelve; si tiene contenido, comprueba si la clave está en la lista de claves vistas, si está eso quiere decir que no se añade a la lista resultado. Si no está la clave se añade el par a la lista y se actualiza la lista de keys_seen con la nueva key añadida.*)

```
let dictionary_of_pairs l =
 dictionary_of_pairs_inner [] l;;

val dictionary_of_pairs_inner : 'a  list -> ('a  * 'b ) list -> ('a  * 'b ) list = <fun>
val dictionary_of_pairs : ('a  * 'b ) list -> ('a  * 'b ) list = <fun>
```

8.6)

```
let rec union a b =
 match a with
  [] -> b
  | (k,v)::t -> add k v (union t b);;
val union : ('a  * 'b ) list -> ('a  * 'b ) list -> ('a  * 'b ) list = <fun>
```

9) MÁS FUNCIONES

El tipo int -> int -> int puede escribirse como int -> (int -> int) ; esto nos sirve por ejemplo para pasarle simplemente un elemento a funciones de este tipo
Por ejemplo: add es una función int -> int -> int pero podemos hacer esto:

```
let f = add 6;;
val f : int -> int = <fun>
```

map (add 6) [10;20;30];; (*el primer argumento que recibe map es de tipo int -> int, por lo tanto es apropiado. Podemos usar también funciones anónimas*)

Recordamos la función mapl del tema 6

```
let rec mapl f l =
 match l with
  [] -> []
  | h::t -> map f h :: mapl f t;;
(*le aplica la función f a una 'a  list list, ej: mapl (fun x -> x*x) [[3;4;5];[6;7;8]]*)

(*Con aplicación parcial*)
let mapl f = map (map f);;
val mapl : 'a  list list -> 'b  list list = <fun>
```

let add = fun x -> fun y -> x + y;; (*equivale a*) let add x y = x + y;;
(*de esta forma podemos ver que nuestras funciones está compuestas por funciones de un solo argumento*)

(*CUESTIONES*)
9.1)
La función g a b c tiene tipo 'a ->'b ->'c ->'d ; que puede ser escrita así: 'a ->('b ->('c ->'d))
Recibe un argumento de tipo 'a y devuelve una función ('b ->('c ->'d)), el cual recibe un argumento 'b y devuelve una función ('c ->'d), la cual cuando recibe un argumento de tipo 'c devuelve algo de tipo 'd .
Debido a esto podemos hacer esto:
let g a b c = ... o let g = fun a = fun b = fun c -> ...

9.2)
```
let rec member x l =
 match l with
  [] -> false
  | h::t ->
  if x = h then true
  else member x t;;
val member : 'a  -> 'a list -> bool = <fun>
```

'a -> ('a list -> bool)

```
let member_all x ls =
 let booleans = map (member x) ls in (*map devuelve una bool list viendo si está el elemento x en todas las listas de listas*)
  not (member false booleans);; (*si está false en la bool list devuelve un true, pero con el not del principio lo niega*)
```

por ejemplo: member_all 1 [[1;2;3];[1;5;6];[1;8;9]]
booleans = [true;true;true]
ahora comprueba si false está en booleans, como no está devolverá false, pero nosotros queremos lo opuesto, por lo tanto lo negamos
member_all 1 [[1;2;3];[1;5;6];[1;8;9]]
- : bool = true

9.3)
No podemos usar map ((/) 2) [10;20;30] porque / es la función que divide 2 por el numero dado, no al revés como la empleamos aquí

```
let div x y = y / x;;
```

9.4)
```
let mapll f = map (map (map f));;
val mapll : 'a  list list list -> 'b  list list list
```

9.5) (*usamos la función take para truncar los n primeros elementos de la lista*)
```
let rec take n l =
 match l with
  [] ->
   if n = 0
```

```
   then []
   else raise (Invalid_argument "take")
 | h::t ->
   if n < 0 then raise (Invalid_argument "take") else
    if n = 0 then [] else h:: take (n-1) t;;
```

(*creamos una función que trunque los n primeros elementos mediante take modificando la excepción*)

```
let truncate_l n l =
 try take n l with Invalid_argument _ -> l;;
```

(*aplicamos la función que creamos y se la pasamos a map para que la aplique a todas las listas de ls*)

```
let truncate n ls =
 map (truncate_l n) ls;;
```

```
val truncate_l : int -> 'a list -> 'a list = <fun>
val truncate : int -> 'a list list -> 'a list list = <fun>
```

9.6)

```
let first_l x l =
 match l with
  [] -> x
  | h::t -> h;;

let first x ls =
 map (first_l x) ls;;
```

```
val first_l : 'a  -> 'a list -> 'a  = <fun>
val first : 'a  -> 'a  list list -> 'a  list = <fun>
```

(*función que devuelve una int list con los primeros elementos de una int list list*)

10) NUEVOS TIPOS DE DATOS

Podemos definir nuevos tipos de datos mediante la palabra reservada type, los constructores deben empezar por letra mayúscula, son los posibles valores que puede tomar color.

```
type color = Rojo | Verde | Amarillo | Azul;;
```

Podemos hacer listas, pares y argumentos con estos nuevos tipos de dato

```
let ceras = [Rojo ; Verde ; Amarillo ; Azul];;
let colpar = ('R', Rojo);;
```

Añadiremos un nuevo color que será el RGB, que tomará valores de 0 a 255 en una tupla de 3 elementos

```
let color = Rojo | Verde | Amarillo | Azul |  RGB of int * int * int;;
let cols = [Rojo ; Verde ; Amarillo ; RGB (255,0,250)];;
```

```
let components c =
 match c with
```

```
  Rojo -> (255,0,0)
  | Amarillo -> (255,255,0)
  | Verde -> (0,255,0)
  | Azul -> (0,0,255)
  | RGB (r,g,b) -> (r,g,b);;


type 'a option = None | Some of 'a;;

let nothing = None;;
let number = Some 50;;
val number : int option = Some 50

let numbers = [Some 12; None ; None ;Some 2];;
val numbers : int option list = [Some 12; None ; None ;Some 2]

let word = Some ['h';'o';'l';'a'];;
val word : char list option = ['h';'o';'l';'a']
```

Los nuevos tipos son útiles para definir excepciones, por ejemplo:

```
let rec lookup_opt x l =
 match l with
  [] -> None
  | (k,v)::t -> if x = k then Some v else lookup_opt x t;;

type 'a sequence = Nil | Cons of 'a * 'a sequence;;

[] -> Nil -> 'a sequence
[1] -> Cons (1,Nil) -> int sequence
['a';'x';'e'] -> Cons ('a',Cons ('x',Cons ('e', Nil))) -> char sequence
[Red, RGB (20,20,20)] -> Cons (Red, Cons (RGB(20,20,20),Nil)) -> color sequence


let rec length s =
 match s with
  Nil -> 0
  | Cons (_,t) -> 1 + length t;;
val length : 'a sequence -> int = <fun>

let rec append a b =
 match a with
  Nil -> b
  | Cons (h,t) -> Cons (h, append t b);;
val append : 'a sequence -> 'a sequence -> 'a sequence = <fun>

type expr =
 Num of int
 | Suma of expr * expr
 | Resta of expr * expr
 | Mult of expr * expr
 | Div of expr * expr;;
```

1+2*3 -> Suma (Num 1, Mult (Num 2, Num 3))

```
let rec evaluate e =
 match e with
  Suma (e,e') -> evaluate e + evaluate e'
  | Resta (e,e') -> evaluate e + evaluate e'
  | Mult (e,e') -> evaluate e * evaluate e'
  | Div (e,e') -> evaluate e / evaluate e';;
val evaluate : expr -> int = <fun>

(*CUESTIONES*)

10.1)
type rect =
 Square of int
 | Rect of int * int;;

let s = Square 7;;
let Rect (5,2);;

10.2)
let area r =
 match r with
  Square s -> c * c
  | Rect (b,h) -> b*h;;

10.3)
(*rotar rectángulo si su altura es igual o mayor que su base*)
let rotate r =
 match r with
  Square s -> Square s
  | Rect (b,h) -> if h >= b then Rect (h,b) else Rect (b,h);;

10.4)
(*dada una rect list devuelve otra lista ordenada por su base*)

let bases r =
 match r with
  Square s -> s
  | Rect (b,_) -> b
val bases : rect -> int = <fun>

let rect_compare r1 r2 =
 bases r1 < bases r2;;
val rect_compare : rect -> rect -> bool = <fun>

let pack rects =
 sort rect_compare (map rotate rects)
val pack : rect list -> rect list = <fun>
```

10.5)

```
let rec take n l =
 if n = 0 then Nil else
 match l with
  Nil -> raise (Invalid_argument "take")
  | Cons (h, t) -> Cons (h, take (n - 1) t);;
val take : int -> 'a sequence -> 'a sequence = <fun>

let rec drop n l =
 if n = 0 then l else
 match l with
  Nil -> raise (Invalid_argument "drop")
  | Cons (_, l) -> drop (n - 1) l;;
val drop : int -> 'a sequence -> 'a sequence = <fun>

let rec map f l =
 match l with
  Nil -> Nil
  | Cons (h, t) -> Cons (f h, map f t);;
val map : ('a -> 'b) -> 'a sequence -> 'b sequence
```

10.6)

```
type expr =
 Num of int
 | Suma of expr * expr
 | Resta of expr * expr
 | Mult of expr * expr
 | Div of expr * expr
 | Power of expr * expr;;

let rec power x n =
 if (n = 0) && (x = 0) then raise (Failure "power") else
  if (n = 0) && (x <> 0) then 1 else
   if (n = 1)  then x else x * (power x (n-1));;

let rec evaluate e =
 match e with
  Num x -> x
  | Suma (e,e') -> evaluate e + evaluate e'
  | Resta (e,e') -> evaluate e + evaluate e'
  | Mult (e,e') -> evaluate e * evaluate e'
  | Div (e,e') -> evaluate e / evaluate e'
  | Power (e,e') -> power (evaluate e) (evaluate e');;
```

10.7) (*trata con el problema de la división por 0 de evaluate*)

```
let evaluate_opt e =
 try Some (evaluate e) with Division_by_zero -> None;;
val evaluate_opt : expr -> int option = <fun>
```

11) ÁRBOLES

```
type 'a tree =
 Br of ('a * 'b) * 'a tree * 'a tree          (*el elemento situado en root y los 2 nodos hijo*)
 | Lf;;

let t1 = Branch (2, Branch (1,Lf,Lf), Branch (4,Lf,Lf));;

(*número de elementos en un árbol, las hojas no cuentan*)
let rec size tree =
 match tree with
  Br (_, l, r) -> 1 + size l + size r
  | Lf -> 0;;
val size : 'a tree -> int = <fun>

(*suma de los contenidos del árbol*)
let rec total tree =
 match tree with
  Br (x, l, r) -> x + total l + total r
  | Lf -> 0;;
val total : 'a tree -> int = <fun>

let max x y =
 if x >= y then x else y;;

(*función que calcula la altura máxima del árbol*)
let rec max_depth tree =
 match tree with
  Br (_,l,r) -> 1 + max (max_depth l)(max_depth r)
  | Lf -> 0;;

(*convierte el árbol en una lista poniendo la raiz entre los dos nodos hijo*)
let rec list_of_tree tree =
 match tree with
  Br (x,l,r) ->list_of_tree l @ [x] @ list_of_tree r
  | Lf -> [];;
val list_of_tree : 'a tree -> 'a list = <fun>

(*aplica una función f a un 'a tree devolviendo el 'a tree con la función aplicada sobre todos los
elementos*)
let rec tree_map f tree =
 match tree with
  Br (x,l,r) -> Br (f x, tree_map f l, tree_map f r)
  | Lf -> Lf;;
```

ÁRBOLES BINARIOS DE BÚSQUEDA (AVL)

```
 let t4 = Br((4,"four"), Br ((2,"two"), Br ((1, "one"), Lf, Lf), Br ((3,"three"), Lf, Lf)), Br ((6,"six"),
Br ((5,"five"), Lf, Lf), Br ((7, "seven"), Lf, Lf)));;

(*función que busca si la clave está, y si la encuentra devuelve su valor (string)*)
let rec search tree k =
```

```
 match tree with
  Lf -> raise Not_found
  | Br ((k',v), l, r) ->
   if k = k' then v
   else if x > k then search r x
    else search l x;;
val search : ('a * 'b) tree -> 'a -> 'b = <fun>

let rec insert tree k v =
 match tree with
  Lf -> Br ((k,v),Lf,Lf)
  | Br ((k',v'),l,r) ->
   if k = k' then Br ((k,v),l,r)
   else if k > k' then Br ((k',v'), l, insert r k v)
    else Br ((k',v'), insert l k v, r);;

(*CUESTIONES*)
11.1)
let presente x tree =
 match tree with
  Lf -> false
  | Br (y, l, r) -> x = y || presente x l || presente x r;;
val presente : 'a -> 'a tree -> bool = <fun>

11.2)
let rec flip_tree tree =
 match tree with
 Lf -> Lf
 | Br (x, l, r) -> Br (x, flip_tree r, flip_tree l);;

11.3)
(*Si tienen la misma estructura (independientemente de su clave), devuelve true; si alguno de los
dos árboles tiene estructura distinta devuelve false, y al ser un &&, devolverá false*)
let rec equal_shape tr1 tr2 =
 match tr1, tr2 with
  Lf, Lf -> true
  | Br (_,l1,r1) , Br (_,l2,r2) ->
   equal_shape l l2 && equal_shape r r2
  | _ , _ -> false;;
val equal_shape 'a tree -> 'b tree -> bool = <fun>

11.4)
let rec tree_of_list d =
 match d with
  [] -> Lf
  | (k,v)::t -> insert (tree_of_list t) k v;;
val tree_of_list : ('a * 'b) list -> ('a * 'b) tree = <fun>

11.5)
let combine d1 d2 =
 tree_of_list (list_of_tree d1 @ list_of_tree d2);;
val combine : ('a * 'b) tree -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
```

11.6)
type 'a mtree = Branch of 'a * 'a mtree list

```
let rec sum n =
 match n with
  [] -> 0
  | h::t -> h + sum t;;

let rec map f l =
 match l with
  [] -> []
  | h::t -> f h :: map f t;;

let rec size (Branch (e,l)) =
 1 + sum (map size l);;

let rec total (Branch (e,l)) =
 e + sum (map total l);;

let rec map_mtree f (Branch (e,l)) =
 Branch (f e, map (map_mtree f) l);;
```

12) ENTRADA / SALIDA

(*Usaremos ; para que realice el print; el tipo unit es un tipo que no devuelve nada*)
print_int -> imprime un valor de tipo int por pantalla
print_newline () -> deja una línea en blanco
print_string -> imprime un string
read_line y read_int -> lee del usuario un string y un int respectivamente
int_of_string -> convierte el string en integer (string_of_int hace lo contrario)

```
let print_dict_entry (k,v) =
 print_int k; print_newline (); print_string v; print_newline ();;
val print_dict_entry : int * string -> unit = <fun>

let rec iter f l =
 match l with
  [] -> ()
  | h::t -> f h; iter f t;;
let print_dict =
 iter print_dict_entry;;
```

read_line (*devuelve el string que se ha introducido hasta el primer salto de línea*)
read_int y read_float (*lee y aplica int_of_string y float_of_string*)

13) PUNTEROS

Como hemos visto hasta ahora, cuando asignamos un valor a un nombre ese valor no cambia; aunque a veces nos convendría que ese valor pueda ser modificado. Podemos hacerlo con la palabra reservada ref.

```
let x = ref 0;; (*asignamos un valor inicial de 0 a x, tiene el tipo int ref*)
val x : int ref = {contents = 0}
(*Podemos extraer el contenido de x usando el operador !, el cual tiene tipo 'a ref -> 'a*)

let p = !x;;
val p : int = 0

(*Mediante := podemos actualizar el valor de x, tiene tipo 'a ref -> 'a -> unit*)
x := 50;; (*cambiamos el valor de x*)
let q = !x;; (*el valor de q es 50*)
p;;       (*el valor de p no ha cambiado, sigue siendo 0*)

let swap a b =
 let t = !a in
  a := !b; b := t;;
(*intercambia los valores de dos 'a ref*)
```

Podemos omitir el else de la estructura if...then...else si va a ser ()

```
if x = y then
 (a := !a + 1              (**Pueden sustituirse los paréntesis con begin y end)
  b := !b - 1)
 else
  c := !c + 1;;
```

Para repetir una acción podemos usar la estructura for ... = ... to ... do ... done

ej: for x = 1 to 5 do print_int x: print_newline () done;;

Bucle while, evalúa una condición n veces hasta que sea false (while ... do ... done)

```
let smallest_pow2 x =
 let t = ref 1 in
  while !t < x do
   t := !t * 2
  done;
  !t;; (*calcula la potencia de 2 igual o mayor que el número introducido*)
```

ARRAYS, se crean mediante [|1;2;3|] este ejemplo sería un int array
Podemos acceder al contenido de la posición mediante nombre.(posición); CUIDADO, LAS POSICIONES DE UN ARRAY EMPIEZAN EN 0

```
let a = [|1;2;3;4;5|]
val a : int array = [|1;2;3;4;5|]

a.(0) -> 1;; (*si intentamos acceder a una posición no válida lanzará una excepción out of bounds*)
a.(0) <- 100;; (*podemos cambiar los valores haciendo una reasignación*)
val a : int array = [|100;2;3;4;5|]
```

ALGUNAS FUNCIONES DEL MÓDULO ARRAY

Array.length 'a array (*muestra el numero de elementos de un array*)
Array.make int 'a (*crea un array de int elementos con el contenido de 'a*)
Array.make 5 'c' (*creará un array de tamaño 5 con todo 'c'*)

(*CUESTIONES*)
13.1)
```
let x = ref 1 in
 let y = ref 2 in
  x := !x + !x;
  y := !x + !y;
  !x + !y
```
(*Valores iniciales: x = 1 ; y = 2
Valores finales: x = 2 ; y = 4
Tipo de la expresión: int , respuesta = 6*)

13.3)
```
let rec for_loop f n m =
 if n <= m then
  begin
   f n;
   for_loop f (n+1) m
  end
```

13.4)
```
[|1; 2; 3|]                 (*int array*)
[|true; false; true|]       (*bool array*)
[|[|1|]|]           (*int array array*)
[|[1; 2; 3]; [4; 5; 6]|]            (*int list array*)
[|1; 2; 3|].(2)             (*int*)
[|1; 2; 3|].(2) <- 4                (*unit*)
```

13.5) (*suma de los elementos de un array*)
```
let rec suma_array a =
 let sum = ref 0 in
  for x = 0 to Array.length a -1 do
   sum := !sum + a.(x)
  done;
  !sum;;
```

13.6) (*invertir un array*)
```
let rec array_rev a =
 if Array.length a > 1 then
  for x = 0 to Array.length a / 2 - 1 do
   let t = a.(x) in
    a.(x) <- a.(Array.length a - 1 - x);
    a.(Array.length a - 1 - x) <- t         (*intercambia los valores hasta llegar al valor situado en la
mitad*)
   done;;
```

```
13.7) (*crea una tabla de multiplicación con ese número*)
let table n =
 let a = Array.make n [| |] in
  for x = 0 to n - 1 do
   a.(x) <- Array.make n 0
  done;
  for y = 0 to n - 1 do
   for x = 0 to n - 1 do
    a.(x).(y) <- (x + 1) * (y + 1)
   done
  done;
  a;;
val table : int -> int array array = <fun>

13.8) (*pasa de mayus a minus y viceversa*)
let uppercase x =
 if int_of_char x >= 97 && int_of_char x <= 122
  then char_of_int (int_of_char x - 32)
 else x;;

let lowercase x =
 if int_of_char x >= 65 && int_of_char x <= 90
  then char_of_int (int_of_char x + 32)
 else x;;
```

14) OTROS NÚMEROS

El tipo float, las operaciones se realizan con el operador seguido de un . ; para los números negativos pondremos -. (+. -. *. /. -.2-5)

FUNCIONES CON FLOATS

sqrt -> raiz cuadrada de un numero
log -> logaritmo natural
log10 -> logaritmo base 10
sin -> seno de un ángulo (en radianes)
cos -> coseno de un ángulo (en radianes)
tan -> tangente de un ángulo (en radianes)
atan -> arcotangente de un ángulo (en radianes)
ceil -> redondea hacia arriba hasta el próximo número entero
floor -> redondea hacia abajo hasta el próximo número entero
print_string / float_of_int / int_of_float / float_of_string / string_of_float

TEORÍA DE CLASE

'\065';; (*devuelve el char correspondiente en la tabla ASCII*)
Char.code char -> int correspondiente en ASCII

Lager) let <f> = function <x> -> <e>
Lazy) let <f> <x> = <e>

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

Forma Curry: usando el operador y los elementos
(+) 3 4;; (^) "para" "sol" (@) l1 l2

```
let fact n =
 let rec aux i f =
 if i = n then f
 else aux (i+1) ((i+1)*f)
in aux 0 1;; (*factorial recursiva terminal*)

let sumlist l =
 let rec aux s l = match l with
  [] -> s
  | h::t -> aux (s + h) t
 in aux 0 l;; (*suma de los elmentos de un lista de forma recursiva terminal*)

let rev l =
 let rec aux li list = match list with
  [] -> []
  | h::t -> aux (h::li) t (*al concatenar con :: el elemento se coloca delante*)
 in aux [] l;; (*li es una lista que recoge los valores con cons (de forma inversa)*)

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

let fib n =
 let rec aux i f a =
 if i = n then f
 else aux (i+1) (f+a) f
in aux 1 1 0;;
```

TIPO REGISTRO

```
type persona {nombre: string; apellido : string; edad : int}

let pepe = {nombre = "Pepe"; edad = 57; apellido = "Fernandez"};; (*los valores pueden ponerse en
cualquier orden*)
```

Para acceder a los campos hacemos por ejemplo: pepe.edad;;

```
let mas_viejo p =
 {nombre = p.nombre;edad = p.edad + 1};;

let old_mary = mas_viejo maria;;

type persona = {nombre:string ; mutable edad: int};;
```

Ahora que la edad es mutable cambiaremos su valor así:

```
# maria.edad <- 19;;
- : unit = ()
- : persona = {nombre = "Maria"; edad = 19}

let envejece p =
 p.edad <- p.edad + 1;;
```

PROGRAMACIÓN ORIENTADA A OBJETOS

Una CLASE define como se crea, se comporta y es un OBJETO; tiene unas variables de instancia y unos métodos. Un OBJETO es una instancia de una clase

```
class nombre_clase p1...pn =
 object [(alias(*equivalente al this de Java*))]
 val [mutable] nombre_var1 = expr1 (*si es mutable puede variar si valor*)
 ...
 val [mutable] nombre_varm = exprm

 method nombre_metodo pm1...pmn = expr1
 ...
 method nombre_metodo pm1...pmn = exprm
end;;
```

Los atributos son siempre privados, si queremos usarlos fuera de la clase debemos definir getters y setters; los métodos son públicos, pero podemos hacerlos privados mediante:
method private nombre_metodo p1...pn = expr (*los provate pueden usarse en la misma clase y en sus subclases*)
Si un método no tiene argumentos, al getter no hace falta pasarle nada, pero al setter hay que pasarle un '()' (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;;
```

NUEVO OBJETO: let p1 = new nombre_clase p1...pn
let make_point x y = new point (x,y);; (*función factoría que crea un punto*)
let p2 = make_point 2 3;;

PARA LLAMAR A LOS MÉTODOS USAMOS # id_objeto #nombre_metodo pm1...pmn

Cada objeto es como cualquier otra variable, podemos usar operadores comparativos (para comparar físicamente usamos '==' y '!='; ya que '= 'y '<>' comparan las direcciones de memoria).

RELACIONES. AGREGACIÓN

Podemos almacenar datos de muchas formas:

```
class arista1 (p1,p2) =
 object
  val vertice_A = p1
  val vertice_2 = p2
 end;;

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

class arista3 (p1,p2) =
 object
  val vertices = [p1;p2]          (*con listas*)
 end;;

class arista3 (p1,p2) =
 object
  val vertices = [|p1;p2|]        (*con arrays*)
 end;;
```

HERENCIA

```
class nombre_subclase p1...pn =
 object
 inherit nombre_superclase pp1...ppm as alias (*suele usarse super en alias*)
  val ...
  method ...
 end;;
```

Prevalece la última definición para los atributos o variables, la herencia múltiple está permitida.

Métodos y atributos abstractos (*declaramos solo su interfaz*)

```
method virtual nombre_metodo p1...pn : tipo
val [mutable] virtual nombre_atributo : tipo
```

Una clase es abstracta si tienen algún método o atributo abstracto (añadimos virtual después de class)
Si una clase hereda de otra abstracta, debe implementar los métodos de dichas clases ya que solamente están declarados. En el caso de que una subclase herede de una virtual y de otra que ya implementa los métodos de la virtual, no tendrá que redefinirlos.

POLIMORFISMO A NIVEL DE OBJETO

```
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;;
ej:
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        (*los tipos ya están instanciados, no se pueden cambiar*)
```

Si ahora queremos usar herencia en las clases parametrizadas, podemos hacer:

```
1) class [‘c, ‘d ] ppair (xx : ‘c)(yy : ‘d) =
    object
    inherit [‘c, ‘d ] par xx yy
    method to_pair() = (x,y)
   end;; (*el inherit indica cual va a ser el 'a y el 'b*)

2) Podemos especializar la clase a unos tipos concretos
class int_float_par xx yy =
  object
  inherit [int, float] par xx yy
end;;

3) Podemos solamente concretizar alguno de los tipos polimórficos, pero no todos
class ['a] int_x temperatura_par xx (yy = 'a) =
 object
 inherit [int, 'a] par xx yy
end;;
```

OPERACIONES MÓDULO LIST

```
type 'a t = 'a list =
 | []
 | (::) of 'a * 'a list

length l (*tamaño de una lista*)
```

compare_lengths l1 l2, compare_length_with l1 int (*0 si son iguales, 1 si l1 > l2, -1 si l1 < l2*)
cons x xs (*x::xs*)
hd l1 y tl l1 (*head y tail de una lista*)
nth l1 n (*devuelve el n-ésimo elemento de la lista*)

```
let rec nth l1 n =
 match l1 with
  [] -> raise (Invalid_argument "nth")
  | h::t ->
   if n > length l1 - 1 then raise (Invalid_argument "nth")
    else if n = 0 then h
     else nth t (n-1);;
```

rev l1 (*invierte la lista*)
append l1 l2 (*concatena dos listas mediante @*)
rev_append l1 l2 (*invierte l1 y lo concatena con l2*)
concat 'a list list (*concatena dos listas o más que estén dentro de otra lista*) (*flatten es la función recursiva terminal*)

equal: ('a -> 'a -> bool) -> 'a list -> 'a list -> bool
compare: ('a -> 'a -> int) -> 'a list -> 'a list -> int

iter: ('a -> unit) -> 'a list -> unit
map ('a -> 'b) -> 'a list -> 'b list

```
let rec fold_left f e l = match l with
 [] -> e
 | h::t -> fold_left f (f e h) t;; (*aplica la funcion f a una lista*)
```

List.fold_left es una función que itera sobre los elementos de una lista comenzando por el primero. Cada vez que se itera, se aplica una función de dos argumentos a los elementos de la lista, empezando por el primero, y se acumula el resultado.

List.fold_right es una función que itera sobre los elementos de una lista comenzando por el último. Cada vez que se itera, se aplica una función de dos argumentos a los elementos de la lista, empezando por el último, y se acumula el resultado.

Si introducimos la misma función obtendremos el mismo resultado para ambas funciones, para obtener diferentes resultados deberemos modificar la función

```
let list = [1;2;3;4]

List.fold_left (fun acc x -> acc * x) 1 list;;          (* Resultado: 24 *)

List.fold_right (fun x acc -> acc * x) list 1;;         (* Resultado: 24 *)
```

exists f [...] (*comprueba si al menos un elemento cumple con f*)
mem a set (*comprueba si el elemento a existe en la lista*)
find f l (*devuelve el primer elmento de la lista que cumple f; filter devuelve todos los elementos que cumplen f*)
split [(a1,b1);...;(an,bn)] (*convierte una lista de pares en dos listas con cada par*)
combine [a1;...;an][b1;...;bn] -> [(a1,b1);...;(an,bn)]
sort : ('a -> 'a -> int) -> 'a list -> 'a list

merge ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list (*ordena l1 y l2 mediante cmp y devuelve una lista con las dos fusionadas y ordenadas*)

OPERACIONES MÓDULO STRING

make n c (*crea un string de longitud n con el caracter c*)
length (*longitud de un string*)
concat (*concatena un string en el medio de una lista de strings*)
cat (*hace lo mismo que ^ *)
equal, compare, contains
sub s p l (*string de longitud l, que contiene s que empieza en la posición p*)

OPERACIONES MÓDULO ARRAY

length (*longitud de un array*)
make n x; init n f; append a1 a2; fill a pos len x; sort f a;