Dentro del archivo:

```
        print_endline "Hello, world!"
```

Guardarlo como hello.ml

En la pantalla de comandos

```
ocamlc hello.ml -o hello
ocamlrun hello
./hello
```

Hacer otro archivo:

```
        print_float (2.0 *. asin 1.0);;
        print_endline "";;
```

Guardalo como pi.ml

En la pantalla de comandos

```
ocamlopt pi.ml -o pi
ocamlrun pi
./pi
```

Abrimos ocaml poniendo en la pantalla de comandos:

```
ocaml (Tiene que salir un #)
```

Despues introducir 2 + 1;;
Devuelve un -: int = 3, es decir, nos dice el tipo de dato que nos devuelve y el valor
Si introducimos #"Hello, World!" nos saldrá un -: string = "Hello, world!"
Si introducimos #2.0;; nos devuelve un -: float = 2 (Si no pusiesemos el 0 también sería float)
El + solo suma enteros en ocaml (Comprobarlo haciendo 1.5 + 1.5)(Es un error de tipos)
Para hacer una suma con números flotantes hace falta poner un . (+.)
El / nos devuelve el cociente de la división entera, si queremos el resto ponemos mod
Si ponemos 7 / 0 se interrumpe el programa y se produce un error de ejecución (una excepción)
Si ponemos 7 / ;; se produce un error de compilación
Si ponemos una palabra que no conoce (como pepe) también se produce un error de compilación
Al poner 'a';; nos devuelve un tipo char, sin embargo, si ponemos "a" nos devuelve un string
Cuando ponemos 'ab';; se produce un error de sintaxis
Si ponemos '\n' o \t' o '\064' nos devolverá un tipo char
Una operación de comparación sería '\064' = '@';; y nos devolvería un tipo bool
Los tipos básicos son: char, int, float, bool y string (falta 1)
Si ponemos 'a' = "A";; nos da un error de tipo
Con strings también hay operaciones como: "ABC" ^ "defg";; (Une las dos strings)
Para las operaciones, los paréntesis se utilizan como en C
Hay tipos/operaciones que ya tienen nombre como:

```
        -max_int;; (El entero más grande de ocaml)
        -min_int;; (El entero más pequeño de ocaml)
        -asin 1.0;; (El arcoseno de 1)
```

-etc

Si ponemos solamente asin;; no daría error, sino que nos daría un dato de tipo float (comprobar),
como los char, etc.
La operación abs es una función de tipo int a int (Comprobarlo poniendo abs;;)(Es la operación de
valor absoluto)
Si ponemos abs -4;; daría error (Es necesario paréntesis)

Consejo: NO UTILIZAR TABULADOR (Utilizar espacios en blanco)
Se puede descargar ledit para que se pueda editar mejor

Continuando con lo visto en la clase anterior:

Si ponemos simplemente print_endline;; nos dará un tipo string -> unit = <fun>, es decir, al
escribir un print_endline con una cadena de caracteres nos devolverá un tipo unit.
El único tipo unic es (). (Se puede observar haciendo un print_endline "hola";;)
El tipo de dato unit es un tipo base. (Es parecido al tipo void de C)
print_float también nos devolverá un tipo unit.
Hay funciones de cualquier origen a cualquier destino, es decir, hay funciones que devuelve de
un int -> int, int -> bool, etc.
La función not es una función de tipo bool (Un ejemplo sería not (2<3);;)
La función char_of_int es una función que devuelve la letra ASCII correspondiente a un int, es
decir, devuelve un int -> char (Un ejemplo sería char_of_int 65;;). Lo mismo pasaría para la
función int_of_char. Hay un problema porque la tabla ASCII tiene carácteres limitados, por lo
que si ponemos: char_of_int 300 nos dará un error de ejecución.
Los nombres de valores en Ocaml SIEMPRE empiezan por una letra en minúscula.
Hay otros nombres que si que empiezan en mayúscula como: String.length, que nos devuelve un
valor de tipo string -> int (Un ejemplo sería String.length "ABCD";;), pero, en este caso,
el nombre real de la función empieza por minúscula (length) y lo que empieza en mayúscula es el
nombre del módulo de la función. Por ejemplo, la función asin tiene un módulo que es Stdlib,
pero no hace falta utilizarlo porque ya se da por hecho. (Se puede probar poniendo: Stdlib.asin;;)
En ocaml.org tenemos el manual de ocaml para poder ver como funcionan las diferentes operaciones.
Hay un módulo que es Char (Ejemplo: Char.code;; que es igual que char_of_int)(Ejemplo: Char.chr;;
que es igual que int_of_char)
La función truncate nos devuelve un dato de tipo float -> int.
La función sqrt es la raíz cuadrada
Para elevar al cuadrado es necesario poner, por ejemplo: (2.0 *. asin 1.) ** 2.;;
Hay un tipo de frases que son las definiciones y sirven para asociar un nombre con un valor.
La frase es: let (nombre al que queremos darle un significado) = (cualquier expresión válida en caml)
Un ejemplo de esta frase es: let pi = 2. *. asin 1.;;
Ahora si pusiésemos pi en ocaml nos daría el valor que le hemos dado a esa palabra
Otro ejemplo de esta función es: let arcsen = asin;;
Se pueden reutilizar los nombres, es decir, si ponemos: let asin = 3;; no dará error, pero creará
una ambiguedad. El compilador utiliza la actualización más reciente. Para poder

acceder a la
función asin después de hacer el cambio se tendría que poner: Stdlib.asin

La conversión de tipos en ocamel es EXPLÍCITA siempre

Las definiciones que hemos visto son definiciones globales, pero también existen definiciones locales
Estas definiciones solo se utilizan en la expresión que se va a utilizar
Estas definiciones se realizan siguiendo el siguiente esquema: <def> in <e>
Un ejemplo de esto es:
#let pi = 3. in pi *. pi +.sqrt pi;;
#pi;;
Como se puede observar el valor de pi no ha cambiado globalmente, pero si que cambió en la
expresión anterior. También se pueden poner expresiones como la siguiente:
let y = let pi = 1024 in pi * pi * pi;;
Not es la función que pasa lo falso a verdadero y lo verdadero a falso (se puede comprobar poniendo not;;)
En caso de que una función que queramos no tenga nombre en ocaml se haría de la siguiente manera:
(En este caso vamos a hacer la función not)
function true -> false | false -> true;; (Probar poniendo:(function true -> false | false -> true;;)(2<3))
Para darle un nombre hay que poner un let: let no = function true -> false | false -> true;;
Los lenguajes funcionales se basan en el λ-cálculo
Si ponemos let f = function true -> false;;
Cuando se aplique esta función, si la expresión es true sabrá el resultado, pero si es false dará un
error, ya que al definir la función no hemos definido todos los casos posibles.
Una función que no es exahustiva puede provocar que el programa se interrumpa.
Si ponemos let g = function true -> true | false -> true;; , y luego g (2 < 3);; sabemos que el
resultado va a ser true, pero ocaml siempre va a comprobarlo aunque solamente haya un resultado posible.
Hay dos tipos de evaluaciones, la evaluación eager y la lazy. Ocaml tiene una evaluación eager.
La evaluación lazy no comprobaría el resultado anterior, mientras que la eager si. La evaluación
eager es más eficiente, ya que si se produce el siguiente caso: g(2 < 3 / 0) la evaluación eager
dará error y la lazy dará true.
Si ponemos g = function _ -> true;; (La _ significa 'Cualquier cosa') El compilador no sabe que tipo
de dato es _, por lo que interpreta que es todo tipo de datos. A esto se le llama plimorfismo.
Podemos probar los siguientes comandos: g true;; g 3;; g "hola";; g asin;;

let alltrue = function _ -> true;;
Esta función hace que todos los tipos de datos introducidos devuelvan un true.
(Ej: alltrue asin;;)
let alltrue = function (_:int) -> true;;
Esta función hace que todos los enteros introducidos devuelvan true, pero no otro tipo de dato.
También se puede definir de la siguiente forma: let alltrue: int -> bool = function _ -> true;;
Hay que fijarse que al compilar nos devuelve los datos que nosotros queremos.
En ocaml se pueden definir funciones de la siguiente manera: let doble = function n -> 2 * n;;
Si primero definimos el valor de una variable (x) y luego definimos una función (function x -> 2*x) no
va a dar ningún problema, ya que la x, en la función, no tendrá el valor predefinido.
Si introducimos (function x -> 2 * x) (2 + 1), nos devolverá el valor de la función que está
relacionado con el resultado de 2 + 1, en este caso devolverá 6.
Muchas veces saldrán definiciones con la siguiente estructura: let <f> = function <x> -> <e>, como
let alltrue = function x -> true;;
Esta expresion se puede simplificar así let <f><x> = <e>, por ejemplo: let doble x = 2 * x;;
Otro ejemplo sería: let sqr x = x * x;;
Se puede definir la aplicación identidad: let id = function x -> x;;
Se pueden utilizar varias expresiones al mismo tiempo, como: id doble 3;;
Otro ejemplo de esto es: doble (sqr 3);; pero si ponemos (doble sqr) 3;; dará error de ejecución.
Si no hubiese paréntesis empezará por la izquierda y, en el último ejemplo, daría error.
doble (sqr 3);; Esto es una composición de funciones
Esta composición también se puede escribir de la siguiente manera: 3 |> sqr |> doble;;

Para salir de ocaml: #quit;;
Un ejemplo de λ-conversión es: function x -> 2 * x;;
let sqr = function x -> x * x;;
let cuadrado = function x -> sqr x;;
Cuando tengo una función en la que se produce un: function x -> f(x), se puede simplificar en f(x)
Un ejemplo de η-reducción para las dos funciones anteriores es: let cuadrado = sqr;;
Si hacemos sqr = cuadrado nos dará un error porque no son la misma función, ya que no es igual
de eficiente definir una función de una manera o de otra.
Las funciones son valores.

```
let rec fact = function
        n -> if n = 0 then 1
                else n * fact (n - 1)
```

La palabra reservada rec se utiliza para hacer recursión

(if <b> then <e1> else <e2>):α (e1 y e2 tienen que ser del mismo tipo de datos)
Otro ejemplo sería:

```
let absf = function
        x -> if x >= 0. then x else -. x;;
```

```
let abs x =
    let f = function true -> 1 | false -> -1 in
    f(x > 0) * x;;
```

Lo malo de utilizar esa definición es que cada vez que utilice la función abs se va a definir la
aplicación local.

```
let abs x =
    let f = function true -> 1 | false -> -1 in
    function x -> f (x > 0) * x;;
```

De esta forma la definición de aplicación local solo se utiliza una vez.
Otro ejemplo:

```
let circ r =
    let dospi = 4. *. asin 1. in
    dospi *. r;;
```

```
let circ r =
    let dospi = 4. *. asin 1. in
    function r -> dospi *. r;;
```

if <b> then <e1> else <e2> es lo mismo que (function true -> <e1> | false -> <e2>) <b>
Poner "hola" == "ho" ^ "la" da falso

Al hacer let x = 13;; no estamos creando una variable
let sumarx = function y -> x + y;; En esta función estamos haciendo que al número que pongamos se le
suma 13 pero, al cambiar de valor la x (let x = 0;;) la función sumarx no cambiaría.
let f = function x -> (function y -> y+x);; Es una función de int a una función de int a int.
Si ponemos f 3;; nos devolverá int -> int, pero si ponemos (f 3) 5;; nos devolverá un entero.
El argumento de f es el 3, y 5 es el argumento de (f 3)
Si se pone (+);; pasa a ser la operación curry que esconden los operadores binarios.
Se puede poner (+) 2 3;; y sería lo mismo que poner 2+3;; Esto pasaría igual con los demás operadores.
Otro ejemplo sería: (^) "Hola, ";;
Se le puede dar un nombre: let saluda = (^) "Hola, ";; y ejecutarlo poniendo saluda pepe;;
Para cualquiera de los tipos de datos, su producto cartesiano también es un tipo de datos.
Un ejemplo de producto cartesiano es: 1+1, 2;; Otro sería: 1+1, not true;;
Si pones fst(2,3), nos devolverá el primer elemento del producto cartesiano, y snd(2,3) el segundo.
Se puede hacer una función que sea: let suma = function p -> fst p + snd p;;
También se puede poner let suma = function (a,b) -> a + b;;

Siguiendo la clase anterior, se pueden utilizar productos cartesianos para definir funciones, como
let suma = function x, y -> x + y;; También se podría definir de la siguiente manera:
let suma (x, y) = x + y;; o también se podría definir como let suma = function x -> function y -> x+y;;
o también let suma x = function y -> x + y;; o también let suma x y = x + y;;
Utilizando funciones curry, es lo mismo definir la función anterior de la siguiente manera:
let suma x y = (+) x y;; o let suma = (+);;
En el caso del producto, hay que introducir unos espacios entre el * y las () para evitar ambiguedades. (*)
Una definición curry sería: ('a*'b -> 'c) -> ('a -> 'b -> 'c);; (ES UN EJERCICIO DE EXAMEN)
Lo contrario a una función curry sería una función uncurry, que se define de la siguiente manera:
('a -> 'b -> 'c) -> ('a*'b -> 'c);;

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

En la anterior función es obligatorio poner rec antes de definir la función, para que el compilador
sepa que la función que está dentro de si misma hace referencia a la misma función.
Si intentamos hacer el factorial de 300000 nos dará un error de ejecución, porque se produciría
un overflow en el stack

Si queremos 300000, también podemos ponerlo como 300_000 (Sería como poner un . (300.000))
Utilizando la función de factorial definida la clase anterior, al poner el factorial de un número
negativo se produce un overflow porque nunca llegaría a cumplirse la condición del if.
Si ponemos el factorial de 20 y el factorial de 30 podemos observar un error, ya que nos devuelve
que el factorial de 20 es más grande que el de 30. Esto se produce porque el factorial de 30 es más
grande que el entero más grande que tiene ocaml (poner max_int;; y min_int;;).
Si ponemos max_int + 1
el programa nos devuelve el min_int, ya que ocaml tiene una aritmética cíclica con módulo 2^63.
El factorial de 63 es el último factorial que da un resultado real.

Definimos la función let rec quo x y =
```
                        if x < y then 0
                        else 1 + quo (x - y) y;;
```
Esta función deja cuentas pendientes (porque antes de hacer la suma en el else tiene que hacer el quo),
lo que dejaría un overflow en muchos casos.
Una función que no tiene cuentas pendientes es: let rec rem x y =
```
                                                  if x < y then x
                                                  else rem (x - y) y;;
```
La función que te devuelva lo mismo que las dos anteriores funciones sería:
```
let rec div x y =
    if x < y then 0, x
    else let q, r = div (x - y) y in
         1 + q, r;;
```

En una función doblemente recursiva, como la de fibonacci, el tiempo que tarda en calcular valores
muy elevados es muy grande, y cada vez que aumenta en uno el valor requerido para la función, el
tiempo también se aumenta (Más o menos el doble). Esto se puede ver como:
sqrt(s)*T(n-1) < T(n) < 2*T(n-1).
La función time del módulo Sys (Sys.time) nos devuelve el tiempo de cpu que ha consumido el proceso
que se está ejecutando. Puede ser muy util para calcular el tiempo que tarda una función en
ejecutarse.
Para estimar el tiempo que tarda se hace el siguiente calculo: k * T(n - 1)
k = (1. +. sqrt 5.) /. 2.
Otra definición de la función de fibonacci sería:

```
let fib n =
    let rec fib2 = function
        1 -> 1, 0
      | n -> let f1, f2 = fib2 (n-1) in
             f1 + f2, f1;;
        in fst (fib2 n)
```

En esta definición, la función no es doblemente recursiva, por lo que los resultados se mostrarán de
forma casi instantánea.

Hablando de otros tipos de datos de los que no se han hablado. Uno de ellos la int list, que se
puede ver ejecutando: [1; 2; 3; 4;];;
Esto son secuencias con un número finito de elementos.
Estas listas pueden ser de cualquiera de los tipos de datos que hay en ocaml, como char list,
float list, string list, (int * int) list, ...
Hay muchas funciones en el módulo list que trabajan con estos tipos de datos, como:
List.length, List.hd (devuelve el primer elemento de la lista), List.tl (devuelve la lista, pero sin
el primer elemento),

En el módulo stdlib hay una función raise que toma un tipo de dato exn y devuelve un 'a
Si ponemos Division_by_zero;; nos devolverá un dato de tipo exn.
Failure "(Cualquier String)", nos devuelve un dato de tipo exn
Una nueva función de ocaml es: match <e> with <p1> -> <e1> | <p2> -> <e2> | ... | <pn> -> <en>
Una lista se crea de la siguiente manera: let l = ['a'; 'e'; 'i'; 'o'; 'u'];;
Otra forma de crear una lista es: 'X' :: ['z'; 'c'];;
De esta forma, podemos crear una lista sabiendo la cabeza y la cola.
Si ponemos let h::t = l;; le asocia a h la cabeza de la lista y a t la cola de la lista. En caso de
que la lista fuese vacía se producirá un error.

Una nueva función es compare, que compara dos valores. Devuelve un número mayor que 0 si el primero es
mayor que el segundo, devuelve 0 si son iguales y devuelve un número menor que 0 si el primero es
menor.
Otra función parecida a la anterior es: List.compare_lengths, que compara la longitud de dos listas.
A la hora de comparar las longitudes de dos listas, utilizar esta función no es la forma más eficiente
de hacer esta tarea. Una forma más eficiente de realizar esta tarea es:

```
let rec compare_lengths = function
    [] -> (function [] -> 0
                  | _::_ -> -1)
  | h::t -> (function [] -> 1
                    | h2 :: t2 -> compare_lengths t t2);;
```

Otra forma de realizar esta función de una manera más cómoda y sencilla de entender:

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

Esta sería otra función que podríamos utilizar con listas.

```
let rec mem x = function
    [] -> false
  | h::t -> x = h || mem x t;;
```

En este caso, realizar esta función utilizando un match with no sería más útil, ya que tendríamos que
definir otra "variable" innecesariamente.

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

Otra función para contar los elementos de una lista de forma eficiente es:

```
let rec suma_length s = function
        [] -> s
      | _::t -> suma_length (s+1) t
```

Otra función sería: `let length l = suma_length 0 l;;`

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

let rec fact_aux (p, i) =
    if i = 0 then p
    else fact_aux (p*i, i-1);;

let fact n = fact_aux (1, n);;

let fact n =
    let rec aux i p =
        if i = 0 then p
        else aux (i - 1) (p*i)
    in aux n 1;;

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

let rec fib n =
    if n > 1 then fib (n - 1) + fib (n - 2)
    else n;;        (*Esta definición es muy ineficiente*)

let fib n =
    let rec fib_aux (i,f,a) =
        if i = n then f
        else fib_aux (i+1, f+a, f)
    in fib_aux(0,0,1)

let rec lmax = function
    h :: [] -> h
  | h :: t -> max h (lmax t);;

let lmax l =
    let rec aux m = function
        [] -> m
      | h::t -> aux (max m h) t
    in aux h t;;
(*Esta función no define el caso en el que la lista está vacía*)

let rec lmax = function
    h::[] -> h
  | h1::h2::t -> lmax (max h1 h2::t)
```

```
let rec append l1 l2 = match l1 with
    [] -> l2
  | h::t -> h :: append t l2;;

let rec append' l1 l2 = match l1 with
    [] -> l2
  | h::t -> append' t (h::l2);;
```

Un ejemplo al ejecutar esta función sería:
append' [1; 2; 3] [4; 5] = append' [2; 3] [1; 4; 5] = append' [3] [2; 1; 4; 5] = append [] [3; 2; 1; 4; 5] = [3; 2; 1; 4; 5]

```
let rev l = rev_append l [];; (Esto creo que vale para una práctica)

let tail_append l1 l2 =
    rev_append (rev l1) l2;;

let rec fold_left op e = function
    [] -> e
  | h::t -> fold_left op (op e h) t;;

let rec fold_right op l e = match l with
    [] -> e
  | h::t -> op h (fold_right op t e);;

let rec sumlist = function
    [] -> 0
  | h::t -> h + sumlist t;;
```

let sumlist l = fold_left (+) 0 l;; (Esta función es más eficiente, porque no da ningún tipo de stack overflow (Porque es una función terminal), mientras que la anterior si)

```
let length l = fold_left (function s -> function _ -> s + 1);;

let rec lmax = function
    h::[] -> h
  | h::t -> max h (lmax t);;

let lmax (h::t) =
    fold_left max h t;;

let last (h::t) = fold_left (fun _ y -> y) h t;;
```

Las funciones que devuelven booleanos son predicados

```
let rec for_all f = function
    [] -> true
  | h::t -> if f h
            then for_all f t
            else false;;

let rec for_all f = function
    [] -> true
  | h::t -> f h && for_all f t;;

let for_all l = fold_left (fun b x -> b && f x) true l
```

Esta función es un ejemplo en el que no se debería utilizar el fold_left, porque hay un problema de
eficiencia, ya que con el fold_left se tiene que recorrer toda la lista, mientras que en las anteriores
definiciones no siempre recorre toda la lista.

Para comprobar si una lista está ordenada podemos realizar esta función:

```
sorted: 'a list -> bool
let rec sorted = function
    h1::h2::t -> h1 <= h2  && sorted (h2::t)
  | _ -> true;;
```

Si tengo que escribir una función según sea la lista (si es vacía, si me importa el 2 elemento...) hay
que utilizar function.

```
let rec insert x = function
    [] -> [x]
  | h::t -> if x <= h
            then x::h::t
            else h::insert x t;;
```

En esta última función la recursión no es terminal.

```
let rec i_sort = function
    [] -> []
  | h::t -> insert h (i_sort t);;
```

Una función para realizar pruebas de eficiencia, es decir, que mida los tiempos de ejecución es:

```
let crono f x =
    let t = Sys.time () in
        f x; (*Esto es para olvidar su valor*)
        Sys.time () -. t;;
```

List.append concatena dos listas, pero no es terminal.
Hay una forma de que esta función sea recursiva terminal:

```
let append l1 l2 =
    try List.append l1 l2 with
        Stack_overflow -> List.rev_append (List.rev l1) l2;;

let tl' l =
    try List.tl l with
        Failure_ -> [];;
```

Implementaciones terminales de las funciones insert e isort serían:

```
let rec insert x l =
    let rec aux front  = function
        [] -> List.rev (x::front)
      | h::t -> if x <= h
                then List.rev_append front (x::h::t)
                else aux (h::front) t
    in aux [] l;;

let isort' l =
    let rec aux acc = function
        [] -> acc
      | h::t -> aux (insert' h acc) t
    in aux [] l;;
```

Las siguientes funciones están realizadas a partir de las del otro día:

```
let rec insert ord x = function
    [] -> [x]
  | h::t -> if ord x h
            then x::h::t
            else h::insert ord x t;;

let rec isort ord = function
    [] -> []
  | h::t -> insert ord h (isort ord t);;
```

La ordenación por fusión es más eficiente que la ordenación por inserción

En este caso divide no es terminal

```
let rec divide = function
    h1::h2::t -> let t1, t2 = divide t in
                 h1::t1, h2::t2
  | l -> l, [];;
```

En este caso merge no es terminal

```
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;;
```

En este caso m_sort no es terminal

```
let rec m_sort = function
    [] -> []
  | [h] -> [h]
  | l -> let l1,l2 = divide l in
           merge(m_sort l1) (m_sort l2);;
```

La siguiente función crea una lista de n valores aleatorios entre 0 y 1_000_000:

```
let rlist n = List.init n (fun _ -> Random.int 1_000_000);;

let divide l =
    let rec aux l1 l2 l3 = match l1 with
        [] -> List.rev(l2),List.rev(l3)
      | h1::h2::t -> aux t (h1::l2) (h2::l3)
      | h::[] -> List.rev(h::l2),List.rev(l3
    in aux l [] [];;
```

Para resolver el problema de las ocho reinas se puede realizar el siguiente algoritmo:

```
let come (i1, j1) (i2,j2) =
    i1 = i2 || j1 = j2 || abs(i2 - i1) = abs(j2 - j1);;

let compatible p l =
    not (List.exists (come p) l);;

(* int -> (int * int) list*)

let queens n =
    let rec completar path (i,j) =
        if i > n
        then path
        else if j > n
             then raise Not_found
             else if compatible (i,j) path
                  then try completar ((i,j)::path) (i+1, 1) with
                       Not_found -> completar path (i, j+1)
                  else completar path (i, j+1)
        in completar [] (1,1);;


        y x x x x
        x x y x x
        x x x x y
        x y x x x
        x x x y x
```

Un nuevo tipo de dato sería el a option que se consigue con la función Some. Por ejemplo, si ponemos Some 3 nos devolvería un int option y si ponemos Some a nos
devolvería char option. La función None también nos devuelve un a option.

Este tipo de funciones/datos se utilizan en casos como por ejemplo:

```
let div x y =
    if y = 0 then None
    else Some (x / y);;

let hd' l =
    try Some (List.hd l)
    with Failure _ -> None;;

let queens n =
    let rec completar path (i,j) =
        if i > n
        then Some path
        else if j > n
             then None
             else if compatible (i,j) path
                  then match completar ((i,j)::path) (i+1, 1) with
                             None -> completar path (i, j+1)
                           | Some s -> Some s
                  else completar path (i, j+1)
        in completar [] (1,1);;

let all_queens n =
    let rec completar path (i,j) =
        if i > n
        then [path]
        else if j > n
             then []
             else if compatible (i,j) path
                  then completar ((i,j)::path) (i+1, 1) @
                       completar path (i, j+1)
                  else completar path (i, j+1)
        in completar [] (1,1);;
```

El a @ b junta las soluciones de a y b (como un append).

```
let no_queens n =
    let rec completar path (i,j) =
        if i > n
        then 1
        else if j > n
             then 0
             else if compatible (i,j) path
                  then completar ((i,j)::path) (i+1, 1) +
                       completar path (i, j+1)
                  else completar path (i, j+1)
        in completar [] (1,1);;
```

```
let rec print_sol = function
    [] -> print_newline ()
  | (_,y)::t -> print_int y; print_char ' ';
                print_sol t;;

let print_queens n =
    let rec completar path (i,j) =
        if i > n
        then print_sol path
        else if j > n
             then ()
             else if compatible (i,j) path
                  then (completar ((i,j)::path) (i+1, 1) ;
                        completar path (i, j+1))
                  else completar path (i, j+1)
        in completar [] (1,1);;
```

Los datos que hemos visto hasta ahora ya estaban definidos en ocaml, pero también se pueden definir
datos manualmente.
Por ejemplo:

```
type int_o_no =
     UnInt of int
   | NoInt;;

let div m n = match m,n with
    UnInt x, UnInt 0 -> NoInt
  | Unint x, UnInt y -> UnInt (x / y)
  | _ -> NoInt;;

let (//) = div;;

type booleano = V | F;;

type dia =  Lu | Ma | Mi | Ju | Vi | Sa | Do;;
```

En este caso, si hacemos Lu < Ma nos devolverá un true, pero si ponemos Ma < Lu devolverá false.
Esto se debe a el orden en el que se ha colocado al definir.

```
let festivo = function
    Sa | Do -> V
  | _ -> F;;

type otro_int = I of int;;

type nombre = Name of String;;

let soyyo = function Name "Raúl" -> true | _ -> false;;

type int2 = L of int | R of int;;

type entero = Pos of nat | Neg of nat;; (*Ese tipo nat no está definido, solo es un ejemplo*)

type numero = F of float | I of int;;
```

Con ese tipo de datos tenemos un dato que agrupa los datos de tipo int y de tipo float.

```
let suma m n = match m,n with
    I x, I y -> I (x+y)
  | F x, F y -> F (x +. y)
  | I x, F y  | F y, I x-> F (float x +. y)

let (++) = suma;;
```

```
type nat = O | S of nat;;
```

Como se puede comprobar, se pueden definir tipos de datos de forma recursiva. Este ejemplo no es
práctico, pero de forma académica nos sirve. Este tipo de datos nos devuelve infinitos datos, en este
caso: O;; S O;; S (S O);; S (S (S O));;...

```
let rec sum m n = match n with
    O -> m
  | S i -> sum  (S m) i;;

type 'a quiza =
     Algo of 'a
   | Nada;;

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

Este último tipo de dato está ya definido en el lenguaje. ('a option)

```
type 'a tree =
     V
   | N of 'a * 'a tree * 'a tree;;
```

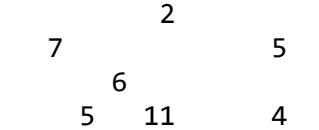
Ejemplos de este tipo de datos serían: V;; N (5,V,V);;
Este tipo de datos nos acabará dando la estructura de un árbol.

```
let t5 = N (5,V,V);;
let t6 = N (6,V,V);;

let h x = N (x, V, V);;

let t12 = N (6, h 5, h 11);;
let t22 = N (9, h 4, V);;
let t1 = N (7, h 2, t12);;
let t2 = N (5, V, t22);;
let t = N (2, t1, t2);;
```

Esto sería la construcción del siguiente árbol:




```
let rec nnodos = function
    V -> 0
  | N (r,i,d) -> 1 + nnodos i + nnodos d;;

let rec altura = function
    V -> 0
  | N (_,i,d) -> 1 + max (altura i) (altura d);;
```

```
let rec preorden = function
    V -> []
  | N (r,i,d) -> r::preorden i @ preorden d;;
```

```
let rec hojas = function
    V -> []
  | N (r,V,V) -> [r]
  | N (r,i,d) -> hojas i @ hojas d;;

type 'a sttree =
     Leaf of 'a
   | Node of 'a * 'a sttree * 'sttree;;

let rec tree_of_sttree = function
    Leaf x -> N (x, V,V)
  | Node (r,i,d) -> N (r, tree_of_sttree i, tree_of_sttree d);;

let rec sttree_of_tree = function
    V -> raise (Invalid_argument "sttree_of_tree")
  | N (r,V,V) -> Leaf r
  | N(r,i,d) -> Node (r, sttree_of_tree i, sttree_of_tree d);;

let rec hojas = function
    Leaf r -> [r]
  | Node (_,i,d) -> hojas i @ hojas d;;

let rec mirror = function
    Leaf r -> Leaf r
  | Node (r,i,d) -> Node (r, mirror d, mirror i);;

type 'a gtree =
     Gt of 'a * 'a gtree list;;

let s x = Gt (x, []);;

let d = Gt (5, [Gt (9, [s 4])]);;
let i = Gt (7, [s 2; s 10; Gt (6, [s 5; s 11])]);;

let t = Gt(2, [i;d]);;

            2
     7              5
 2  10   6           9
        5  11        4


let rec nnodos (Gt(r,l)) = List.fold_left (+) 1 (List.map nnodos l);;
```

```
let h x = Gt (x, []);;

let rec nnodos (Gt(_,l)) = List.fold_left (+) 1 (List.map nnodos l);;

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

let anchura (Gt (r,l))=
    let rec aux acc present next = match present, next with
        [], [] -> List.rev acc
      | Gt (r1,l1)::t, _ -> aux (r1:: acc) t (next @ l1)
      | [], _ -> aux acc next []
     in aux [r] l [];;

let anchura (Gt (r,l))=
    let rec aux acc next = match next with
        [] -> List.rev acc
      | Gt (r1,l1)::t -> aux (r1:: acc) (t @ l1)
     in aux [r] l;;

let rec anchura = function
    Gt (r, []) -> [r]
  | Gt (r, Gt (r1,l1)::t) -> r :: anchura (Gt (r1, t @ l1));;
```

Entramos en la parte imperativa del lenguaje.

Utilizamos esta parte del lenguaje para la entrada/salida de datos.

output_char;; Es una función de tipo: out_channel -> char -> unit

Una función de tipo out_channel es stdout;;
Un ejemplo sería: output_char stdout 'X';;
output_char stdout 'A'; output_char stdout 'B';;
let print_char c = output_char stdout c;;

En imperativo se necesitan variables, estructuras de control (;), estructuras condicionales (if then
else) y bucles.

```
print_char 'A' ; print_char 'B' ; print_char 'C' ; print_char '\n';;
```

```
let s = "ABCD";;
```

Una función para coger cada caracter de un string es: String.get;;
A esta función hay que pasarle un string y un entero. Este último indica la posición del char que se
quiere sacar del string. También se puede hacer s. [int]

```
let output_string sal s =
    let n String.length s in
    let rec loop i =
        if i >= n
        then ()
        else (output_char sal s.[i]); loop (i+1)
    in loop 0;;
```

Por ejemplo: output_string stdout "ABCD\n";;

La siguiente función es por comodidad:

```
let print_string s = output_string stdout s;;
```

Si ponemos print_string "ABCD\n";; pondrá lo mismo que antes.

Se puede definir la anterior función con un bucle:

```
let output_string sal s =
    for i = 0 to String.length s - 1 do
        output_char sal s.[i]
    done;;

let print_string s = output_string stdout s;;

let print_endline s = print_string (s ^ "\n");;
```

La función open_out "prueba" crea un fichero que se llama prueba. Si el archivo ya existiese, lo
sobreescribe (es decir, lo borra).

```
let sal = open_out "prueba";;
output char sal 'A';;
output_string sal "BCD\n";;
```

Ahora, si abrimos el fichero, no podemos ver que se ha escrito ABCD, ya que esos datos se encuentran
en el buffer de salida.
Si queremos que se actualice el fichero, tenemos que cerrar el fichero:

```
close_out sal;;
```

Ahora, si abrimos el fichero, podremos ver que se ha escrito ABCD en el fichero.
También podemos ir al terminal y poner ls -las prueba y cat prueba.
Podemos ver que el archivo ocupa más de 0 bytes y se ha escrito lo que queríamos.

No es necesario cerrar el fichero para que se escriban los datos. Si hacemos flush sal;; y abrimos
el fichero, ahora podemos comprobar que se ha escrito en el fichero, pero éste no está cerrado.

La función de ocaml print_endline s, ya tiene incorporado el flush.

Hasta ahora solo hemos visto funciones relacionadas con la salida de datos, pero también hay
funciones relacionadas con la entrada de datos. Un ejemplo es input_char;; o open_in;;

La función open_in lee un archivo. Si el archivo existe, lo lee y, sino, salta un error.

Un ejemplo de esto es:

```
close_out sal;; (Porque no habíamos cerrado el archivo todavía)
let ent = open_in "prueba";;
input_char ent;;
```

Esto devuelve una A, ya que la función solamente coge UN char. En este caso el primero.
Si seguimos poniendo input_char ent;; nos devolverá los siguientes datos, es decir, A, B, C, D.

Si ponemos input_line, recorre el archivo hasta que encuentra un salto de línea.

Si ponemos read_line ();; podemos poner una frase en la terminal, que leerá ocaml. (como un scanf)

Vamos a hacer un ejercicio:

Si queremos guardar muchos strings en un archivo podemos hacer una función de la siguiente manera:

```
(*output_string_list : out_channel -> string list -> unit *)

let rec output_string_list sal sl = match sl with
    [] -> ()
  | h::t -> output_string sal (h ^ "\n"); output_string_list sal t;;

output_string_list stdout ["A"; "AB";"ABC"; "FIN"];;
```

Ahora vamos a crear una función que lea un archivo en vez de escribirlo.
Esta función será de tipo:
input_string_list : in_channel -> string list;;

```
let rec input_string_list f =
    try
       input_line f :: input_string_list f
    with End_of_file -> [];;

let f = open_in "prueba";;

(*Dentro del archivo prueba, hay 3 líneas: ABC, ABCDEF y AB*)
```

Esta función no funcionará, porque lo que hace moverse al puntero que señala la línea que se quiere
leer es input_line, que se evalúa después de input_string_list, por lo que se dará un Stack Overflow
Una buena forma de definir esta función será:

```
let rec input_string_list f =
    try
       let s = input_line f in
       s :: input_string_list f
    with End_of_file -> [];;
```

De esta forma se soluciona el problema ya que, al hacer un let in nos aseguramos que el input_line
se ejecute antes de input_string_list.

Si ponemos read_int_opt ();; y escribimos un número nos devolverá el número que hemos escrito. Además,
si escribimos 25x por ejemplo, no nos dará error, sino que devolverá None.

Si ponemos ref;; Nos devolverá 'a -> 'a ref;;

Si ponemos ref 0;; nos devuelve un int ref que contiene un entero (en este caso el 0)

Podemos definir un ref como para cualquier cosa.

```
let i = ref 0;;
```

Para representar el valor de la variable que hay contenida en ref hay que hacer un !i;;

Si hacemos !i +1 nos devolverá un 1.

Con la función (:=) hace que el segundo argumento que se le pase se guarde en el primer elemento
Un ejemplo sería;

```
i := 2+1;;
i := !i +1;;
```

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

Ya hemos visto la función for, asi que ahora toca ver la función while.

while <b> do <e> done (<b>:bool EN TODO CASO)

¡¡¡¡¡¡¡¡¡UNA ANOTACIÓN SOBRE LA FUNCIÓN FOR!!!!!!!!!!!!!!

La función for sería for <i> = <e1> to <e2> do <e> done

En este caso la función iría en orden ascendente, así que no podríamos hacer un for en orden
descendente. Para hacer esto, tenemos que cambiar "to" por "downto"

Ahora vamos a definir la función fact con la función while:

```
let fact n =
    let f = ref 1 in
    let i = ref 1 in
    while !i <= n do
       f := !f * !i;
       i := !i + 1
    done;
    !f;;
```

Ahora vamos a hablar de vectores (IMPORTANTE PARA EL ÚLTIMO EJERCICIO DE LA ÚLTIMA PRÁCTICA)

Un vector se crea de la siguiente manera: [|1;2;3;4|];;

Hay funciones para arrays como en las listas, como por ejemplo:

```
let v = [|1;2;3;4|];;

Array.length v;;
Array.get v 2;;
v.(2);; //Hace lo mismo que Array.get v 2

Array.set v 1 7;; //Cambia lo que está guardado en la posición 2 del array

v.(1) <- 7 //Hace lo mismo que Array.set v 1 7

Array.init 10 (fun i -> i);;
Array.init 10 (fun i -> char_of_int  (64+i));;

let vprod v1 v2 =
    if Array.length v1 = Array.length v2
    then let p = ref 0. in
         for i = 0 to Array.length v1 - 1 do
             p := v1.(i) *. v2.(i) +. !p
         done
    else raise (Invalid_argument "vprod");;
```

Otro dato muy utilizado son las estructuras (structs de c).

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

Para representar los valores de este tipo hay que hacer:

```
let p1 = {nombre = "Pepe"; edad = 32};;
```

Cómo se accede al campo de un registro?

```
p1.nombre;; //Esto devuelve el campo nombre del registro (struct) p1

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

mas_viejo p1;;
```

Sin embargo, al hacer esto no modificamos el valor de p1. Si ponemos p1;; podemos comprobarlo.

A la hora de crear el dato podemos hacer que sus atributos pueden ser modificados. Esto se hace de la siguiente manera:

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

Ahora la edad es mutable.

```
let p1 = {nombre = "Pepe"; edad = 32};;

let mas_viejo p = {nombre = p.nombre; edad = p.edad + 1};; //Al hacer esto seguimos sin modificar nada
```

Para modificar el campo tenemos que hacer: p1.edad <- 20;;

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

Si hacemos mas_viejo p1;; y luego p1;; sigue igual, pero si hacemos:

envejece p1;; y luego p1;; podemos observar que el registro ha cambiado.