Ocaml_21_Notes_Infinite

January 27, 2022

1 OCAML NOTES 21 by InfiniteDuck

2 1.Primitive Types

2.1 1.1 Integer

[ ]:
```
(* Int *) 1;;
```

2.2 1.2 Float

[ ]:
```
(* Float *) 1. +. 1.;;
```

2.3 1.3 Boolean

[ ]:
```
true;;
```

[ ]:
```
false;;
```

Ocaml has "Lazy Evaluation". -> Evaluates a expresion part by part

[39]:
```
(* Logical AND *) true && true;; (* First evaluates `true &&`, then evaluates the
 ↪second part,
                                  as the compiler can't tell what the result will
 ↪be *)

false && true;; (* First evaluates `false &&` and returns false, since the result
 ↪will be "False" no matter what's
                   on the other side of the expresion.*)
```

[39]:
```
- : bool = true
```

[39]:
```
- : bool = false
```

[40]:
```
(* Logical OR *) true || false;; (* Same here *)
```

[40]:
```
- : bool = true
```

Disyunciones “b1” || “b2” equivale: if “b1” then true else “b2” “b1” && “b2” equivale: if “b1” then “b2” else false

if “b” then “e1” else “e2” equivale a: (function true -> “e1” | false-> “e2”)“b”

2.4 1.3.1 Equality Operators

Please check the Ocaml Documentation for more info ocaml.org/api/Stdlib.html

2.4.1 1.3.1.1 Structural Equality (= and <>)

Until we’ve studied the imperative –> Compares Values

[ ]:
```
1 = 1;; (* true *)

"A"="B";; (* false *)

1 <> 2;; (* true *)
```

2.4.2 1.3.1.1 Physical Equality (== and !=) ! =

“Compares Pointers in Memory” **only use == if you really know what you’re doing**

[ ]:
```
let a = "ABC";;
let b = "ABC";;

a = b;;
a == b;;
```

2.5 1.4 Characters

[ ]:
```
'a';;
```

[ ]:
```
char_of_int;;

char_of_int 88;;
```

[ ]:
```
int_of_char;;

int_of_char 'A';;
```

There is no *char_of_string*

[ ]:
```
"abc".[0];;
```

2.6 1.5 String

[ ]:
```
"Hi!";;
```

[ ]:
```
"abc" ^ "def";; (*String Concat*)
```

2.6.1 1.5.1 Converting to string

For three of the primitive types, there are built-in functions:

[ ]:
```
string_of_int;; (*int -> string*)

string_of_int 123;;

(*string_of_int 1.22 --->*) (*Exception: Invalid_argument "string_of_int".*)
```

[ ]:
```
int_of_string;; (*string -> int*)

int_of_string "123";;
```

[ ]:
```
string_of_float;; (* float -> string*)

string_of_float 1.2;;
```

[ ]:
```
float_of_string;; (* string -> float*)

float_of_string "1.223";;
```

[ ]:
```
string_of_bool;; (*boolean -> string*)

string_of_bool true;;
```

[ ]:
```
bool_of_string;;

bool_of_string "true";;
```

Strangely, there is no string_of_char, but the library function String.make can be used to accomplish the same goal

[ ]:
```
String.make;;

String.make 1 'z';;

String.make 2 'z';;
```

3 2. Functions and Let Expressions

Podemos usar tanto o keyword `function` como `fun`.

Values can be given names using `let`.

3.1 2.1 Definition

[ ]:
```
let sqr x = x*x;; (*First Way*)

sqr 2;;
```

[ ]:
```
let square = function x -> sqr x;; (*Second Way*)

square 3;;
```

[ ]:
```
let absf = function (* Returns Absolute Value *)
    x -> if x >= 0. then x else -. x;;


absf (-1.2);;
(*let absf x = if x >= 0. then x else -. x;;*)
```

Remember! `if <b> then <e1> else <e2> ==>(function true -> <e1> | false -> <e2>) <b>`

[ ]:
```
let abs x = (function true -> x | false -> -x) (x > 0);;

abs (-1);;
```

3.2 2.2 Let Expressions DUDA!!!!!!!!!!!!!!!!

Remember! `let <x> = <eL> in <eG> ==>(function <x> -> <eG>) <eL>`

We're binding a value "eL" to the name "x" then using that binding inside another expression, "eG".

[29]:
```
(*Se o definimos estando f en local, en caso de que fagamos moitas chamada
de cada chamada volvese a definir f polo que e moi costoso*)
let abs x =
    let f = function true -> 1 | false -> -1 in f (x>0) * x;;

abs (-100);;
```

[29]:
```
val abs : int -> int = <fun>
```

[29]:
```
- : int = 100
```

[142]:
```
(*Asi non se definiria de cada vez*)
let abs =
    let f = function true -> 1 | false -> -1 in
```

```
    function x -> f (x>0) * x ;;

abs (-12);;
```

[142]:
```
val abs : int -> int = <fun>
```

[142]:
```
- : int = 12
```

[ ]:
```
(* Exemplo *)

abs(2);;
abs(-7);;
```

[128]:
```
(* Asi calculas de cada execucion dospi, aunque sempre vai dar o mismo *)
let circ r =
    let dospi = 4. *. asin 1. in
    dospi *. r;;

(* Ao evaluarse queda: *)

let circ r =
   (function dospi -> dospi *. r) (4. *. asin 1.);;

(* que reescribiendo r para que la función quede explícita sería:*)

let circ = function r -> (function dospi -> dospi *. r) (4. *. asin 1.);;

circ 2.;;
```

[128]:
```
val circ : float -> float = <fun>
```

[128]:
```
val circ : float -> float = <fun>
```

[128]:
```
val circ : float -> float = <fun>
```

[128]:
```
- : float = 12.5663706143591725
```

[131]:
```
(* Asi non se calcularia de cada vez *)
let circ =
    let dospi = 4. *. asin 1. in
    function r -> dospi *. r;;
```

```
(* Ao evaluarse queda *)

let circ =
   (function dospi -> (function r -> dospi *. r)) (4. *. asin 1.);;

(* Y estamos definiendo circ como la aplicación de la función (function dospi␣
 ↪-> ...) a (4. *. asin 1. ),
así que se evaluaría esa aplicación,para eso se evalúa primero el parámetro:*)

let circ =
   (function dospi -> (function r -> dospi *. r)) 6.2832;;


(* Y se sustituye en el cuerpo de la función: *)

let circ = function r -> 6.2832 *. r;;

circ 2.;;
```

```
[131]: val circ : float -> float = <fun>
```


```
[131]: val circ : float -> float = <fun>
```

```
[131]: val circ : float -> float = <fun>
```

```
[131]: val circ : float -> float = <fun>
```


```
[131]: - : float = 12.5664
```

```
[14]: let f = function x -> (function y -> x + y);;

f 1 2;;
```

```
[14]: val f : int -> int -> int = <fun>
```

```
[14]: - : int = 3
```

3.3 2.3 Currying

(Operator) Operands

[24]:
```
(+);;

(+) 2 3;; (* Sum *)

(^) "Hi!" "Bye!";; (* Concat *)

(<=) 2 3;; (* Compare *)

(* VALIDO PARA TODAS AS OPERACIONS *)
(* ATENCION *) (* PARA A MULTIPLICACION DEIXASE UN ESPACIO ENTRE PARENTESIS E␣
 ↪ASTERISCO PARA NON CONFUNDIR CO COMENTARIO ( * ) *)
```

[24]:
```
- : int -> int -> int = <fun>
```

[24]:
```
- : int = 5
```

[24]:
```
- : string = "Hi!Bye!"
```

[24]:
```
- : bool = true
```

[23]:
```
let succ = (+) 1;;
succ 3;;


let op = (-) 10;;
op 4;;
```

[23]:
```
val succ : int -> int = <fun>
```

[23]:
```
- : int = 4
```

[23]:
```
val op : int -> int = <fun>
```

[23]:
```
- : int = 6
```

3.3.1 2.3.1 Producto Cartesiano

[21]:
```
fst;;

snd;;

let suma = function p -> fst p + snd p;;

suma(2,3);;
```

[21]:
```
- : 'a * 'b -> 'a = <fun>
```

[21]:
```
- : 'a * 'b -> 'b = <fun>
```

[21]:
```
val suma : int * int -> int = <fun>
```

[21]:
```
- : int = 5
```

[26]:
```
let fstt = function x,_,_ -> x;;

fstt (true,0,"Hi!");;
```

[26]:
```
val fstt : 'a * 'b * 'c -> 'a = <fun>
```

[26]:
```
- : bool = true
```

[27]:
```
let p = (true,0),"What!";;

let x,y = p;;
```

[27]:
```
val p : (bool * int) * string = ((true, 0), "What!")
```

[27]:
```
val x : bool * int = (true, 0)
val y : string = "What!"
```

3.4 2.4 Recursive

IMPORTANTE:

-Condicionales: (if “b” then “e1” else “e2”) e1 y e2 tenen que ser do mismo tipo.

REC: Con “rec” ocaml entende que a funcion vaise a chamar a si mesma.

[151]:
```
let rec fact = function (* Devolve factorial *)
    n -> if n = 0 then 1
    else n * fact (n-1);;

fact 2;;
```

[151]:
```
val fact : int -> int = <fun>
```

[151]:
```
- : int = 2
```

3.5 2.5 Tail recursion (Pag 53 Real World Ocaml)

A recursive call in tail position does not need a new stack frame. It can just reuse the existing stack frame. That's because there's nothing left of use in the existing stack frame! There's no computation left to be done, so none of the local variables, or next instruction to execute, etc. matter any more.

[9]:
```
(*Transformar a factorial en Tail Recursive*)
let fact n = (*f -> accumulator, i -> contador ata destino*)
    let rec aux (i,f) =
        if i = n then f
        else aux (i+1,f * (i+1))
    in aux (0,1);;

fact 10;;
```

[9]:
```
val fact : int -> int = <fun>
```

[9]:
```
- : int = 3628800
```

[139]:
```
(*Dividir usando a suma e a resta -> quo*) (*Non Recursivo final, pode crear␣
↪stack overflow*)
let rec quo x y = (*"Precondicion" x>=0 ; y>0*)
    if x < y then 0
    else 1 + quo (x-y) y;;

quo 10 2;; (*Non Tail Recursive!!*)
```

[139]:
```
val quo : int -> int -> int = <fun>
```

[139]:
```
- : int = 5
```

[147]:
```
(* Quo -> Tail Recursive *)
let quo x y = (* "Precondicion" x>=0 ; y>0 *)
    let rec aux x y s =
        if x < y then s
        else aux (x-y) y (s+1)
    in aux x y 0;;

quo 10 2;; (* Tail Recursive!! *)
```

[147]:
```
val quo : int -> int -> int = <fun>
```

[147]:
```
- : int = 5
```

[31]:
```
(* Funcion que me da o resto *) (* Recursividade final/terminal, non deixa contas␣
 ↪pendientes. Non crea stack overflow *)
let rec rem x y = (* "Precondicion" x>=0 ; y>0 *)
    if x < y then x
    else rem (x-y) y;;

rem 10 3;;
```

[31]:
```
val rem : int -> int -> int = <fun>
```

[31]:
```
- : int = 1
```

[32]:
```
(* Opcion 1 --> let div x y = quo x y, rem x y;;*) (* Combino as duas funcions␣
 ↪anteriores *)
let rec div x y =
    if x<y then 0, x
    else let q, r = div (x-y) y in
         1 + q, r ;;

div 10 2;;
```

[32]:
```
val div : int -> int -> int * int = <fun>
```

[32]:
```
- : int * int = (5, 0)
```

[36]:
```
Sys.time();;  (* Tempo de CPU que leva consumido OCAMl executando operacions *)
```

[36]:
```
- : float = 0.270862
```

3.5.1 2.5.1 Fibonacci

[37]:
```
let k =  (1. +. sqrt 5.)/. 2.;; (*Para saber canto tempo levaria calcular o fib␣
 ↪de un num*)
```

[37]:
```
val k : float = 1.6180339887498949
```

[39]:
```
let rec fib n = (*n >= 0*)
    if n>1 then fib (n-1) + (n-2)
    else n;;

fib 4;;
```

[39]:
```
val fib : int -> int = <fun>
```

[39]:
```
- : int = 4
```

[13]:
```
let fib n =  (*Alternativa mais eficiente ao de arriba*)
  let rec fib2 = function
    0 -> 0,1
  | 1 -> 1, 0
  | n -> let f1, f2 = fib2 (n-1) in
         f1 + f2, f1
  in fst(fib2 n);;

fib 0;;
fib 1;;
fib 2;;
fib 3;;
fib 4;;
fib 10;;
```

[13]:
```
val fib : int -> int = <fun>
```

[13]:
```
- : int = 0
```

[13]:
```
- : int = 1
```

[13]:
```
- : int = 1
```

[13]:
```
- : int = 2
```

[13]:
```
- : int = 3
```

[13]:
```
- : int = 55
```

[16]:
```
(* Mellor implementacion *)
let fib n = (*Mecanismo da tabla "cruzada" visto en Algoritmos*)
    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);; (*O anterior ao de 0 e 1 *)

fib 300_000;; (*Ahora xa nn hai stackoverflow*)
```

[16]:
```
val fib : int -> int = <fun>
```

[16]:
```
- : int = -199128287061131648
```

4 3.LISTAS

[47]:
```
[1;2;3;4];; (*Tipo LISTA*) (*Secuencias finitas de ints*)
```

[47]:
```
- : int list = [1; 2; 3; 4]
```

[46]:
```
let l = ['a';'e';'i';'o';'u'];; (*Char List*)
```

[46]:
```
val l : char list = ['a'; 'e'; 'i'; 'o'; 'u']
```

[48]:
```
[(1,2);(2,3)];; (*Lista de int*int (cartesiano)*)
(*(int*int) list e distinto de int*int list*)
1,[2;3];;
```

[48]:
```
- : (int * int) list = [(1, 2); (2, 3)]
```

[48]:
```
- : int * int list = (1, [2; 3])
```

[49]:
```
(*Hai infinitos tipos de lista*)

[];; (*Lista de tipo polimorfica*)
```

[49]:
```
- : 'a list = []
```

4.1 3.1 List Functions

Funcions modulo list

4.1.1 3.1.1 Length

[51]:
```
List.length;;
List.length l;; (*Devolve num elementos de unha lista*)
```

[51]:
```
- : 'a list -> int = <fun>
```

[51]:
```
- : int = 5
```

[100]:
```
let rec list_length l =  (*Lonx e 1 mais que a cola , si esta vacia e 0*)␣
 ↪(*Not- Tail_Recursive*)
  if l = [] then 0
  else 1 + list_length(List.tl l);;

list_length l;;
```

[100]:
```
val list_length : 'a list -> int = <fun>
```

[100]:
```
- : int = 5
```

[472]:
```
let length l =  (* Tail Recursive *)
  let rec aux l count = match l with
    [] -> count
  | h::t -> aux t (count+1)

  in aux l 0;;
```

[472]:
```
val length : 'a list -> int = <fun>
```

4.1.2 3.1.2 Head (hd)

[53]:
```
List.hd;;
List.hd l;; (*Devolve o primeiro elemento da lista*)
```

[53]:
```
- : 'a list -> 'a = <fun>
```

```
[53]: - : char = 'a'
```

```
[470]: let rec hd = function
           [] -> raise(Failure "hd")
         | h::_ -> h;;
```

```
[470]: val hd : 'a list -> 'a = <fun>
```

4.1.3 3.1.3 Tail (tl)

```
[55]: List.tl;;
      List.tl l;; (*Devolve a lista que lle pasaches pero sin o primeiro elemento*)

      List.tl [1;2];; (*Devolve a lista pero sin o 1*)

      List.tl [2];; (*Devolve vacia *)
```

```
[55]: - : 'a list -> 'a list = <fun>
```

```
[55]: - : char list = ['e'; 'i'; 'o'; 'u']
```

```
[55]: - : int list = [2]
```

```
[55]: - : int list = []
```

```
[471]: let rec tl = function
           [] -> raise(Failure "tl")
         | h::t -> t;;
```

```
[471]: val tl : 'a list -> 'a list = <fun>
```

4.1.4 3.1.4 Last

```
[62]: (* last: 'a list -> 'a *)
      (*let rec last l =  (*Ao usar List.length e moi costoso*)
          if List.length l = 1 then List.hd l
          else last (List.tl l);; (*Non quedan cuentas pendientes, non crea stack␣
       ↪overflow*)
      *)
      let rec last l =  (*Ao usar List.length e moi costoso*) (*Tail Recursive*)
          if List.tl l = [] then List.hd l
```

```
    else last (List.tl l);; (*Non quedan cuentas pendientes, non crea stack␣
 ↪overflow*)

last l;;
```

[62]: val last : 'a list -> 'a = <fun>

[62]: - : char = 'u'

4.1.5 3.1.5 @ (“Concat”)

[64]:
```
(@);; (*Usar @ vale para unir 2 listas *)

(*Vale para concatenar listas equivalendo a List.append*)
let l = [1;2;3] @ [4;5;6]

(*O append e moi costoso ,como o List.length*)
```

[64]: - : 'a list -> 'a list -> 'a list = <fun>

[64]: val l : int list = [1; 2; 3; 4; 5; 6]

4.1.6 3.1.6 Reverse

[71]:
```
List.rev ;;

List.rev l ;; (*Imprime a lista a inversa*)
l;;
```

[71]: - : 'a list -> 'a list = <fun>

[71]: - : int list = [6; 5; 4; 3; 2; 1]

[71]: - : int list = [1; 2; 3; 4; 5; 6]

4.1.7 3.1.7 Nth

[67]:
```
List.nth;;

List.nth l 2;; (*Se aplica a una lista y a un entero e devolve un elemento do␣
 ↪tipo lista.*)
```

```
                (* Devolve o elemento que ocupa esa posicion. Empezase a contar␣
 ↪por 0 *)
```

```
- : 'a list -> int -> 'a = <fun>
```

```
- : int = 3
```

```
(* Outra forma de definir Last. Usando nth *)
let last l = List.nth l (List.length l -1);;
```

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

4.1.8 3.1.8 Map

```
List.map;; (* Aplicar unha funcion a cada elemnto da lista. Ex: De int list a␣
 ↪float list *)

List.map abs [0;-1;1;2;-2];;

List.map float_of_int [0;-1;1;2;-2];;

List.map int_of_char ['a';'e';'i';'o';'U'];;
```

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

```
- : int list = [0; 1; 1; 2; 2]
```

```
- : float list = [0.; -1.; 1.; 2.; -2.]
```

```
- : int list = [97; 101; 105; 111; 85]
```

4.1.9 3.1.9 Filter

```
List.filter;; (* Aplicase aos predicados (funcions de booleanos) *)

List.filter (function n -> n > 0) [0;-1;1;2;-2];; (* Filtra a lista deixando solo␣
 ↪os elemento que dean true *)
```

```
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
[74]: - : int list = [1; 2]
```

```
[76]: List.mem;; (* Devolve un bool en conforme un elemento pertence a lista ou no *)

      List.mem 0 [-1;1;2;-2];;
      List.mem 0 [0;-1;1;2;-2];;
```

```
[76]: - : 'a -> 'a list -> bool = <fun>
```

```
[76]: - : bool = false
```

```
[76]: - : bool = true
```

```
[77]: List.exists;; (*Aplicase a un predicado, como o filter pero devolve un bool si␣
       ↪hai un elemento que cumpla o predicado*)

      List.exists (function n -> n>0) [0;-1;1;2;-2];;
      List.exists (function n -> n>0) [];;
```

```
[77]: - : ('a -> bool) -> 'a list -> bool = <fun>
```

```
[77]: - : bool = true
```

```
[77]: - : bool = false
```

```
[79]: List.for_all;; (*Aplicase a predicado. Como o filter pero en funcion de si todos␣
       ↪os elementos cumplen o predicado*)

      List.for_all (function n -> n>0) [0;-1;1;2;-2];;
      List.for_all (function n -> n>0) [];; (**for_all na listas vacias sempre da␣
       ↪TRUE*)
```

```
[79]: - : ('a -> bool) -> 'a list -> bool = <fun>
```

```
[79]: - : bool = false
```

```
[79]: - : bool = true
```

[82]:
```
List.find;; (*Collemos predicado e devolvemos o primeiro elemento que o cumple*)

List.find (function n -> n>0) [0;-1;1;2];;
(*List.find (function n -> n>0) [-1];;*) (*Se non encontra devolve␣
 ↪excepcion*) (*Exception: Not_found.*)
```

[82]:
```
- : ('a -> bool) -> 'a list -> 'a = <fun>
```

[82]:
```
- : int = 1
```

[85]:
```
List.init;; (*Cando se aplica a enteiro e function, devolve unha lista con␣
 ↪tantos elemento como indique o enteiro,
            aplicase a funcion*)

List.init 5 (float_of_int);; (*Devolve unha lista cos elementos do que valia a␣
 ↪funcion nesa posicion *)

List.init 10 (function i -> char_of_int (65+i));;
```

[85]:
```
- : int -> (int -> 'a) -> 'a list = <fun>
```

[85]:
```
- : float list = [0.; 1.; 2.; 3.; 4.]
```

[85]:
```
- : char list = ['A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J']
```

[102]:
```
List.iter ;;(*Recorre a lista aplicando unha funcion a unha lista. Fai o que ti␣
 ↪queiras con eso, podelo mostrar, gardar ....*)
```

[102]:
```
- : ('a -> unit) -> 'a list -> unit = <fun>
```

4.1.10 Outra forma de definir Listas : :

Outra forma de representar listas `cons` : : **OLLO: cons e asociativo pola dereita**

`(<h>::<t>): x list` representa a unha lista que ten como cabeza h e cola t

[97]:
```
true :: [true];;

1::2::3::4::[];;

let l = ['a';'e';'i';'o';'u'];;

(*let h::t = l;;*) (*Si l fose unha lista vacia daria error de ejecucion*)
```

```
                (*h queda asociado a 'a' e t a lista eiou*)
                (*Warning 8 [partial-match]: this pattern-matching is not␣
 ↪exhaustive.*)

(*let x::y = t;;*)
```

[97]: `- : bool list = [true; true]`

[97]: `- : int list = [1; 2; 3; 4]`

[97]: `val l : char list = ['a'; 'e'; 'i'; 'o'; 'u']`

4.1.11 Redefinimos algunhas funcions usando o `cons::`

[107]:
```
let list_1 = [1;2;3;4;5;2];;
let list_2 = [3;4;5;6;5];;
```

[107]: `val list_1 : int list = [1; 2; 3; 4; 5; 2]`

[107]: `val list_2 : int list = [3; 4; 5; 6; 5]`

[103]:
```
let rec list_length = function (*Lonx e 1 mais que a cola , si esta vacia e 0*)
   [] -> 0
  |h::t -> 1 + list_length t;;

list_length l;;
```

[103]: `val list_length : 'a list -> int = <fun>`

[103]: `- : int = 5`

[105]:
```
let rec list_last = function
  [] -> raise (Failure "list_last")
  | h::[] -> h
  | h::t -> list_last t;;  (*cola t no vacia*)

list_last l;;
```

[105]: `val list_last : 'a list -> 'a = <fun>`

[105]:
```
- : char = 'u'
```

[110]:
```
(* Concatenacion *) (* Equivale a append: 'a list -> ('a list -> 'a list) *)
let rec append = function
    [] -> (function l -> l)
  | h::t -> (function l  -> h:: append t l);;

append list_1 list_2;;
```

[110]:
```
val append : 'a list -> 'a list -> 'a list = <fun>
```

[110]:
```
- : int list = [1; 2; 3; 4; 5; 2; 3; 4; 5; 6; 5]
```

[111]:
```
(* Usando match_with e o mismo que o de arriba e entendese moito mellor *)
let rec append l1 l2 = match l1 with
    [] -> l2 (* Si l1 vacia *)
  | h::t -> h :: append t l2;;

append list_1 list_2;;
```

[111]:
```
val append : 'a list -> 'a list -> 'a list = <fun>
```

[111]:
```
- : int list = [1; 2; 3; 4; 5; 2; 3; 4; 5; 6; 5]
```

List Compare

[ ]:
```
compare;; (* Compara 2 elementos Como en C si o primeiro elemento e menor devolve␣
 ↪-1 se e igual devolve 0 se e maior 1 *)

compare 1 2;;
```

[113]:
```
(* Aplicado a lista *)
List.compare_lengths;; (* Compara tamano de listas *)
```

[113]:
```
- : 'a list -> 'b list -> int = <fun>
```

[117]:
```
(* Implementacion manual *)

(* let compare_length l1 l2 =  Non eficiente
    compare (List.length l1) (List.length l2) *)
```

```
let rec compare_lengths = function
    [] -> (function [] -> 0 (*Lista vacia*)
                    | _ -> -1 ) (*Calquera lista non vacia. A primeira e vacia*)
  | h::t -> (function [] -> 1
                      | h2::t2 ->  compare_lengths t t2);; (*Como as 2 listas␣
↪tenen cabeza e cola podemos comparar solo a cola
                                                          e xa vale asi a que␣
↪antes se acabe sabemos que e a mais vacia*)
```

[117]:
```
val compare_lengths : 'a list -> 'b list -> int = <fun>
```

[120]:
```
let l1 = ['a';'e';'i';'o';'u'];;
let l2 = ['a';'e';'i'];;

compare_lengths  l1 l2;;
```

[120]:
```
val l1 : char list = ['a'; 'e'; 'i'; 'o'; 'u']
```

[120]:
```
val l2 : char list = ['a'; 'e'; 'i']
```

[120]:
```
- : int = 1
```

[122]:
```
(*Rescribimos a function pero con match with*)

let rec compare_lengths l1 l2 = match l1,l2 with
    [],[] -> 0
  | [],_ -> -1 (*Ou [],_::_ vale igual*)
  | _::_,[] -> 1
  | _::t1,_::t2 -> compare_lengths t1 t2;;

compare_lengths  l1 l2;;
```

[122]:
```
val compare_lengths : 'a list -> 'b list -> int = <fun>
```

[122]:
```
- : int = 1
```

List Mem

[124]:
```
(*Di si aparece o elemento na lista*)

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

```
mem 'a' l1;;
```

[124]:
```
val mem : 'a -> 'a list -> bool = <fun>
```

[124]:
```
- : bool = true
```

5 4.Tail recursion 2

Sempre hai que intentar que as funcions recursivas sexan terminales, para evitar stackoverflow en casos moi grandes

[138]:
```
(* Suma length *)

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

let length l = suma_length 0 l;;

(* Integrando todo en unha funcion *)

let length l =
    let rec aux s = function
    [] -> s
   | _::t -> suma_length (s+1) t
   in aux 0 l;;

(* l1 = ['a';'e';'i';'o';'u'] *)

length l1;;
```

[138]:
```
val suma_length : int -> 'a list -> int = <fun>
```

[138]:
```
val length : 'a list -> int = <fun>
```

[138]:
```
val length : 'a list -> int = <fun>
```

[138]:
```
- : int = 5
```

Obter valor max de unha lista

[ ]:
```
(*Funcion que de unha lista me dea o valor maximo*)
```

[18]:
```
let rec lmax_ntail = function (* NOT TAIL RECURSIVE *)
    (*Non defino o caso lista vacia e asi da error*)
    h::[] -> h
  | h::t -> max h (lmax_ntail t);; (*Uso a funcion max*) (*Pode ser ou a cabeza␣
↪ou o maximo da cola.
                            Collo o maximo da cola e comprbo si a cabeza e maior␣
↪que ese maximo*)
```

```
File "[18]", lines 1-4, characters 21-32:
1 | ...function (* NOT TAIL RECURSIVE *)
2 |     (*Non defino o caso lista vacia e asi da error*)
3 |     h::[] -> h
4 |   | h::t -> max h (lmax_ntail
t)...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
```

[18]:
```
val lmax_ntail : 'a list -> 'a = <fun>
```

[24]:
```
let lmax_tail l = (*Tail Recursive*) (*Comparas coas cabezas*) (*En caso de␣
↪lista vacia queremos que dea error*)
   let rec aux m = function
       [] -> m
     | h::t -> aux (max m h) t
   in aux (List.hd l) (List.tl l);;


lmax_tail [1;2;5;0;3];;
```

[24]:
```
val lmax_tail : 'a list -> 'a = <fun>
```

[24]:
```
- : int = 5
```

[48]:
```
let rec lmax = function  (*Tail Recursive sin usar funcion auxiliar*)
  [] -> raise(Failure "max")
  | h::[] -> h (*Lista con solo cabeza*)
  | h1::h2::t -> lmax (max h1 h2::t);; (*Buscamos simplificar a lista, enton␣
↪collo e descarto o minimo dos 2 primeiros elementos.
                                    Asi teno unha lista mais pequena e o␣
↪maximo segue a ser o mesmo*)

let l = [1;2;3;4;5;6;7;8;9;10];;
```

```
lmax l;;

lmax_tail l;;
```

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

```
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

```
- : int = 10
```

```
- : int = 10
```

```
let l1 = [1;2;3];;
let l2 = [4;5];;
```

```
val l1 : int list = [1; 2; 3]
```

```
val l2 : int list = [4; 5]
```

Rev_Append Terminal

```
let rec rev_append l1 l2 = match l1 with  (*O(n)*)
    [] -> l2
  | h::t -> rev_append t (h::l2);; (*Reunir elementos*)(*Concatena a inversa da primeira coa segunda*)

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

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

```
- : int list = [3; 2; 1; 4; 5]
```

```
rev_append l1 [];; (*Si facemos rev_Append con dunha lista con unha vacia invertimos a lista*)
```

```
- : int list = [3; 2; 1]
```

Reverse Terminal

[32]:
```
let rev l = rev_append l1 [];;

rev l1;;
```

[32]:
```
val rev : 'a -> int list = <fun>
```

[32]:
```
- : int list = [3; 2; 1]
```

Append Terminal

[33]:
```
(*Podese facer como o rev_append do rev(l1) e l2*)
let tail_append l1 l2 =
    rev_append (rev l1) l2;;

tail_append [1;2;3] [4;5;6];;
```

[33]:
```
val tail_append : 'a -> int list -> int list = <fun>
```

[33]:
```
- : int list = [1; 2; 3; 4; 5; 6]
```

Fold Left e Right

[36]:
```
let rec fold_left op e l = match l with   (* Tail Recursive *)
    [] -> e
  | h::t -> fold_left op (op e h) t;;

fold_left (+) 3 [2;1;4];;  (* 3+2= 5 , 5+1 = 6, 6+4= 10*) (*(((3+2)+1)+4)*)
```

[36]:
```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

[36]:
```
- : int = 10
```

[37]:
```
let rec fold_right op l e = match l with  (*NON RECURSIVA TERMINAL*)
    [] -> e
  | h :: t -> op h (fold_right op t e);;

fold_right (+) [1;2;3] 1;;
```

[37]:
```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

[37]:
```
- : int = 7
```

[41]:
```
let rec sumList = function (*NOT Tail Recursive*) (* Suma todos os elementos de␣
↪unha lista *)
    [] -> 0
  | h::t -> h + sumList t;;


let sumList l = fold_left (+) 0 l;;   (*Tail Recursive*)
```

[41]:
```
val sumList : int list -> int = <fun>
```

[41]:
```
val sumList : int list -> int = <fun>
```

[43]:
```
let length l = fold_left (function s -> function _ -> s + 1) 0 l;; (* Tail␣
↪Recursive *)
                                                                    (* s ->␣
↪contador, l -> lista restante*)

(*function x -> function y -> e *) (*==*) (*fun x y -> e*)
let length l = fold_left (fun s _ -> s + 1) 0 l;;(*OUTRA FORMA de escribir*)
```

[43]:
```
val length : 'a list -> int = <fun>
```

[43]:
```
val length : 'a list -> int = <fun>
```

[52]:
```
let lmax l = match l with    (*Tail Recursive*)
    [] -> raise(Failure "max")
    | h::t -> fold_left max h t ;;

lmax l;;
```

[52]:
```
val lmax : 'a list -> 'a = <fun>
```

[52]:
```
- : int = 10
```

[55]:
```
let last l = match l with  (* Tail Recursive *)
        [] -> raise(Failure "max")
    | h::t -> fold_left (fun _ y -> y ) h t;; (*Colle dous valores e devolve␣
↪sempre o segundo, vai asocianto sempre
                                                a tail (t) con y *)
last l;;
```

```
[55]: val last : 'a list -> 'a = <fun>
```

```
[55]: - : int = 10
```

```
[57]: let rev l = fold_left (fun l x-> x::l) [] l;; (*Outra forma de definir o rev*)␣
      ↪(*Tail Recursive*)

      rev [1;2;3;4];;
```

```
[57]: val rev : 'a list -> 'a list = <fun>
```

```
[57]: - : int list = [4; 3; 2; 1]
```

```
[63]: let for_all pred lis = match lis with (*List.for_all*) (*Da true si ao aplicar␣
      ↪unha funcion en todos os da lista da true*) (*EXERCICIO*)
          [] -> true
        | h::t ->List.fold_left (fun a b -> pred b && a) true lis  (*Pouco eficiente,␣
      ↪porque se hai un que non se cumple seguimos probando,
                                                          cando xa sabemos que vai␣
      ↪dar false*)
          ;;

      let rec for_all f l = match l with (*Tail Recursive *) (*Mais eficiente, non␣
      ↪recorremos toda a lista en caso de false*)
        [] -> true
      | h::t -> if f h then for_all f t else false;;

      for_all (function n -> n>0) [0;-1;1;2;-2];;
```

```
[63]: val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

```
[63]: val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

```
[63]: - : bool = false
```

```
[64]: let rec exists f l = match l with
        [] -> false
      | h::t -> if f h then true else exists f t;;

      exists (function n -> n>0) [0;-1;1;2;-2];;
```

```
[64]: val exists : ('a -> bool) -> 'a list -> bool = <fun>

[64]: - : bool = true
```

6 5. Algoritmos Ordenacion

```
[ ]: let big_list = List.init 1_000_000 abs;;  (* Inicializo lista *)
```

```
[66]: let rec sorted = function  (* Comproba ordenacion en orden ascendente *)
          h1::h2::t -> h1 <= h2 && sorted (h2::t)
        | _ -> true;;

      sorted big_list;;

[66]: val sorted : 'a list -> bool = <fun>

[66]: - : bool = true
```

6.0.1 5.1 Not Tail Recursive

```
[76]: let rec insert f x = function (* Not Tail Recursive *) (* Inserta mantendo certo␣
       ↪orden *)
          [] -> [x]
        | h::t -> if f x h  then x::h::t
                  else h :: (insert f x t);;

[76]: val insert : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

```
[77]: let rec i_sort f = function  (* Ordenar lista *) (* Not Tail Recursive *)
          [] -> []
        | h::t -> insert f h (i_sort f t);;

      i_sort (>=) [4;8;1;6];;

[77]: val i_sort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

[77]: - : int list = [8; 6; 4; 1]
```

Funcion para cronometrar o tempo de ordenacion

[71]:
```
let crono f x =
    let t = Sys.time () in
    let _ = f x in
    Sys.time () -. t ;;
```

[71]:
```
val crono : ('a -> 'b) -> 'a -> float = <fun>
```

[73]:
```
(*Random.int n   saca pseudoaleatoriamente n numeros int*)
(*List.init 10_000 (function _ -> Random.int 1_000_000);;*)

crono i_sort (List.init 10_000 (function _ -> Random.int 1_000_000));;␣
 ↪(*Cronometramos o tempo que lle leva ordenar esa lista*)
```

[73]:
```
- : float = 1.06143699999999974
```

[81]:
```
let f s1 s2 = (*Tamen se pode aplicar a Strings, etc ...*)
    let l1 = String.length s1 in
    let l2 = String.length s2 in
    if l1 = l2 then s1 <= s2 else l1<l2;;

i_sort f ["gopher";"duck";"fennec"];;
```

[81]:
```
val f : string -> string -> bool = <fun>
```

[81]:
```
- : string list = ["duck"; "fennec"; "gopher"]
```

6.0.2 5.2 Tail Recursive

[83]:
```
let insert' f x l =  (*Insert terminal*)
    let rec aux (before,after) = match after with
        [] -> List.rev (x::before)   (*Cando elemento a insertar e o mais␣
 ↪grande*)
      | h::t -> if f x h (* Comprobo ordenacion, formato curry*)
                then List.rev_append before (x::after)
                else aux (h::before, t)
    in
        aux([],l)
    ;;

(*let big_list = List.init 300_000 abs;;*)  (*Inicializo lista*)

(*insert' (<=) 300_000 big_list;;*) (*A;adeo pero non se ve, se non daria␣
 ↪overflow*)
```

```
[83]: val insert' : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

Isort Terminal

```
[86]: let i_sort' f l =  (* Tail Recursive *)
          let rec aux ordenados = function
              [] -> ordenados
            | h::t -> aux (insert' f h ordenados) t
          in aux [] l
          ;;

      i_sort' (<=) [5;2;7;9;-1];;
```

```
[86]: val i_sort' : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

```
[86]: - : int list = [-1; 2; 5; 7; 9]
```

6.0.3 Comparacion Tempos Ordenacion Terminal / Non Terminal

```
[ ]: let l1 = List.init 10_000 (function _ -> Random.int 1_000_000);;
     let l2 = List.init 20_000 (function _ -> Random.int 1_000_000);;
     let l4 = List.init 40_000 (function _ -> Random.int 1_000_000);;
```

```
[88]: crono (i_sort' (<=)) l1;;
```

```
[88]: - : float = 1.83158200000000093
```

```
[89]: crono (i_sort (<=)) l1;;
```

```
[89]: - : float = 1.22307400000000044
```

```
[90]: crono (i_sort' (<=)) l2;;
```

```
[90]: - : float = 8.407716
```

```
[91]: crono (i_sort (<=)) l2;;
```

```
[91]: - : float = 5.380357
```

6.0.4 5.3 Ordenacion por Fusion (+ info na P10)

5.3.1 Divide (Not Tail Recursive)

[95]:
```
(* divide: 'a list -> 'a list * 'a list *) (* Dividimos os elementos da lista,␣
↪un para cada lado*)

let rec divide = function (*Not Tail Recursive*)
    h1::h2 ::t -> let t1, t2 = divide t in  (*Empezamos repartindo os elemento␣
↪da cola *)
                                            (*Minimo ten que ter 2 elementos␣
↪para mandar un para cada lado*)
                  h1::t1, h2::t2   (*Ahora anadimos os que falta, que son os dous␣
↪iniciales*)

  | l -> l, [];;  (*Se non ten 2 elementos metemos o que temos nunha lista e a␣
↪outra vacia*)


divide ["a";"e";"i";"o";"u"];;
```

[95]:
```
val divide : 'a list -> 'a list * 'a list = <fun>
```

[95]:
```
- : string list * string list = (["a"; "i"; "u"], ["e"; "o"])
```

(Tail Recursive)

[100]:
```
let divide' l =  (* Inserccion Par -> Derecha , Impar -> Izquierda*) (*Tail␣
↪Recursive*)
   let rec aux l left right pos = match l with
       [] -> (List.rev_append right [],List.rev_append left [])
     | h::t -> if pos mod 2 == 0 then aux t left (h::right) (pos+1) else aux t␣
↪(h::left) right (pos+1)
   in aux l [] [] 0;;

divide' ["a";"e";"i";"o";"u"];;
```

[100]:
```
val divide' : 'a list -> 'a list * 'a list = <fun>
```

[100]:
```
- : string list * string list = (["a"; "i"; "u"], ["e"; "o"])
```

5.3.2 Merge (Not Tail Recursive)

[99]:
```
let rec merge l1 l2 = match l1,l2 with (* Not Tail Recursive *)
    [],l | l,[] ->  l    (*Podense unir casos asi, como cando no switch non pos␣
↪break para usar o mismo para multiples casos*)
```

```
   | h1::t1, h2::t2 -> if h1<= h2 then h1:: merge t1 l2
                       else h2:: merge l1 t2;;

merge [1;3;10;100] [2;4;6;1000];;
```

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

```
- : int list = [1; 2; 3; 4; 6; 10; 100; 1000]
```

(Tail Recursive)

```
let merge' f (l,l1) = (*Tail Recursive*)
    let rec aux f l l1 laux = match l,l1 with
        [],a | a,[] -> List.rev_append laux  a
    | h1::t1,h2::t2 -> if f h1 h2 then aux f t1 (h2::t2) (h1::laux)
                       else aux f (h1::t1) t2 (h2::laux)
    in aux f l l1 [];;
```

```
val merge' : ('a -> 'a -> bool) -> 'a list * 'a list -> 'a list = <fun>
```

5.3.3 M_Sort (Tail Recursive)

```
(*merge_sort: 'a list -> a' list *)

let rec m_sort f l = match l with
    [] -> []
  | h::[] -> [h] (*ou [h] -> [h]*) (*Temos que meter este caso para listas con␣
 ↪un unico elemento. *)
  | l  ->  let l1,l2 = divide' l in
           merge' f ((m_sort f l1),(m_sort f l2));;


m_sort (>=) [9;3;5;0;2];;

(*Nesta non me importa tanto a recursividade terminal, xa que por exemplo para␣
 ↪unha lista de 1000 elementos partimos en 2 de 500...
Para chegar a 0 temos que dividir unhas 10 veces a lista de 1000 elementos  ==␣
 ↪log2(10) son moi poucas, polo que non hai problema
Para 1_000_000 \log _2\left(1000000\right) solo necesitamos 20 niveles*)
```

```
val m_sort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

```
- : int list = [9; 5; 3; 2; 0]
```

Random List Init

[110]:
```
Random.int 100_000;;
```

[110]:
```
- : int = 35178
```

[2]:
```
let randomList n = List.init n (fun _ -> Random.int 1_000_000);;

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


crono randomList 1_000_000;;
```

[2]:
```
val randomList : int -> int list = <fun>
```

[2]:
```
val crono : ('a -> 'b) -> 'a -> float = <fun>
```

[2]:
```
- : float = 0.316580999999999946
```

7 6. Options (Some,None..)

Imagine an Option being like a box. That box is either **Empty** (`None`) or **there's something in it** (`Some`) of type `'a`, it could be a Number, a Char ..... `type 'a option = None | Some of 'a`

[11]:
```
None;;
```

[11]:
```
- : 'a option = None
```

[9]:
```
Some 1;;
```

[9]:
```
- : int option = Some 1
```

7.1 6.1 Examples

[21]:
```
let div x y = (* Divide without exposing to Exceptions *)
    if y = 0 then None
    else Some (x / y);;

div 7 0;;
```

```
div 7 1;;
```

```
[21]: val div : int -> int -> int option = <fun>

[21]: - : int option = None

[21]: - : int option = Some 7
```

Example with `try with`.

```
[23]: let div x y = (* Outra forma de Division, pero con try with. *)
          try
              Some (x / y)
          with
              Division_by_zero -> None;;

      div 7 0;;

      div 7 1;;
```

```
[23]: val div : int -> int -> int option = <fun>

[23]: - : int option = None

[23]: - : int option = Some 7
```

```
[28]: let hd' l =
          try Some (List.hd l)
          with Failure _ -> None;; (* Use _ as a wildcard *)

      hd' [];; (* The original `List.hd` would return <Exception: Failure "hd".> *)

      hd' [2;3];;
```

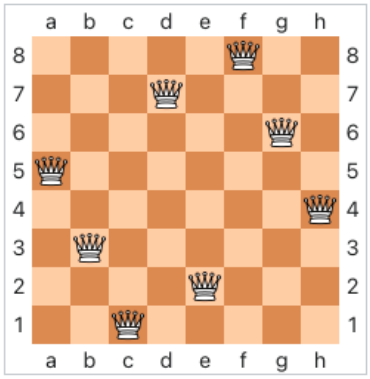
```
[28]: val hd' : 'a list -> 'a option = <fun>

[28]: - : 'a option = None

[28]: - : int option = Some 2
```

8 7. Eight Queens Puzzle More info @ Wikipedia

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n* queens problem** of placing *n* non-attacking queens on an *n*×*n* chessboard, for which solutions exist for all natural numbers *n* with the exception of *n* = 2 and *n* = 3.[1]

History

Chess composer Max Bezzel published the eight



The only symmetrical solution to the eight queens puzzle (up to rotation and reflection)

The eight queens puzzle has 92 distinct solutions. If solutions that differ only by the symmetry operations of rotation and reflection of the board are counted as one, **the puzzle has 12 solutions**.

[30]:
```
(* Comproba si se comen *)
let come (i1,j1) (i2,j2) =
    i1=i2 || j1=j2 || abs(i1-i2)=abs(j1-j2);;   (* abs(i1-i2)=abs(j1-j2) DIAGONALES *)
```

[30]:
```
val come : int * int -> int * int -> bool = <fun>
```

[41]:
```
(* Para saber si polo camino que vai e compatible. Comprobando o de comer *)
(*
current_p --> Posicion actual
path -> lista cos seguintes movementos
*)
let rec compatible current_p path = match path with
    [] -> true (* Si non ten a donde ir ==> Compatible *)
  | h::t -> not (come current_p h) && compatible current_p t;;(* Remember Lazy Evaluation *)
```

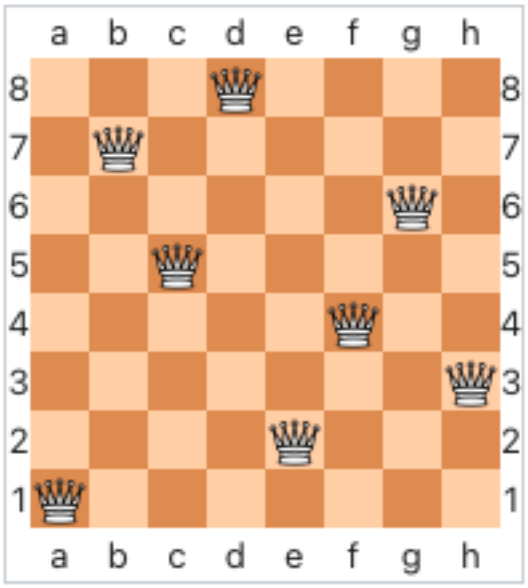
[41]:
```
val compatible : int * int -> (int * int) list -> bool = <fun>
```

[56]:
```
let reinas n =    (* Main Function *)
    let rec completa camino (i,j) =
        if i>n then camino                  (* Fila sobrepasa limite tablero *)
        else if j>n then raise Not_found      (* Columna sobrepasa limite tablero *)
        else if compatible (i,j) camino then
```

```
            try
                completa ((i,j)::camino) (i+1,1)
            with
                Not_found -> completa camino (i,j+1)

        else completa camino (i,j+1)  (*Si chegas aqui => Non sobrepasas limites tablero e
                                        a tua posicion actual --> Non compatible *)
    in completa [] (1,1) (*Empezamos desde (1,1) *)
    ;;

reinas 8;;
```

[56]:
```
val reinas : int -> (int * int) list = <fun>
```

[56]:
```
- : (int * int) list =
[(8, 4); (7, 2); (6, 7); (5, 3); (4, 6); (3, 8); (2, 5); (1, 1)]
```



Solution 3

[62]:
```
let reinas n =    (*Alternativa usando Some None*)
    let rec completa camino (i,j) =
        if i>n then Some camino                  (*Fila sobrepasa limite tablero*)
        else if j>n then None      (*Columna sobrepasa limite tablero*)
```

```
        else if compatible (i,j) camino then
                 match completa ((i,j)::camino) (i+1,1) with (* Cambia o try_with
 ↪por match *)
                     None -> completa camino (i,j+1)
                   | Some  s -> Some s

        else completa camino (i,j+1)
    in completa [] (1,1)
    ;;

reinas 8;;
reinas 0;;
```

[62]:
```
val reinas : int -> (int * int) list option = <fun>
```

[62]:
```
- : (int * int) list option =
Some [(8, 4); (7, 2); (6, 7); (5, 3); (4, 6); (3, 8); (2, 5); (1, 1)]
```

[62]:
```
- : (int * int) list option = Some []
```

Outra forma

[65]:
```
(* OUTRA FORMA *)
let reinas n =   (* Esta forma danos todas as solucions posibles. 92 solucions *)
    let rec completa camino (i,j) =
        if i>n then [camino]                  (* Fila sobrepasa limite tablero *)
        else if j>n then []      (* Columna sobrepasa limite tablero *)
        else if compatible (i,j) camino
             then completa ((i,j)::camino) (i+1,1) @  (* Intenta con i+1 e
 ↪despois j+1, en caso de non encontrar *)
                  completa camino (i,j+1)             (* inserta lista vacia, e
 ↪decir nada *)

        else completa camino (i,j+1)
    in completa [] (1,1)
    ;;
```

[65]:
```
val reinas : int -> (int * int) list list = <fun>
```

[63]:
```
let rec print_solucion = function
    [] -> print_newline ()
  | (_,y)::t -> print_int y; print_char ' ';
                print_solucion t;;
```

[63]:
```
val print_solucion : ('a * int) list -> unit = <fun>
```

[ ]:
```
let n_reinas n =
        let rec completa camino (i,j) =
        if i>n then print_solucion camino           (*Fila sobrepasa limite␣
 ↪tablero*)
        else if j>n then ()      (*Columna sobrepasa limite tablero*)
        else if compatible (i,j) camino
             then (completa ((i,j)::camino) (i+1,1) ;
                  completa camino (i,j+1))
        else completa camino (i,j+1)
    in completa [] (1,1)
    ;;

n_reinas 8;;
```

9 8. Exceptions

raise sirve para invocar errores

[66]:
```
(*raise    Sirve para  errores*)
(*EX: raise(Failure "hd")*)
Division_by_zero;;

Failure "a";;

Invalid_argument "e";;

Not_found;;
```

[66]:
```
- : exn = Division_by_zero
```

[66]:
```
- : exn = Failure "a"
```

[66]:
```
- : exn = Invalid_argument "e"
```

[66]:
```
- : exn = Not_found
```

Con exception creanse excepcions propias

[68]:
```
exception Fib (*Asi se crean excepcions propias*)
```

[68]:
```
exception Fib
```

9.1 8.1 Try With

[73]:
```
(* Interceptar excepcions *)
let hd' l =
    try Some (List.hd l)
    with Failure _ -> None;;
```

[73]:
```
val hd' : 'a list -> 'a option = <fun>
```

10 9. Type Synonyms Definir Novos Tipos de Datos

A *type synonym* is a new name for an already existing type.

[76]:
```
type maybe_an_int = (* Pode ser un int ou non ser nada *)
    Some of int
  | None
;;
```

[76]:
```
type maybe_an_int = Some of int | None
```

[77]:
```
type int_o_no =
    UnInt of int
   | NoInt;;
```

[77]:
```
type int_o_no = UnInt of int | NoInt
```

[78]:
```
let div m n = match m,n with
    UnInt x, UnInt 0 -> NoInt
    | UnInt x, UnInt y -> UnInt (x / y)
    | _ -> NoInt;;
```

[78]:
```
val div : int_o_no -> int_o_no -> int_o_no = <fun>
```

[82]:
```
let (//) x y = match (x,y) with  (* let (//) Creamos funcion tipo curry *)
    Some _, Some 0 -> None
  | Some a, Some b -> Some (a/b)
  | _ -> None
;;
```

```
[82]: val ( // ) : maybe_an_int -> maybe_an_int -> maybe_an_int = <fun>
```

```
[81]: Some 3 // Some 0;;
```

```
[81]: - : maybe_an_int = None
```

```
[83]: Some 7 // Some 3 // Some 2;; (*(7/3)/2*)
```

```
[83]: - : maybe_an_int = Some 1
```

```
[85]: type booleano = V | F;;

      V;;
      F;;
```

```
[85]: type booleano = V | F
```

```
[85]: - : booleano = V
```

```
[85]: - : booleano = F
```

```
[86]: let (&&&) b1 b2 = match b1,b2 with
          V, V -> V
        | _ -> F
      ;;

      F &&& V;;
```

```
[86]: val ( &&& ) : booleano -> booleano -> booleano = <fun>
```

```
[86]: - : booleano = F
```

```
[87]: let (|||) b1 b2 = match b1,b2 with
          V,_ | _,V -> V
        | _ -> F
      ;;

      F ||| V;;
```

```
[87]: val ( ||| ) : booleano -> booleano -> booleano = <fun>
```

```
[87]: - : booleano = V
```

10.1 9.1 Variants

A variant is a data type representing a value that is one of several possibilities. At their simplest, variants are like enums from C or Java

```
[90]: (* Variants Page 78 Real World Ocaml *)

type palo = Trebol | Diamante | Corazon | Pica;;

Trebol;; (* Individual names of the values of a variant are called␣
 ↪'Constructors' *)
```

```
[90]: type palo = Trebol | Diamante | Corazon | Pica
```

```
[90]: - : palo = Trebol
```

```
[91]: type palo =
    Trebol  of unit
  | Diamante of unit
  | Corazon of unit
  | Pica of unit;;

Pica();;
```

```
[91]: type palo =
    Trebol of unit
  | Diamante of unit
  | Corazon of unit
  | Pica of unit
```

```
[91]: - : palo = Pica ()
```

```
[92]: type otroint = Int of int;;

Int 3;;
```

```
[92]: type otroint = Int of int
```

[92]:
```
- : otroint = Int 3
```

[100]:
```
type numero = I of int | F of float;;

I 5;;
F 5.1;;
```

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

```
- : numero = I 5
```

```
- : numero = F 5.1
```

[103]:
```
let rec (++) n1 n2 = match n1,n2 with (*Podemos facer unha funcion que sume Ints e Floats*)
    I x, I y -> I (x + y)
  | F x, F y -> F (x +. y)
  | I x, F y -> F (float x +. y) (*"Casteo o int a float"*)
  | _ -> n2 ++ n1
;;

F 3.4 ++ I 5;;
```

```
val ( ++ ) : numero -> numero -> numero = <fun>
```

```
- : numero = F 8.4
```

[122]:
```
type nat = O | S of nat;; (** O -> Represenata o 0 (cero)*) (*S entendo que representa 1*)

O;;

S O;;
```

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

```
- : nat = O
```

```
- : nat = S O
```

[116]:
```
let rec sum m n = match n with (* Tail Recursive *)
    O -> m (*Caso base*) (*Representa o 0*)
| S i -> sum (S m) i ;;
```

[116]:
```
val sum : nat -> nat -> nat = <fun>
```

[123]:
```
let uno = S O;; (*O 1 esta formado por O (cero) e S*)
```

[123]:
```
val uno : nat = S O
```

[118]:
```
let dos = S uno;;
```

[118]:
```
val dos : nat = S (S O)
```

[120]:
```
let tres = S dos;;
```

[120]:
```
val tres : nat = S (S (S O))
```

[121]:
```
sum dos tres;; (* Funciona, obtemos 5 S *)
```

[121]:
```
- : nat = S (S (S (S (S O))))
```

10.2 9.2 Definicion Parametrizada

[124]:
```
type 'a quiza = (* Definicion de tipo parametrizada *)
    Algo of 'a
  | Nada;;

Algo 5;;
Nada;;
```

[124]:
```
type 'a quiza = Algo of 'a | Nada
```

[124]:
```
- : int quiza = Algo 5
```

[124]:
```
- : 'a quiza = Nada
```

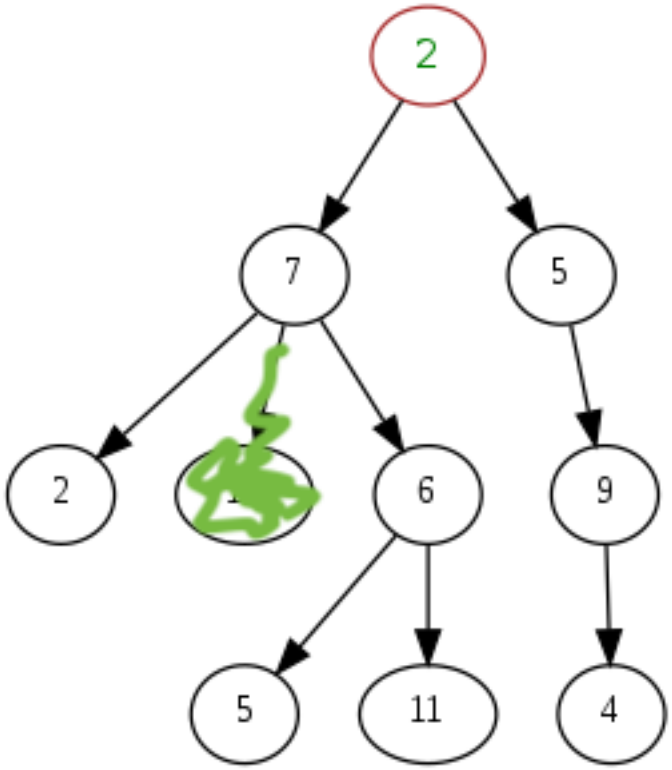
[126]:
```
(* Seria como o de arriba pero asi definimos unha infinidade de datos *)
type 'a option = (* Asi se define o option en OCAML *) (* Seria como o de␣
 ↪arriba pero asi definimos unha infinidade de datos *)
```

```
    Some of 'a
  | None;;
```

[126]:
```
type 'a option = Some of 'a | None
```

11 10. Trees

11.1 10.1 Binary Tree



11.1.1 10.1.1 Definition

[264]:
```
(* Funcion *)
type 'a tree = (* Se nos fixamos na estructura, esto e un arbol. V -> Raiz, N->␣
 ↪Fillos *)
    V
  | N of 'a * 'a tree * 'a tree;;
```

[264]:
```
type 'a tree = V | N of 'a * 'a tree * 'a tree
```

[265]:
```
V;;
N (5,V,V);;
```

```
[265]: - : 'a tree = V
```

```
[265]: - : int tree = N (5, V, V)
```

```
[266]: (* MiniExemplo *)
       let t5 = N (5,V,V);;
       let t6 = N (6,V,V);;

       N (8,t5,t6);;
       (* ------------------- *)
```

```
[266]: val t5 : int tree = N (5, V, V)
```

```
[266]: val t6 : int tree = N (6, V, V)
```

```
[266]: - : int tree = N (8, N (5, V, V), N (6, V, V))
```

```
[267]: let h x = N (x,V,V);; (* Arbol Hoja , solo ten raiz *)
```

```
[267]: val h : 'a -> 'a tree = <fun>
```

```
[268]: (* Representamos o arbol da imaxe *)
       let t6 = N (6, h 5,h 11);;
       let t9 = N (9,h 4,V);;
       let t5 = N (5,V,t9);;
       let t7 = N (7,h 2,t6);;
       let t  = N (2,t7,t5);;
```

```
[268]: val t6 : int tree = N (6, N (5, V, V), N (11, V, V))
```

```
[268]: val t9 : int tree = N (9, N (4, V, V), V)
```

```
[268]: val t5 : int tree = N (5, V, N (9, N (4, V, V), V))
```

```
[268]: val t7 : int tree = N (7, N (2, V, V), N (6, N (5, V, V), N (11, V, V)))
```

```
[268]: val t : int tree =
         N (2, N (7, N (2, V, V), N (6, N (5, V, V), N (11, V, V))),
```

```
N (5, V, N (9, N (4, V, V), V)))
```

11.1.2 10.1.2 Functions

10.1.2.1 Num_Nodos

[269]:
```
let rec n_nodos = function (* Calculo Nodos *)
    V -> 0
  | N (r,i,d) -> 1 + n_nodos i + n_nodos d;;        (*r-> raiz, i -> izquierda ,␣
 ↪d-> derecha*)

n_nodos t;;
```

[269]:
```
val n_nodos : 'a tree -> int = <fun>
```

[269]:
```
- : int = 9
```

10.1.2.2 Altura

[270]:
```
let rec altura = function (* Calculo altura de arbol *)
    V -> 0 (*Arbol vacio*)
  | N (r,i,d) -> 1 + max (altura i) (altura d);;

altura t;;
```

[270]:
```
val altura : 'a tree -> int = <fun>
```

[270]:
```
- : int = 4
```

10.1.2.3 Recorridos

[271]:
```
let rec preorder = function (*Root,Left,Right*)
    V -> []
  | N (r,i,d) -> r:: (preorder i @ preorder d);;

preorder t;;
```

[271]:
```
val preorder : 'a tree -> 'a list = <fun>
```

[271]:
```
- : int list = [2; 7; 2; 6; 5; 11; 5; 9; 4]
```

[272]:
```
let rec postorder = function (*Left,Right,Root*)
    V -> []
```

```
  | N (r,i,d) -> (postorder i @ postorder d) @ [r] ;;

postorder t;;
```

```
[272]: val postorder : 'a tree -> 'a list = <fun>
```

```
[272]: - : int list = [2; 5; 11; 6; 7; 4; 9; 5; 2]
```

```
[273]: let rec inorder = function (*Left,Root,Right*)
         V -> []
       | N (v, l, r) -> inorder l @ (v :: inorder r);;

inorder t;;
```

```
[273]: val inorder : 'a tree -> 'a list = <fun>
```

```
[273]: - : int list = [2; 7; 5; 6; 11; 2; 5; 4; 9]
```

10.1.2.4 Leaf

```
[274]: let rec leaf = function
         V -> [] (*Vacio*)
       | N (r,V,V) -> [r] (*Arbol con solo raiz*)
       | N (r,i,d) -> leaf i @ leaf d;;

leaf t;;
```

```
[274]: val leaf : 'a tree -> 'a list = <fun>
```

```
[274]: - : int list = [2; 5; 11; 4]
```

10.1.2.5 Mirror

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

```
[276]: val mirror : 'a tree -> 'a tree = <fun>
```

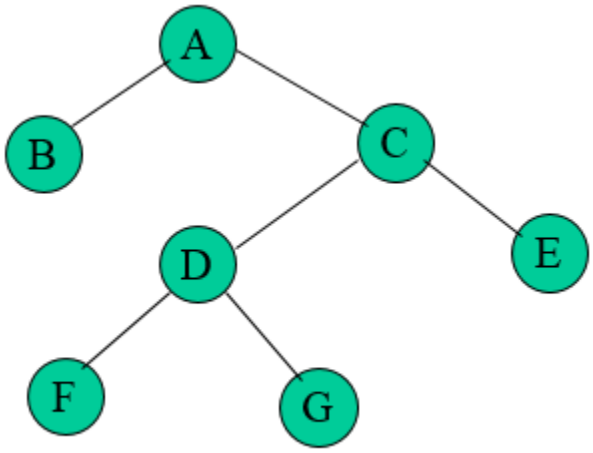
```
[277]: t;;
```

[277]:
```
- : int tree =
N (2, N (7, N (2, V, V), N (6, N (5, V, V), N (11, V, V))),
 N (5, V, N (9, N (4, V, V), V)))
```

[278]:
```
mirror t;;
```

[278]:
```
- : int tree =
N (2, N (5, N (9, V, N (4, V, V)), V),
 N (7, N (6, N (11, V, V), N (5, V, V)), N (2, V, V)))
```

11.2 10.2 Strict Binary Tree



A binary tree in which **every node has either 0 or two children** is called strict binary tree.

11.2.1 10.2.1 Definition

[229]:
```
(* Intentamos redefinir leaf para arboles strictos *)
type 'a sttree =
    SL of 'a                          (* SL -> Leaf *)
  | SN of 'a * 'a sttree * 'a sttree;; (* SN -> Node *)
```

[229]:
```
type 'a sttree = SL of 'a | SN of 'a * 'a sttree * 'a sttree
```

[230]:
```
SL 'F';;
```

[230]:
```
- : char sttree = SL 'F'
```

[231]:
```
SN ('A',SL 'F',SL 'F');;
```

[231]:
```
- : char sttree = SN ('A', SL 'F', SL 'F')
```

[237]:
```
(*Representamos o arbol da imaxe*)
let tD = SN ('D',SL 'F',SL 'G');;

let tC = SN ('C',tD,SL 'E');;

let st  = SN ('A',SL 'B',tC);;
```

[237]:
```
val tD : char sttree = SN ('D', SL 'F', SL 'G')
```

[237]:
```
val tC : char sttree = SN ('C', SN ('D', SL 'F', SL 'G'), SL 'E')
```

[237]:
```
val st : char sttree =
  SN ('A', SL 'B', SN ('C', SN ('D', SL 'F', SL 'G'), SL 'E'))
```

11.2.2 10.2.2 Funcions

10.2.2.1 Leaves

[301]:
```
let rec leaves = function (*List of leaves*)
    SL r -> [r]
  | SN (_,i,d) -> leaves i @ leaves d;;

leaves st;;
```

[301]:
```
val leaves : 'a sttree -> 'a list = <fun>
```

[301]:
```
- : char list = ['B'; 'F'; 'G'; 'E']
```

10.2.2.2 Tree_of_Sttree

`tree_of_sttree` converts `'a sttree` onto `'a tree`

[239]:
```
let rec tree_of_sttree = function (* Converts 'a sttree to binary tree (10.1.1)␣
↪*)
    SL x -> N (x,V,V)
  | SN (r,i,d) -> N (r,tree_of_sttree i, tree_of_sttree d);;
```

[239]:
```
val tree_of_sttree : 'a sttree -> 'a tree = <fun>
```

[244]:
```
st;; (* Devolve como sttree *)
```

```
[244]: - : char sttree =
       SN ('A', SL 'B', SN ('C', SN ('D', SL 'F', SL 'G'), SL 'E'))
```

```
[245]: let tt = tree_of_sttree st;; (* Devolve como tree *)
```

```
[245]: val tt : char tree =
         N ('A', N ('B', V, V),
          N ('C', N ('D', N ('F', V, V), N ('G', V, V)), N ('E', V, V)))
```

10.2.2.3 Sttree_of_Tree

```
[254]: let rec sttree_of_tree = function
           V -> raise (Invalid_argument "sttree_of_tree")
         | N (r,V,V) -> SL r
         | N (r,i,d) -> SN (r,sttree_of_tree i, sttree_of_tree d);;
```

```
[254]: val sttree_of_tree : 'a tree -> 'a sttree = <fun>
```

```
[255]: sttree_of_tree tt;;
```

```
[255]: - : char sttree =
       SN ('A', SL 'B', SN ('C', SN ('D', SL 'F', SL 'G'), SL 'E'))
```

10.2.2.4 Mirror

```
[256]: let rec mirror = function
           SL r -> SL r
         | SN (r,i,d) -> SN (r,mirror d, mirror i);;
```

```
[256]: val mirror : 'a sttree -> 'a sttree = <fun>
```

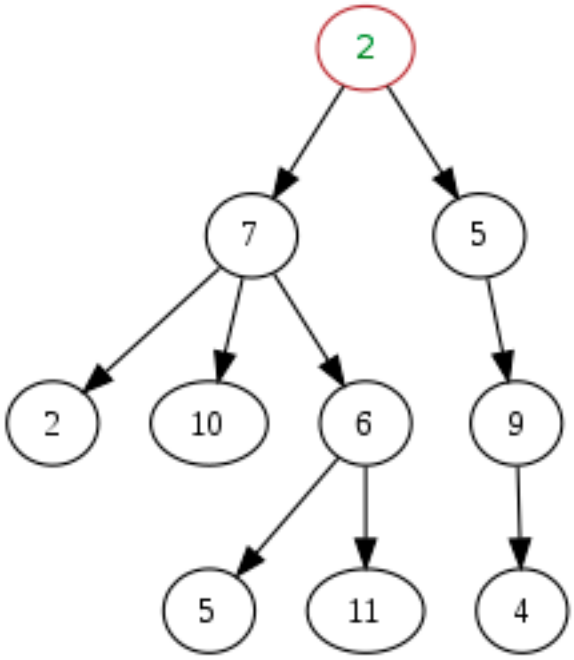
```
[251]: st;;
```

```
[251]: - : char sttree =
       SN ('A', SL 'B', SN ('C', SN ('D', SL 'F', SL 'G'), SL 'E'))
```

```
[257]: mirror st;;
```

```
[257]: - : char sttree =
       SN ('A', SN ('C', SL 'E', SN ('D', SL 'G', SL 'F')), SL 'B')
```

11.3 10.3 Spanning Tree (w/ Lists)



11.3.1 10.3.1 Definition

[288]:
```
type 'a gtree =  (* Constructor *)
    Gt of 'a * 'a gtree list ;;
```

[288]:
```
type 'a gtree = Gt of 'a * 'a gtree list
```

[289]:
```
let s x = Gt (x,[]);; (* Leaf Constructor "Shortcut" *)
```

[289]:
```
val s : 'a -> 'a gtree = <fun>
```

[290]:
```
(* Tree from Image *)
let t5 = Gt (5, [Gt (9,[s 4])]);;

let t7 = Gt (7,[s 2;s 10; Gt (6,[s 5;s 11])]);;

let t = Gt (2 ,[t7;t5]);;
```

[290]:
```
val t5 : int gtree = Gt (5, [Gt (9, [Gt (4, [])])])
```

[290]:
```
val t7 : int gtree =
  Gt (7, [Gt (2, []); Gt (10, []); Gt (6, [Gt (5, []); Gt (11, [])])])
```

[290]:
```
val t : int gtree =
  Gt (2,
   [Gt (7, [Gt (2, []); Gt (10, []); Gt (6, [Gt (5, []); Gt (11, [])])]);
    Gt (5, [Gt (9, [Gt (4, [])])])])
```

11.3.2 10.3.2 Functions

10.3.2.1 Height

[303]:
```
let rec height = function
      Gt (_,[]) -> 1
    | Gt (v,h::t) -> 1 - (List.length (h::t) - 1) + max (height h) (height␣
 ↪(Gt(v,t)));;

height t;;
```

[303]:
```
val height : 'a gtree -> int = <fun>
```

[303]:
```
- : int = 4
```

10.3.2.2 Num_Nodes

[291]:
```
let rec n_nodos_gt (Gt (r,l)) = List.fold_left (+) 1 (List.map n_nodos_gt l);;␣
 ↪(* Contador numero de nodos *)

n_nodos_gt t;;
```

[291]:
```
val n_nodos_gt : 'a gtree -> int = <fun>
```

[291]:
```
- : int = 10
```

[293]:
```
let rec nnodos = function (* Outra forma de contar os nodos *)
    Gt (_,[]) -> 1
  | Gt (m,h::t) -> nnodos h + nnodos (Gt (m,t));;

n_nodos_gt t;;
```

[293]:
```
val nnodos : 'a gtree -> int = <fun>
```

[293]:
```
- : int = 10
```

10.3.2.3 Leaves

[302]:
```
let rec leaves tree = match tree with
  Gt (a,[]) -> [a]
| Gt (a,h::t) ->if List.length (h::t) == 1 then leaves h else leaves h @ leaves␣
 ↪(Gt (a,t))
;;

leaves t;;
```

[302]:
```
val leaves : 'a gtree -> 'a list = <fun>
```

[302]:
```
- : int list = [2; 10; 5; 11; 4]
```

10.3.2.4 Mirror

[304]:
```
let rec mirror tree = match tree with
      Gt (a,[]) -> Gt (a,[])
    | Gt (a,l) -> Gt (a, List.rev_append (List.map (mirror) l) []);;
```

[305]:
```
t;;
```

[305]:
```
- : int gtree =
Gt (2,
 [Gt (7, [Gt (2, []); Gt (10, []); Gt (6, [Gt (5, []); Gt (11, [])])]);
  Gt (5, [Gt (9, [Gt (4, [])])])])
```

[306]:
```
mirror t;;
```

[306]:
```
- : int gtree =
Gt (2,
 [Gt (5, [Gt (9, [Gt (4, [])])]);
  Gt (7, [Gt (6, [Gt (11, []); Gt (5, [])]); Gt (10, []); Gt (2, [])])])
```

10.3.2.3 Traverse

10.3.2.3.1 Breadth First

[298]:
```
let anchura (Gt (r,l)) =  (* acc -> acomulador *) (* r-> raiz l -> lista ramas␣
 ↪*) (*Tail Recursive*)
   let rec aux acc  next = match next with
       [] -> List.rev_append acc []
     | Gt (r1,l1)::t -> aux (r1::acc) (t @ l1)
   in aux [r] l;;

anchura t;;
```

[298]:
```
val anchura : 'a gtree -> 'a list = <fun>
```

[298]:
```
- : int list = [2; 7; 5; 2; 10; 6; 9; 5; 11; 4]
```

[297]:
```
let rec anchura = function  (* Not Tail *)
    Gt (r,[]) -> [r]
  | Gt (r, Gt (r1,l1)::t) -> r :: anchura (Gt (r1,t @ l1));;

anchura t;;
```

[297]:
```
val anchura : 'a gtree -> 'a list = <fun>
```

[297]:
```
- : int list = [2; 7; 5; 2; 10; 6; 9; 5; 11; 4]
```

10.3.2.3.2 Preorder

[307]:
```
let rec preorder =
  let rec aux l_out l_in = match l_in with
    [] -> l_out
  | h::t -> aux (l_out @ (preorder h)) t
in
  function
    Gt (a,[]) -> [a]
  | Gt (a,l) -> a::aux [] l
  ;;
```

[307]:
```
val preorder : 'a gtree -> 'a list = <fun>
```

[308]:
```
preorder t;;
```

[308]:
```
- : int list = [2; 7; 2; 10; 6; 5; 11; 5; 9; 4]
```

10.3.2.3.3 Postorder

[311]:
```
let rec postorder tree = match tree with
      Gt (a,[]) -> [a]
    | Gt (a,h::t) ->  postorder h @ postorder (Gt (a,t))
;;
```

[311]:
```
val postorder : 'a gtree -> 'a list = <fun>
```

[310]:
```
postorder t;;
```

[310]:
```
- : int list = [2; 10; 5; 11; 6; 7; 4; 9; 5; 2]
```

12 11. Programacion Imperativa

12.0.1 Estes de output non se mostran ben en Jupyter. Metolle o flush sempre por culpa do Jupyter

12.1 Todo sobre entrada e salida en Documentacion OCaml

12.2 11.1 I/O

[341]:
```
output_char stdout 'X';; (*Expected --> *)  (* X- : unit = () *)
```

[341]:
```
- : unit = ()
```

[341]:
```
- : unit = ()
```

[342]:
```
output_char stdout 'A'; output_char stdout 'B';; (* AB- : unit = () *)
```

[342]:
```
- : unit = ()
```

[342]:
```
- : unit = ()
```

[313]:
```
let print_char c = output_char stdout c;; (*Correcto*)
```

[313]:
```
val print_char : char -> unit = <fun>
```

[315]:
```
"hola".[3];; (* Devolve a letra 3 *) (*Visto ao principio de todo dos apuntes *)
```

[315]:
```
- : char = 'a'
```

[317]:
```
let output_string canal s = (*Salida Correcta*) (*Deletrea a entrada mostrandoa␣
↪polo canal indicado *)
   let n = String.length s in
   let rec loop i =
       if i>= n then ()
       else (output_char canal s.[i]; loop (i+1))
   in
       loop 0;;
```

```
[317]: val output_string : out_channel -> string -> unit = <fun>
```

```
[343]: output_string stdout "hola";; (*Expected --> hola- : unit = ()*)
```

```
[343]: - : unit = ()
```

```
[343]: - : unit = ()
```

```
[319]: let print_string s = output_string stdout s;; (* Mostrar por stdout un String *)
```

```
[319]: val print_string : string -> unit = <fun>
```

```
[325]: let print_endline s = print_string (s ^ "\n");; (* Mostrar por stdout un String␣
        ↪metendo un salto de linea  *)

       let print_newline () = print_endline "";; (*O mismo*)

       (*
       utop # print_endline "Hi!";;
       Hi!
       - : unit = ()
       *)
```

```
[325]: val print_endline : string -> unit = <fun>
```

```
[325]: val print_newline : unit -> unit = <fun>
```

`input_line` : Read characters from the given input channel, until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

```
[326]: let read_line () = input_line stdin;; (*Lee caracteres por entrada ata encontrar␣
        ↪un \n *)
```

```
[326]: val read_line : unit -> string = <fun>
```

Volcar a archivos. Como en C

12.2.1 11.1.1 Open_Out

`open_out`:Open the named file for writing, and return a new output channel on that file, positioned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists.

[327]:
```
open_out ;;
```

[327]:
```
- : string -> out_channel = <fun>
```

`close_out`: Close the given channel, flushing all buffered write operations. **Output functions raise a Sys_error exception when they are applied to a closed output channel, except close_out and flush, which do nothing when applied to an already closed channel**. Note that close_out may raise Sys_error if the operating system signals an error when flushing or closing.

[331]:
```
close_out;;
```

[331]:
```
- : out_channel -> unit = <fun>
```

`flush`: Flush the buffer associated with the given output channel, performing all pending writes on that channel. Interactive programs must be careful about flushing standard output and standard error at the right time.

[354]:
```
let s = open_out "../Test/prueba.txt";; (* Creou archivo prueba e meteu a salida␣
 ↪dentro *)
output_string s "ABCDE";;
flush s;; (* Para forzar escritura *)
close_out s;;
```

[354]:
```
val s : out_channel = <abstr>
```

[354]:
```
- : unit = ()
```

[354]:
```
- : unit = ()
```

[354]:
```
- : unit = ()
```

[347]:
```
let rec output_string_list c = function
    [] -> ()
  | h::t -> output_string c (h ^ "\n");
            output_string_list c t;;
```

[347]:
```
val output_string_list : out_channel -> string list -> unit = <fun>
```

[351]:
```
output_string_list stdout ["Welcome";"to";"utop"];;

(*
```

```
utop # output_string_list stdout ["Welcome";"to";"utop"];;
Welcome
to
utop
- : unit = ()

*)
```

[351]:
```
- : unit = ()
```

[349]:
```
(* Outra forma --> Non recursiva *)
let output_string_list c l =
    List.iter (fun s -> output_string c (s ^ "\n")) l
    ;;
```

[349]:
```
val output_string_list : out_channel -> string list -> unit = <fun>
```

12.2.2 11.1.2 Open_In

`open_in`: Open the named file **for reading**, and return a new input channel on that file, positioned at the beginning of the file.

[352]:
```
open_in;;
```

[352]:
```
- : string -> in_channel = <fun>
```

[361]:
```
let c = open_in "../Test/prueba.txt";; (* Con open_in temoslle que meter o nome␣
↪dun ficheiro QUE EXITA *)
```

[361]:
```
val c : in_channel = <abstr>
```

[362]:
```
input_char c;;
```

[362]:
```
- : char = 'A'
```

[363]:
```
input_line c ;;
```

[363]:
```
- : string = "BCDE"
```

[367]:
```
(* input_char c;;*) (* Salta excepcion porque se acabou o contido do arquivo *)␣
↪(* Exception: End_of_file.*)
```

```
close_in c;; (*Temos que cerrar o archivo*)
```

[367]:
```
- : unit = ()
```

input_string_list : in_channel -> string_list;;

[365]:
```
(*let rec input_string_list f = (*Ollo asi esta mal. Neste caso evaluase
↪primeiro a cola e despois a cabza, polo que nos da un bucle infinito*)
   try
       input_line f :: input_string_list f (*Si se produce end of line e porque
↪input_line o dou*)
   with End_of_file -> []   (*Cando se acabe a lista ou si desde o principio o
↪archivo estaba vacio  devolver unha lista vacia *)
*)
```

[371]:
```
let f = open_in "../Test/prueba.txt";;
```

[371]:
```
val f : in_channel = <abstr>
```

[373]:
```
let rec input_string_list f = (*Leemos o archivo e sacamos unha lista coas
↪lineas *)
   try
       let s = input_line f in (*Usamos let in para asegurarnos de que primeiro
↪se fai input_line e despois input_string_list. Xa que o fai ao reves teremos
↪bucle infinito*)
           s :: input_string_list f (*Si se produce end of line e porque
↪input_line o dou*)
   with End_of_file -> [];;
```

[373]:
```
val input_string_list : in_channel -> string list = <fun>
```

[374]:
```
input_string_list f;;
```

[374]:
```
- : string list =
["Defqon.1 Weekend Festival 2017 | Phuture Noize"; ""; "0"; "TweetShare";
 "Roundup January 23rd 2022"; "";
 "The sixty-nine installment of the 9to5Linux Weekly Roundup is here for the
week ending on January 23rd, 2022, keeping";
 "you guys up to date with the most important things happening "]
```

[375]:
```
close_in f;;
```

[375]:
```
- : unit = ()
```

12.3 11.2 Variables (Variables != let)

Son especificas ao tipo de dato (como en C)

Para calquera tipo de dato 'a exite a variable 'a ref . Ex: char -> char ref

12.3.1 11.2.1 Creation

Crear variable, crease con `ref` e para facerlle referencia metese `let`

Para creala tes que inicializala

[377]:
```
ref;;
```

[377]:
```
- : 'a -> 'a ref = <fun>
```

[378]:
```
(!);; (*Aplicase as variables para devolver o seu contido*)
```

[378]:
```
- : 'a ref -> 'a = <fun>
```

[379]:
```
let i = ref 0;; (*Devolve unha variable donde podo almacenar ints. Neste momento␣
↪conten un 0*)
                (*i e unha caixa que conten ints*)
```

[379]:
```
val i : int ref = {contents = 0}
```

[382]:
```
i;;
```

[382]:
```
- : int ref = {contents = 0}
```

[383]:
```
!i;;  (*Asi accedemos ao seu contido*)
```

[383]:
```
- : int = 0
```

[384]:
```
!i + 1;;
```

[384]:
```
- : int = 1
```

12.3.2 11.2.2 Modificar unha varible

[385]:
```
(:=);; (*Usase para modificar variable*)
```

[385]:
```
- : 'a ref -> 'a -> unit = <fun>
```

[387]:
```
!i;;
```

[387]:
```
- : int = 3
```

[391]:
```
i := 5+1;;
```

[391]:
```
- : unit = ()
```

[392]:
```
!i;;
```

[392]:
```
- : int = 6
```

[393]:
```
i;;
```

[393]:
```
- : int ref = {contents = 6}
```

12.4 11.3 Bucles

12.4.1 11.3.1 Bucle FOR

Exer: Redefinir o fact sin usar recursividade

[394]:
```
let fact n =
    let f = ref 1 in
    for i = 1 to n do
        f := !f * i
    done; (*Primeiro facemos o calculo*)
    !f;; (*Despois devolvemos resultado*)
```

[394]:
```
val fact : int -> int = <fun>
```

[395]:
```
fact 6;;
```

[395]:
```
- : int = 720
```

12.4.2 11.3.2 Bucle WHILE

[396]:
```
let fact n =
    let f = ref 1 and i = ref 1 in
    while !i <= n do
        f:= !f * !i;
        i:= !i + 1
    done;
```

```
    !f (* En f queda gardado o resultado *)
    ;;
```

[396]:
```
val fact : int -> int = <fun>
```

[397]:
```
fact 5;;
```

[397]:
```
- : int = 120
```

12.5 11.4 Structs

12.5.1 11.4.1 ARRAY

Vacio

11.4.1.1 Construction

[398]:
```
[||];;
```

[398]:
```
- : 'a array = [||]
```

De tipo int

[399]:
```
let array = [|8;1;4;3|];;
```

[399]:
```
val array : int array = [|8; 1; 4; 3|]
```

11.4.1.2 Related Functions

[403]:
```
array.(0);; (*Acceso a posicion 0*)

Array.get ;;

Array.get array 0;; (* Outra forma de acceder*)
```

[403]:
```
- : int = 8
```

[403]:
```
- : 'a array -> int -> 'a = <fun>
```

[403]:
```
- : int = 8
```

[404]:
```
Array.set ;;

Array.set array 1 20;; (*Insertar/SOBRESCRIBIR nunha posicion concreta*)
```

```
array;;
```

```
[404]: - : 'a array -> int -> 'a -> unit = <fun>
```

```
[404]: - : unit = ()
```

```
[404]: - : int array = [|8; 20; 4; 3|]
```

```
[405]: Array.make ;;

Array.make 10 0;; (* Crea array de 10 ceros *)
```

```
[405]: - : int -> 'a -> 'a array = <fun>
```

```
[405]: - : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
[408]: let v = Array.init 5 (fun _ -> Random.float 1.);; (* Como coas listas *)
```

```
[408]: val v : float array =
  [|0.94674605093970754; 0.0011806719842075924; 0.535789075169210594;
    0.949948072115459508; 0.625081654014535859|]
```

```
[409]: let w = Array.copy v;;
```

```
[409]: val w : float array =
  [|0.94674605093970754; 0.0011806719842075924; 0.535789075169210594;
    0.949948072115459508; 0.625081654014535859|]
```

COIDADO !! SORT E DESTRUCTIVO

```
[410]: Array.sort compare v;; (* COMPARA *)

v;;
```

```
[410]: - : unit = ()
```

```
[410]: - : float array =
[|0.0011806719842075924; 0.535789075169210594; 0.625081654014535859;
  0.94674605093970754; 0.949948072115459508|]
```

[412]:
```
Array.sort compare w;; (* COMPARA *) (* SE NON QUEREMOS DESTRUIR FACEMOS COPY *)

w;;
```

[412]:
```
- : unit = ()
```

[412]:
```
- : float array =
[|0.00118067198420759249; 0.535789075169210594; 0.625081654014535859;
  0.94674605093970754; 0.949948072115459508|]
```

PRODUCTO ESCALAR

[413]:
```
let vprod v1 v2 = (* Forma 1 *)
    let l = Array.length v1 in
        if Array.length v2 = l then
            let p = ref 0. in
                for i = 0 to l-1 do
                    p:= !p +. v1.(i) *. v2.(i)
                done;
                !p
        else
            raise (Invalid_argument "vprod")
    ;;
```

[413]:
```
val vprod : float array -> float array -> float = <fun>
```

[414]:
```
vprod v w;;
```

[414]:
```
- : float = 2.47652782592836962
```

[415]:
```
let vprod v1 v2 =  (* Outra forma *)
    Array.fold_left (+.) 0. (Array.map2 ( *. ) v1 v2)
;;
```

[415]:
```
val vprod : float array -> float array -> float = <fun>
```

[416]:
```
vprod v w;;
```

[416]:
```
- : float = 2.47652782592836962
```

12.5.2 11.4.2 Records

[417]:
```
type persona = {nombre : string ; edad : int; };;
```

[417]:
```
type persona = { nombre : string; edad : int; }
```

[418]:
```
let p1 = {nombre= "Jose" ; edad = 17};;

p1.nombre;;
p1.edad;;
```

[418]:
```
val p1 : persona = {nombre = "Jose"; edad = 17}
```

[418]:
```
- : string = "Jose"
```

[418]:
```
- : int = 17
```

[419]:
```
let p2 = {edad=50 ; nombre= "Hi"};; (* Orden da igual *)

p2.nombre;;
p2.edad;;
```

[419]:
```
val p2 : persona = {nombre = "Hi"; edad = 50}
```

[419]:
```
- : string = "Hi"
```

[419]:
```
- : int = 50
```

[422]:
```
let mas_vieja p = {nombre =p.nombre ; edad=p.edad+1};;

mas_vieja p2;; (* Modifica en local, pero non o original *)

p2;;
```

[422]:
```
val mas_vieja : persona -> persona = <fun>
```

[422]:
```
- : persona = {nombre = "Hi"; edad = 51}
```

[422]:
```
- : persona = {nombre = "Hi"; edad = 50}
```

```
[424]: let mas_vieja p = {p with edad=p.edad+1};; (* Outra forma *)

       mas_vieja p2;;

       p2;;
```

```
[424]: val mas_vieja : persona -> persona = <fun>
```

```
[424]: - : persona = {nombre = "Hi"; edad = 51}
```

```
[424]: - : persona = {nombre = "Hi"; edad = 50}
```

Pode ser mutable

```
[425]: type persona_mutable = {nombre : string ; mutable edad : int; };;
```

```
[425]: type persona_mutable = { nombre : string; mutable edad : int; }
```

```
[426]: let p1 = {nombre= "Jose" ; edad = 17};;
```

```
[426]: val p1 : persona_mutable = {nombre = "Jose"; edad = 17}
```

```
[427]: let envejece p = p.edad <- p.edad +1;;

       envejece p1;;

       p1;;
```

```
[427]: val envejece : persona_mutable -> unit = <fun>
```

```
[427]: - : unit = ()
```

```
[427]: - : persona_mutable = {nombre = "Jose"; edad = 18}
```

13 12. Miscellaneous

```
[428]: let n = ref 0;;

       let turno () = (* Devolve o valor de n despois de incrementala *)
           n:= !n + 1;
```

```
    !n;;

let reset () =
    n:=0
```

```
[428]: val n : int ref = {contents = 0}
```

```
[428]: val turno : unit -> int = <fun>
```

```
[428]: val reset : unit -> unit = <fun>
```

```
[437]: turno ();;
```

```
[437]: - : int = 1
```

```
[437]: turno ();;
```

```
[437]: - : int = 1
```

Executando neste orden, vemos como os let non son variables. Devolver 12

```
[429]: let suman x = x + n ;;
       let n = 1000;;
```

```
File "[429]", line 1, characters 18-19:
1 | let suman x = x + n ;;
                      ^
Error: This expression has type int ref
       but an expression was expected of type int
```

Si a defino asi en local, non vou poder acceder nin modificar n fora da funcion turno

```
[438]: let turno = function () ->
           let n = ref 0 in
           n:= !n + 1;
           !n;;
```

```
[438]: val turno : unit -> int = <fun>
```

```
[439]: turno ();;
```

[439]:
```
- : int = 1
```

[440]:
```
turno ();;
```

[440]:
```
- : int = 1
```

[452]:
```
let turno = (*Asi si se modifica*)
    let n = ref 0 in
    function () ->
        n:= !n + 1;
        !n;;
```

[452]:
```
val turno : unit -> int = <fun>
```

[453]:
```
turno ();;
```

[453]:
```
- : int = 1
```

[454]:
```
turno ();;
```

[454]:
```
- : int = 2
```

13.1 12.1 Como definir Modulo (o do .mli)

[455]:
```
module Counter :
sig (*:sig para comezar a interfaz. O que se pode ver desde fora *)

val turno: unit -> int
val reset: unit -> unit


end
= struct (*=struct para comezar a implementacion*)

    let n = ref 0

    let turno () = (* Devolve o valor de n despois de incrementala *)
        n:= !n + 1;
        !n

    let reset () =
        n:=0
```

```
end;;
```

```
[455]: module Counter : sig val turno : unit -> int val reset : unit -> unit end
```

```
[456]: Counter.turno ();;
```

```
[456]: - : int = 1
```

```
[457]: Counter.reset ();;
```

```
[457]: - : unit = ()
```

```
[458]: Counter.turno ();;
```

```
[458]: - : int = 1
```

12.1.1 FUNCTOR Co Functor podo crear multiples instancias de un modulo

```
[460]: module Counter () : (* ASI SERIA UN FUNTOR *)
sig (*:sig para comezar a interfaz. O que se pode ver desde fora *)

val turno: unit -> int
val reset: unit -> unit


end
= struct (*=struct para comezar a implementacion*)

    let n = ref 0

    let turno () = (* Devolve o valor de n despois de incrementala *)
        n:= !n + 1;
        !n

    let reset () =
        n:=0

end;;
```

```
[460]: module Counter :
  functor () -> sig val turno : unit -> int val reset : unit -> unit end
```

[461]:
```
module A = Counter ();; (* Creamos un modulo A que actua como Counter *)

module B = Counter ();;
```

[461]:
```
module A : sig val turno : unit -> int val reset : unit -> unit end
```

[461]:
```
module B : sig val turno : unit -> int val reset : unit -> unit end
```

[463]:
```
A.turno ();;
```

[463]:
```
- : int = 2
```

[464]:
```
A.turno ();;
```

[464]:
```
- : int = 3
```

[465]:
```
B.turno ();;
```

[465]:
```
- : int = 2
```

[466]:
```
A.reset ();;
```

[466]:
```
- : unit = ()
```

[467]:
```
A.turno ();;
```

[467]:
```
- : int = 1
```