APUNTES_PP-2223.pdf

miguel_cd263

Paradigmas de Programación

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

PP

APLICACIÓN DE UNA FUNCIÓN

- <f> <x> → <aplicación de la función> <argumento>
  - (function x -> 2*x) (3+1)
- Siempre se evalúa primero el argumento y luego se hace la imagen
  - (function x -> 2*x) (2+1) → (function x -> 2*x) (3)
- Todas las expresiones (e) tienen que ser compatibles con los patrones (p)
  - (function <p1> -> <e1> | · · · | <p4> -> <e4>) <e>

PATTERN MATCHING

- Similar a la forma function
  - match <e> with <p1> -> <e1> | · · · | <p4> -> <e4>

POLIMORFISMO

- Función de un mismo tipo (ej. función int en int)

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# id 8;;
- : int = 8
# id abs;;
- : int -> int = <fun>
# id abs 7;;
- : int = 7
```

- Si hay más de una función (id y abs) se empieza por la izquierda (asociación por la izquierda) pero si hay más de una operación se asocian por la derecha
  - id abd 7 → (id abs) 7
  - 2.0**3.0**2.0 = 2.0**9.0 = 512

SENTENCIA IF ELSE

- <b>: bool
- <e1>: alfa
- <e2>: alfa
- if <b> then <e1> else <e2>: alfa
  - if 2 < 3 then 2 + 1 else 3 + 5;;

SENTENCIA ABS CON IF ELSE

```
# let abs = function x ->
        if x < 0 then -x else x;;
# let abs x = if x < 0 then -x else x;;
```

SENTENCIA ABS SIN IF ELSE

```
# let abs x = (function true -> -x | false -> x) (x < 0);;
```

&&

- Si b1 y b2 son de tipo bool se puede poner && y es de tipo bool
- <b1>: bool
- <b2>: bool
- (<b1> && <b2>): bool
- Primero se evalua b1, si da false, toda la frase da false, si b1 da true, se evalua b2 y la frase es lo que de b2

FUNCIONES SIMILARES

```
# let l = function r ->
        let pi = 2. +. asin 1.
        in 2. *. pi *. r;;
# let l = let pi = 2. *. asin 1.
        in function r -> 2. *. pi *. r;;
```

FUNCIONES CURRY

- Las funciones que devuelven funciones se llaman "curry"
- Función curry de f:
  - let f = function x -> (function y -> x + y);;

```
# let f = function x -> (function y -> x + y);;
# f 1 7;; → esto es lo mismo que (f 1) 7
- : int = 8
```

FUNCIÓN PRED

- Suma "-1" al valor que se escriba a continuación
  - let p = (+) (-1);;

```
# pred;;
- : int -> int = <fun>
# pred 3;;
- : int = 2;;
# pred (-3);;
- : int = (-4);;
```

FUNCIÓN OP

- Multiplica por "-1" el valor que se escriba a continuación

```
# let op = ( * ) (-1);;
val op : int -> int = <fun>
# op 4;;
- : int = -4
```

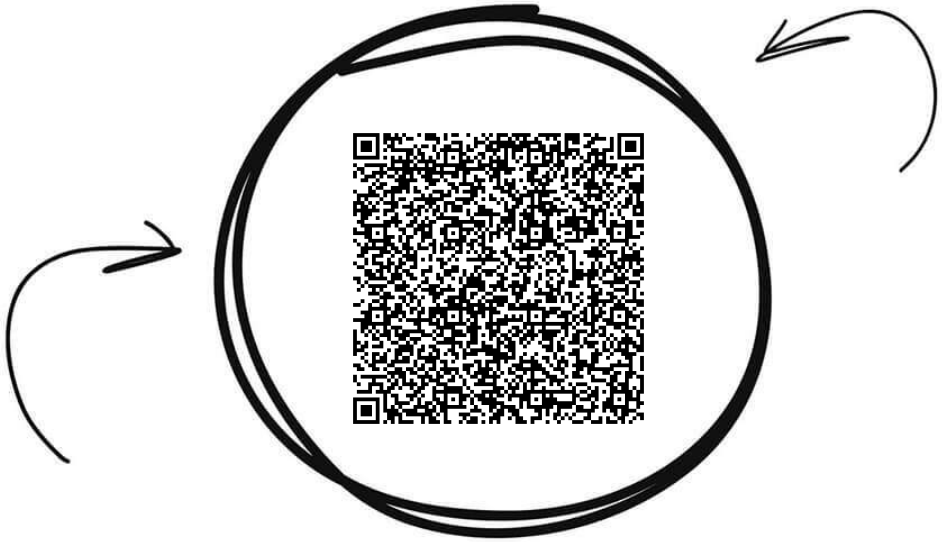

SUBRAYA ESTO: CÓMETE UN TACO.

PEKIS®
for Foodies

¿RELLENANDO APUNTES? RELLENATE UN TACO.

RELLENOS PARA TACOS PEKIS® BRUTALES.

Paradigmas de Programación



Banco de apuntes de la

WUOLAH

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

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

PRODUCTO CARTESIANO, PAR DE ENTEROS

- Producto de dos enteros

```
# 3,2;;
- : int * int = (3, 2)
```

- Producto de un entero y un float

```
# 3,2.;;
- : int * float = (3, 2.0)
```

- Producto de un string y un booleano

```
# "true", false;;
- : string * bool = ("true", false)
```

- Cuando en un tipo hay productos cartesianos y flechas, y no se ponen paréntesis, primero se evalua el producto cartesiano. Tiene prioridad el asterisco "*" antes que la flecha "->"
    - let s = int * int -> int = <fun>

OBRADORES INFIJOS

```
# (^) "Hola";;
- : string -> string = <fun>
# (=);;
- : 'a -> 'a -> bool = <fun>
# let g = (=) 3;;
val g: int -> bool = <fun>
# g 8;;
- : bool = false
# (<=);;
- : 'a -> 'a -> bool = <fun>
# let positivo = (<=) 0;;
val positivo : int -> bool = <fun>
# positivo 5;;
- : bool = true
```

FUNCIÓN MAX Y MIN

```
# max 3 7;;
- : int = 7
# min 3 7;;
- : int = 3
```

FUNCIÓN MAX Y MIN SIN PREDEFINIR

```
# let max = function x -> function y -> if x >= y then x else y;;
```

- (mejor forma de escribirlo:)

```
# let max x = function y -> if x >= y then x else y;;
```

- (la mejor forma de escribirlo:)

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

PRODUCTOS CARTESIANO

```
# (true, ()), "falso trio";;      → al tener un paréntesis, es un par
# let p = (true, ()), "falso trio";;
val p : (bool * unit) * string = ((true, ()), "falso trio")
# true, (), "falso trio";;        → ahora al no llevar paréntesis, es un trío
# let t = true, (), "falso trio";;
val t : bool * unit * string = (true, (), "falso trio");;
```

FUNCIÓN FST

- Devuelve el primer elemento del par

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst (8,5);;
- : int = 8
```

FUNCIÓN FST SIN PREDEFINIR

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

- (mejor forma:)

```
# let fst (x, _) = x;;   → la barra "_" es un comodín, puede ir cualquier cosa en su
                           lugar
```

FUNCIÓN FACTORIAL RECURSIVA

```
# let rec fact n =
        if n = 0 then 1
        else n * fact (n-1);;
val fact : int -> int = <fun>
# let g x y = x + y, x * y;;
val g : int -> int -> int * int = <fun>
# g 2 3;;
- : int * int = (5, 6)
# let g2 (x,y) = x + y, x * y;;
val g2 : int * int -> int * int = <fun>
# g2 (2,3);;
- : int * int = (5, 6)
```

IMPLEMENTACIÓN (SENCILLA) COCIENTE

```
# let rec quo x y =   → ( x >= 0, y > 0 )
        if x < y then 0
        else 1 + quo (x-y) y;;
# quo 7 3;;
- : int = 2;;
# quo 7 0;;    → ( si y=0 entra en un bucle infinito y se llena el stack )
Stack overflow during evaluation (looping recursion?).
```

IMPLEMENTACIÓN (SENCILLA) RESTO

```
# let rec rem x y =   → (* x >= 0, y > 0 *)
        if x < y then x
        else rem (x-y) y;;
# rem 17 3;;
- : int = 2
```

- La función rem no puede producir overflow porque no deja cuentas pendientes
- La recursividad que no produce overflow se llama recursividad terminal

IMPLEMENTACIÓN DIVISIÓN (USANDO QUO Y REM)

```
# let div x y = quo x y, rem x y;;
```

IMPLEMENTACIÓN DIVISIÓN (SIN USAR QUO Y REM)

- Quo y rem se implementan asi igual, por lo que se puede ahorrar tiempo si se implementan a la vez

```
# let rec div x y =
        if x < y then 0, x
        else 1 + fst (div(x-y) y), snd (div(x-y) y);;
```

→ (esto es poco eficiente porque calcula 2 veces div(x-y) y por lo que es muy poco eficiente. NUNCA calcular 2 veces lo mismo)

- (de esta forma es mucho más eficiente:)

```
# let rec div x y =
        if x < y then 0, x
        else let p = div(x-y) y in
                1 + fst p, snd p;;
```

- (de esta forma además de eficiente queda mucho mejor escrito:)

```
# let rec div x y =
        if x < y then 0, x
        else let q,r = div(x-y) y in
                1 + q, r;;
```

COMPARADORES

- Igualdad → “=” / “<>”
  - “11” = “1” ^ “1”
    - : bool = true
- Identidad → “==” / “!=”
  - “11” == “1” ^ “1”
    - : bool = false

IMPLEMENTACIÓN RECURSIVA FIBONACCI

```
# let rec fib n = if n <= 1 then n
        else fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>
```

- Ej. Crear una función fibonacci que devuelva fibonacci y su anterior (pares)

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

FUNCIÓN PARA CRONOMETRAR

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

- (otra forma de escribir la función crono de arriba:)

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

- Cada vez que se utiliza "Sys.time ()" devuelve el tiempo que tarda el proceso anterior (o la sumo de todos los procesos y funciones anteriores) en la cpu

LISTAS

- char list = ['a'];; != char 'a' → (no son del mismo tipo)
- [];; → lista vacía 'a list (puede ser lista de enteros, de char…)
- List.hd → función polimórfica de 'a list en 'a
- List.tl de 'a list a 'a list → otra lista que resultaría si la primera no tuviese el primer elemento
- int list;; y List.tl list → permiten obtener cualquier información de la lista (int list devuelve el contenido y List.tl devuelve el contenido de la lista menos el 1º elemento)
    -
```
# let list = [1;2;3;100];;
val list : int list = [1; 2; 3; 100]
# int list;;
- : int list = [1; 2; 3; 100]
# List.tl list;;
- : int list = [2; 3; 100]
```

- List.init 27 (function n -> Char.chr (65 + n));; → va creando una lista de las 27 letras del abecedario (la letra 'A' corresponde al caracter 65
    -
```
# List.init 27 (function n -> Char.chr (65+n));;
- : char list =
['A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J'; 'K'; 'L'; 'M'; 'N'; 'O';
 'P'; 'Q'; 'R'; 'S'; 'T'; 'U'; 'V'; 'W'; 'X'; 'Y'; 'Z'; '[']
```

FUNCIÓN LENGTH

- Si la función length no estuviera en el módulo list, se podría definir solo usando List.hd y List.tl

```
# let rec length l = if l <> [] then 1 + length(List.tl l) else 0;;
```


SUBRAYA ESTO: CÓMETE UN TACO.

EXCEPTION

- Si aparece la palabra "exception" significa que la ejecución se ha interrumpido

IMPLEMENTACIÓN FUNCIÓN HD

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

FUNCIÓN NTH

- Devuelve el elemento de la lista en la posición que se indica
  - # List.nth;;
    - : 'a list -> int -> 'a = <fun>
  - # let l2 = ['a';'e';'i';'o'];;
    val l2 : char list = ['a'; 'e'; 'i'; 'o']
    # l2;;
    - : char list = ['a'; 'e'; 'i'; 'o']
    # List.nth l2 2;;
    - : char = 'i':
    # List.nth l2 4;;
    Exception: Failure "nth" → (no existe la posición 4)

IMPLEMENTACIÓN FUNCIÓN NTH

```
# let rec nth l n =
        if n < 0 then raise (Invalid_argument "nth")
        else if l = [] then raise (Failure "nth")
        else if n = 0 then List.hd l
        else nth (List.tl l) (n-1);;
val nth : 'a list -> int -> 'a = <fun>
# nth ['a';'e';'i';] 2;;
- : char = 'i'
# nth ['a';'e';'i';] 3;;
Exception: Failure "hd".
```

- (otra versión de la función nth pero con patrones en vez de con if then else :)

```
# let rec nth l n = match (l,n) with
        [], _ -> raise (Failure "nth")
      | h::_, 0 -> h
      | _::t, _ -> nth t (n-1);;
```

- (mejor versión de nth que comprueba 1 sola vez si es negativo:)

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

IMPLEMENTACIÓN FUNCIÓN APPEND

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

IMPLEMENTACIÓN FUNCIÓN COCIENTE

```
# let rec quo x y =
        if x < y then 0
        else 1 + quo (x-y) y;;
```

IMPLEMENTACIÓN FUNCIÓN LAST

```
# let rec last = function
        h::[] -> h
      | _::t -> last t;;
```

FUNCIÓN AUX

- Calcula una especie de contador con una lista y un entero

```
# let rec aux i = function
        [] -> i
      | _::t -> aux(i + 1) t;;
```

- (otra forma de escribirlo:)

```
# let rec aux i l = match l with
        [] -> i
      | _::t -> aux(i + 1) t;;
```

IMPLEMENTACIÓN FUNCIÓN LENGTH

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

IMPLEMENTACIÓN FACTORIAL RECURSIVA

```
# let rec fact = function
        0 -> 1
      | n -> n * fact(n-1);;
# let rec aux p i =
        if i = 0 then p
        else aux (p * i) (i-1)
# let fact n = aux 1 n;;
```

IMPLEMENTACIÓN FACTORIAL CON RECURSIVIDAD TERMINAL

```
# let fact n =
        let rec aux p = function
                0 -> p
              | i -> aux (p * i) (i-1)
        in aux 1 n;;
```

IMPLEMENTACIÓN FIB RECURSIVA

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

IMPLEMENTACIÓN FIB RECURSIVA TERMINAL

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

FUNCIÓN LMAX

- Función lo más sencilla posible que devuelva el mayor elemento de una lista
  - lmax = 'a -> 'a

```
# let rec lmax = function
        [] -> raise (Failure "lmax")
      | h::[] -> h        →       si no devolviese h al tener la cola vacía daría un error
      | h::t -> let m = lmax t in
                if h >= m then h else m;;
```

- (otra definición:)

```
# let rec lmax = function
        [] -> raise (Failure "lmax")
      | h::[] -> h
      | h::t -> max h (lmax t);;        →       no es recursiva terminal porque la última operación no es lmax
```

FUNCIÓN LMAX RECURSIVA TERMINAL

```
# let lmax = function
        [] -> raise (Failure "lmax")
      | h::t -> let rec aux m r =match r with
                [] -> m
              | h::t -> aux (max m h) t
        in aux h t;;    →       h es el primer elemento de la lista y t el último
```

```
# lmax [1;7;3]
aux 1 [7;3]    →       pasos que sigue la operación
aux 7 [3]
aux 7 []
7
# lmax [1;7;3]
- : int = 7
# let l1 = List.init 300000 abs;;
val l1 : int list =
  [0; 1; 2; 3; 4; 5; ...]
# lmax l1;;
- : int = 299999
```

- (otra función lmax recursiva terminal:)

```
# let rec lmax = function
        [] -> raise (Failure "lmax")
      | h::[] -> h
      | h1::h2::t -> lmax (max h1 h2 :: t);;
```

→ lista sin el más pequeño de los 2 primeros, un poco más ineficiente porque crea una lista en cada paso

FUNCIÓN LAST RECURSIVA TERMINAL

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

FUNCIÓN APPEND

- Concatena 2 listas

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

FUNCIÓN REV_APPEND

- Invierte l1 y lo concatena a l2

```
# let rec rev_append l1 l2 =match l1 with
        [] -> l2
      | h::t -> rev_append t (h::l2);;
```

FUNCIÓN REV

- Invierte la lista

```
# let rec rev = function
        [] -> []
      | h::t -> append (rev t) [h];;
```

→ intentar no implementarla con append porque es una operación muy costosa y tarda demasiado, se hacen tantos appends como elementos tenga la lista, es decir, crece como n^2

FUNCIÓN REV RECURSIVA TERMINAL

```
# let rev l = List.fold_left (fun t h -> h::t) [] t;;
```

FUNCIÓN FOLD_LEFT RECURSIVA TERMINAL

```
# let rec fold_left f e l = match l with
        [] -> e
      | h::t -> fold_left f (f e h) t;;
```

FUNCIÓN QUE DEVUELVA LA SUMA DE LOS ELEMENTOS DE LA LISTA

```
# let sum l = List.fold_left (+) 0 l;;
```

FUNCIÓN QUE DEVUELVA EL NÚMERO DE ELEMENTOS DE LA LISTA

```
# let length l = List.fold_left (function s -> function _ -> s + 1) 0 l;;
```

NOTACIÓN FUNCTION Y FUN

- con la notación "function" solo se puede poner 1 argumento pero varias reglas
  - function <x> -> function <y> -> <e>
- con la notación "fun" solo se pone 1 flecha pero se pueden poner varios argumentos
  - fun <x> <y> -> <e>

FUNCIÓN QUE AÑADA UN ELEMENTO A UNA LISTA

- coge una lista t y un elemento h y devuelve la lista con cabeza h y cola t

```
# List.fold_left (fun t h -> h::t) [] l;;
```

FUNCIÓN LENGTH USANDO FUN

```
# let length l = List.fold_left (fun s _ -> s + 1) 0 l;;
```

FUNCIÓN FOR_ALL RECURSIVA TERMINAL

```
# let rec for_all p l = match l with
        [] -> true
      | h::t -> p h && for_all p t;;
```

FUNCIÓN FOR_ALL CON FOLD_LEFT

```
# let for_all p l =
      List.fold_left (fun b x -> b && p x) true l;;
```

→ no es conveniente usar el fold_left para definirla

FUNCIÓN INSERT

- Inserta elementos en una lista ordenada

```
# let rec insert x l = match l with
        [] -> [x]
      | h::t -> if x <= h then x::l
                else h :: insert x l;;
```

FUNCIÓN INSERT RECURSIVA TERMINAL

```
# let insert' x l =
      let rec aux p1 p2 = match p1 with
            [] -> List.rev (x::p2)
          | h::t -> if x <= h then List.rev_append p2 (x::p1)
                    else aux t (h::p2)
      in aux l [];;
```

ALGORITMO ORDENACIÓN

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

```
let isort' l =
        let rec aux p1 p2 = match p1 with
                [] -> p2
              | h::t -> aux t (insert' h p2)
        in aux l [];;
# let big = List.init 1_000_000 (fun x -> -x);;
val big : int list =
[0; -1; -2; -3; -4; -5; ... ]
# isort' big;;
- : int list =
[-999999; -999998; -999997; -999996; -999995; ... ]
let rec insert_g ord x = function
        [] -> [x]
      | h::t -> if ord x h then x::h::t
                else h::insert_g ord x t;;
let isort_g ord = function
        [] -> []
      | h::t -> insert_g ord h;;
# isort_g (<=) [2;0;9;7;18];;
- :  int list [0; 2; 7; 9; 18]
# isort_g (>=) [2;0;9;7;18];;
- : int list [18; 9; 7; 2; 0]
```

FUNCIÓN PARA DIVIDIR

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

FUNCIÓN PARA REPARTIR (FUSIÓN)

```
let rec merge function
        ([], l) | (l, []) -> l
      | (h1::t1, h2::t2) -> if h1 <= h2 then h1::merge (t1, h2::t2)
                else h2::merge (h1::t1, t2);;
let rec msort l = match l with        → no terminal, la recursividad es log base 2 de 10
        [] | _::[] -> l
      | _ -> let l1, l2 = divide l in     → divide devuelve 2 listas
                merge (msort l1, msort l2);;          → con esto salen ordenadas
```

FUNCIÓN QUEEN

```
let come (i1, j1) (i2, j2) =
        j1 = i2 ||
        j1 = j2 ||
        abs (i1 - i2) = abs (j1 - j2);;
let compatible p l =
        not (List.exists (come p) l);;
```


```
let queen n =
        let rec completa promesa (i,j) =
                if i > n then Some promesa
                else if j > n then None
                else if compatible (i,j) promesa
                        then match completa ((i,j)::promesa) (i + 1, 1) with
                                None -> completa promesa (i, j + 1)
                              | s -> s
                        else completa promesa (i, j + 1)
        in completa [] (1,1);;
let find_opt p l = try Some (List.find p l) with
                        Not_found -> None;;
```

- (otra forma de definir queen:)

```
 let queen n =
        let rec completa promesa (i,j) =
                if i > n then [promesa]
                else if j > n then raise []
                else if compatible (i,j) promesa
                        then completa ((i,j)::promesa) (i + 1, 1) @
                                completa promesa (i, j + 1)
                        else completa promesa (i, j + 1)
                in completa [] (1,1);;
```

- imprimir soluciones:

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

- imprimir todas las reinas:

```
let print_all_queen n =
        let rec completa promesa (i,j) =
                if i > n then print_sol promesa
                else if j > n then raise []
                else if compatible (i,j) promesa
                        then (completa ((i,j)::promesa) (i+1,1);
                                completa promesa (i,j+1))
                        else completa promesa (i,j+1)
                in completa [] (1,1);;
        print_all_queen (int_of_string (Sys.argv.(1)));;
```

GENERAR LISTAS ALEATORIAS

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

DEFINICIÓN TIPOS DE DATOS

```
type maybeAnInt =
        NotAnInt
      | AnInt of int;;
```

```
# NotAnInt;;
- : maybeAnInt = NotAnInt
# AnInt 1;;
- : maybeAnInt = AnInt 1;;
# AnInt 3 = AnInt 2;;
- : bool = false;;
# NotAnInt = NotAnInt;;
- : bool = true;;
```

FUNCIÓN COCIENTE

```
let quo x y = match x, y with
        _, AnInt 0 -> NotAnInt
      | AnInt m, AnInt n -> AnInt (m / n)
      | _ -> NotAnInt;;
# quo (AnInt 3) (AnInt 2);;
- : maybeAnInt = AnInt 1
# quo (AnInt 3) NotAnInt;;
- : maybeAnInt = NotAnINt
```

CONSTRUCTORES CONSTANTES

- Si solo se utilizan constructores constantes tendrías tantos tipos como constructores y serían finitos

DEFINICIÓN TIPO PALO CARTA

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

DEFINICIÓN TIPO BOOLEAN

```
type boolean = T | F;;
```

- conjunción para tipo boolean:

```
let conj a b = match a, b with
                F, _ -> F
              | _, F -> F
              | _ -> T;;
```

- los nombres de valores van con minúscula y los nombres de patrones con mayúscula

```
# let verdadero = T;;
# let falso = F;;
```

- si se define conj asi (↓), solo devuelve el boolean de la primera regla, la 2ª y 3ª regla no las usa

```
let conj a b = match a,b with
        falso, _ -> falso
      | _, falso -> falso
      | _ -> verdadero;;
```

DEFINICIÓN TIPO CONSTRUCTOR

```
type tipo = Con of int;;
```

DEFINICIÓN TIPO IZQUIERDA / DERECHA

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

DEFINICIÓN TIPO NÚMERO

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

DEFINICIÓN TIPOS DE DATOS

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

DEFINICIÓN TIPO NATURAL

```
type nat =
        Z | S of nat;;
# Z;;
- : nat = Z
# S Z;;
- : nat = S Z
# S (S Z);;
- : nat = S (S Z)
# S Z = S (S Z);;
- : bool = false
```

FUNCIÓN SUMA

```
let rec sum n1 = function
        Z -> n1
      | S n2 -> sum(S n1) n2;;
# let cero = Z;;
# let uno = S cero;;
# let dos = S uno;;
# let tres = sum uno dos;;
val tres : nat = S (S (S Z))
# let seis = sum tres tres;;
val seis : nat = S (S (S (S (S Z))))
```

FUNCIÓN PASAR DE INT A NAT

```
let rec nat_of_int = function
        0 -> Z
      | n -> S (nat_of_int (n-1));;
```

- (otra definición:)

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

DEFINICIÓN ÁRBOL

```
type 'a btree =
        E | N of 'a * 'a btree * 'a btree;;
```

- valores de tipo btree:

```
# E;;
- : 'a btree = E
# N (1,E,E);;
- : int btree = N (2,E,E)
# let h x = N (x,E,E);;
val h : 'a -> 'a btree = <fun>
```

Nº NODOS DE UN ÁRBOL

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

ALTURA DE UN ÁRBOL

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

RECORRIDO PREORDEN

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

HOJAS DE UN ÁRBOL

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

DEFINICIONES ÁRBOLES

```
type 'a btree =
        E | N of 'a * 'a btree;;
```

- árboles estrictamente binarios (no se pueden representar ramas vacías ↓)

```
type 'a st_tree =
        Leaf of 'a
      | Node of 'a * 'a st_tree * 'a st_tree;;
type 'a btree =
        E | N of 'a * 'a btree * 'a btree;;
```

→ esta definición de árbol es muy restrictiva porque solo representa árboles con 2 hijos o vacíos, no valen árboles con 1 hijo


```
# let h x = N(x, E, E);;
val h : 'a -> 'a btree = <fun>
# let t6 = N (6, h 5, h 11);;
val t6 : int btree = N (6, N (5, E, E), N (11, E, E))
# let t9 = N (9, h 4, E);;
val t9 : int btree = N (9, N (4, E, E), E)    →    árbol con raíz 9, rama izquierda 4 y
        rama derecha vacía
# let t7 = N (7, h 2, t6);;
val t7 : int btree = N (7, N (2, E, E), N (6, N (5, E, E), N (11, E, E)))
# let t5 = N (5, E, t9);;
val t5 : int btree = N (5, E, N (9, N (4, E, E), E))
# let t = N (2, t7, t9);;
val t : int btree = N (2, N (7, N (2, E, E), N (6, N (5, E, E), N (11, E, E))), N (9, N (4, E, E),
E))
```

ÁRBOL ESPECULAR

- polimórfica de 'a st_tree a 'a st_tree (intercambia derecha con izquierda)

```
let rec mirror = function
        Leaf v -> Leaf v      →      si solo tiene raíz no cambia
      | Node (v, l, r) -> Node (v, mirror r, mirror l);;
```

TRANSFORMACIÓN ÁRBOLES

- pasa de un árbol binario a un árbol estrictamente binario

```
let rec b_of_st = function          →      no falla nunca
        Leaf v -> N (v, E, E)
      | Node (v, l, r) -> N (v, b_of_st l, b_of_st r);;
```

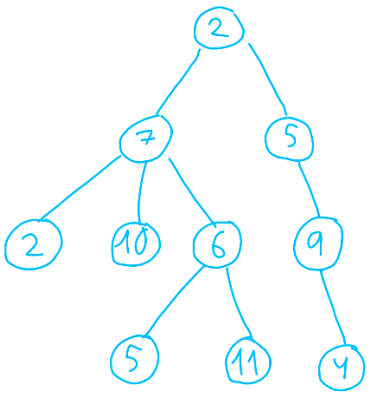
- pasa de un árbol estrictamente binario a un árbol binario

```
let rec st_of_b = function          →      falla a veces
        E -> raise (Invalid_argument "st_of_b")
      | N (v, E, E) -> Leaf v
      | N (v, l, r) -> Node (v, st_of_b l, st_of_b r);;
```

ÁRBOLES NO BINARIOS

- hay nodos que pueden tener más de 2 hijos

```
type 'a gtree =
        GT of 'a * 'a gtree list;;      →      GT no permite
                poner árboles vacíos, lo más
                pequeño es una raíz y el resto vacío
# let root x = GT (x, []);;             →      (dibujo →)
# let t9 = GT (9, [root 4]);;
# let t5 = GT (5, [t9]);;
```



FUNCIÓN CALCULAR Nº NODOS

```
let rec nngt = function
        GT (_, l) -> List.folf_left(+) 1 (List.map nngt l);;
```

- (misma función pero pasando GT en el nombre:)
  ```
  let rec nngt (GT (_, l)) =
          List.fold_left (+) 1 (List.map nngt l);;
  ```
- (misma función pero más sencilla:)
  ```
  let rec nngt = function
          GT (_, []) -> 1
        | GT (v, h::t) -> nngt h + nngt (GT(v, t));;
  ```

ENTRADA / SALIDA (PARTE IMPERATIVA)

- trataremos con dispositivos de entrada salida secuenciales (1 byte siempre, tipo char en ocaml)
- comando (con salida) > nombre archivo → la salida va al archivo
- output_char;; → en qué dispositivo queremos escribir y qué queremos escribir
- stdout;; → valor de tipo out channel que representa la salida estándar
  ```
  # output_char stdout 'X';;
  ```
  → se muestra la X
  ```
  X- : unit = ()
  # let print_char c = output_char stdout 'X';;
  val print_char : 'a -> unit = <fun>
  # print_char 's';;
  ```
  → siempre va a imprimir X, da igual q letra vaya después de print_char
  ```
  X- : unit = ()
  ```
- mostrar X seguido de Y
  ```
  # let _ = print_char 'X' in print_char 'Y';;
  XY- : unit = ()
  # print_char 'X'; print_char 'Y';;
  XY- : unit = ()
  ```
- si se ponen operaciones como "a" ^ "b"; 2*3;; el compilador lanza un warning, para que no pase eso se ignora el primer resultado
  ```
  # let ignore _ = ();;
  val ignore : 'a -> unit = <fun>
  # ignore("a" ^ "b"); 2*3;;
  - : int = 6
  ```

DEFINIR OUTPUT STRING

- String.get "Hola" 2;; == "Hola".[2];;
  ```
  # String.get "Hola" 2;;
  - : char = 'l'
  # "Hola".[2];;
  - : char = 'l'
  ```


SUBRAYA ESTO: CÓMETE UN TACO.

```
let output_string c s =
        let n = String.length s in
        let rec loop i =
                if i >= n then ()
                else output_char c s.[i];
                loop(i+1)
        in loop 0;;
```
→ esto es un bucle infinito tipo out_channel -> string -> 'a, no devuelve unit, devuelve 'a → el " ; " es más débil que el if then else (el bucle se hace pero se sigue incrementando)

- se añaden paréntesis después del "else" (begin end) para arreglar la definición

```
let output_string c s =
        let n = String.length s in
        let rec loop i =
                if i >= n then ()
                else (output_char c s.[i];
                loop(i+1))
        in loop 0;;
let output_string c s =
        let n = String.length s in
        let rec loop i =
                if i >= n then ()
                else begin
                        output_char c s.[i];
                        loop(i+1)
                        end
        in loop 0;;
# let print_string s = output_string stdout s;;
```
→ todas están definidas en el módulo stdlib

```
val print_string : string -> unit = <fun>
# let print_endline s = print_string (s ^ "\n");;
val print_endline : string -> unit = <fun>
# let print_newline () = print_endline "";;
val print_newline : unit -> unit = <fun>
# let print_int n = print_string (string_of_int n);;
val print_int : int -> unit = <fun>
# let print_float x = print_string (string_of_float x);;
val print_float : float -> unit = <fun>
```

ESCRIBIR EN UN ARCHIVO EN DISCO

- open_out;; → devuelve un canal de salida, para un archivo de disco será el nombre del archivo con su ruta

```
# open_out;;
- : string -> out_channel = <fun>
```

let sal = open_out "pru";; → crea un archivo pru (borra el archivo si ya existía)

val sal : out_channel = <abstr>

- close_out sal;; → close_out se utiliza para cerrar
- flush;; → se pide al SO que vuelque la salida

let sal = open_out "pru";;
val sal : out_channel = <abstr>
output_string sal "ABC";;
- : unit = ()
output_string sal;;
- : string -> unit = <fun>
let sal = open_out "pru";;
val sal : out_channel = <abstr>
output_string sal "ABC";;
- : unit = ()
flush sal;;
- : unit = ()
output_string sal "XYZ\n";;
- : unit = ()
flush sal;;
- : unit = ()
close_out sal;;
- : unit = ()

- input_char;; → función para leer (tipo in_channel -> char)
- let ent = open_in “pru”;; → el archivo “pru” tiene que existir y mi programa tiene que tener permiso para leerlo, sino Sys.error
- input_char ent;; → la lectura / escritura son secuenciales, devolverá el primer caracter → a medida que se ejecuta irá devolviendo ‘A’ ; ‘B’ ; ‘C’ ; ‘D’ ; \n ; ‘X’ ; ‘Y’ ; ‘Z’ ; \n ; Exception: End_of_file
- sdin;; → devolverá la letra que le envíe por el teclado

input_char stdin;; → la entrada estándar funciona en base a líneas, hay que darle a ENTER

- input_line;; → se usa para que no de error

input_line stdin;;
XYZ
- : string = “XYZ”

- let read_line () = input_line stdin;; → ya definida
- let print_endline s = print_string (s ^ "\n");
  flush stdout;; → al volcar el buffer si es correcta la definición (antes no lo era)
- let read_line () = flush stdout;
  input_line stdin;; → antes de leer la entrada estándar, vuelca la salida

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi lado.

Siempres me has ayudado
Cuando por exámenes me he agobiado

Oh Wuolah wuolitah
Tu que eres tan bonita

IMPLEMENTACIÓN FUNCIONES ENTRADA / SALIDA

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

- (otra forma de escribirlo:)

```
let output_string_list out l =
        List.iter (fun s -> output_string out (s ^ "\n")) l;;
let rec input_string_list input =
        try     →     la función input_line solo devuelve la 1ª linea
                let s = input_line input in
                s :: input_string_list input
        with End_of_file -> [];;
```

ÁRBOLES (USANDO ENTRADA / SALIDA)

```
type 'a btree =
        E
      | N of 'a * 'a btree * 'a btree;;
let h x = N(x, E, E);;
let t = N (0, N (1, h 2, h 3), h 4);;
# let salida = open_out "arboles";;
val salida : out_channel = <abstr>
# output_value salida t;;
- : unit = ()
# output_value salida (h 100);;
- : unit = ()
# close_out salida;;
- : unit = ()
# let entrada = open_in "arboles";;
val entrada : in_channel = <abstr>
# let (t2: int btree) = input_value entrada;;
val t2 : int btree = N (0, N (1, N (2, E, E), N (3, E, E)), N (4, E, E))
```

REFERENCIAS

```
let ref;;
- : 'a -> 'a ref = <fun>
let ref 3;;
- : int ref = {contents = 3}
```

- acceder al valor de una referencia:

```
# (!);;
- : 'a ref -> 'a = <fun>
# let i = ref 0;;
val i : int ref =  {contents = 0}
# !i;;
- : int = 0
```

- modificar el valor de una referencia:

```
# (:=);;
- : 'a ref -> 'a -> unit = <fun>
# i := 100;;
- : unit = ()
# !i;;
- : int = 100
```

FUNCIÓN FACTORIAL

- utilizando ":=" , "!" y un bucle for

```
let fact n =
        let p = ref 1 in
        for i = 1 to n do
                p := !p * i
        done;
        !p;;
# for i = 0 to 10 do
        print_int (fact i);
        print_newline()
done;;
1
1
2
6
24
120
720
5040
40320
362880
3628800
- : unit = ()
```

BUCLE FOR

```
<i1>: int → m
<i2>: int → n
<e>: 'a
for <i> = <i1> to <i2> do
        <e>
done
m = 3
n = 5
let <i> = 3 in <e>
let <i> = 4 in <e>      → va desde 3 hasta 5 realizando al operación (<e>)
let <i> = 5 in <e>
```

BUCLE WHILE

```
<b>: bool
<e>: 'a
(while <b> do
        <e>
done): unit
```

FUNCIÓN FACTORIAL CON BUCLE 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;;
```

FUNCIÓN TURNO

- por ejemplo, los turnos de los puestos en un mercado

```
let n = ref 0;;
let turno () =
        n := !n + 1;
        !n;;
let reset () =
        n := 0;;
let turno =
        let n = ref 0 in
        function () -> n := !n + 1;
                        !n;;
```

- (otra forma de escribirlo, todo una sola función:)

```
let turno, reset =
        let n = ref 0 in
        (fun () -> n := !n + 1; !n),
        (fun () -> n := 0);;
```

MÓDULOS

```
module Counter () : sig
        val turno: unit -> int          →       signatura (interfaz) (mli)
        val reset: unit -> unit
end = struct
        let n = ref 0           →       estructura (implementación) (ml)
        let turno () =
                n := !n + 1;
                !n
        let reset () =
                n:= 0
end;;
```

- (Counter por sí solo no es un módulo)
  ```
  # module C1 = Counter ();;
  module C1 : sig val turno : unit -> int val reset : unit -> unit end
  # module FrutaC = Counter();;
  module FrutaC : sig val turno : unit -> int val reset : unit -> unit end
  # FrutaC.turno();;
  - : int = 1
  # FrutaC.turno();;
  - : int = 2
  # C1.turno();;
  - : int = 1
  ```

FUNCIÓN COMPARE

```
module IntPair =    →    tiene que tener un tipo t y una función compare
struct
        type t = int * int
        let compare = Stdlib.compare
end;;
module IPSet = Set. make(IntPair)    →    función make en el módulo set
        (set.make)    →    lo usaremos para representar ABB para pares de enteros
```

- Int Pair SET conjuntos de valores de tipo t de int pair → genera un módulo
- respuesta del compilar → tengo un modelo IPSet que tiene accesible todo lo que tiene en su signatura
  ```
  let trees = List.init 50_000 (fun _ -> 1 + Random.int 500, 1 + Random.int 500)    →
          casillas aleatorias de un tablero 500 * 500
  ```

- generar el árbol:
  ```
  let trees_S = List.fold_left (fun s p -> IPSet.add p s) IPSet.empty trees;;
  ```
- generar el árbol con of_list es más rápido que con add
  ```
  let trees_S = IPSet.of_list trees;;
  # IPSet.mem (424, 299) trees_S;;
  ```
- generar una lista de 5000 pares:
  ```
  let to_find = List.init 5000 (fun _ -> 1 + Random.int 500, 1 + Random.int 500)
  ```
- buscar elementos en listas:
  ```
  let r1 = List.filter (fun p -> List.mm p trees) to_find;;
  ```
- buscar elementos en un árbol binario:
  ```
  let r2 = List.filter (fun p -> IPSet.mem p trees_S) to_find;;
  ```
- r2 es más rápido que r1, y cuantos más elementos haya en la lista más se va a notar la diferencia
  ```
  let crono f x =    →    función cronómetro para ver si tarda más r1 o r2
          let t = Sys.time () in
          f x; Sys.time () -. t;;
  ```

VECTORES / ARRAYS

- los arrays se crean igual que las listas pero añadiendo " | " después y antes de " [ ] "
- así se crean las listas:

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

- así se crean los arrays:

```
# [|1;2;3|];;
- : int array = [|1; 2; 3|]
# let v = [|1;2;3|];;
val v : int array = [|1; 2; 3|]
# v.(1);;        →   devuelve el valor de la posición 1 del vector v
- : int = 2
# Array.init 1000 (fun _ -> 0.0);;        →   crea un vector con 1000 ceros
```

VECTORES FLOTANTES

```
let sprod v1 v2 =      →   multiplica v1 * v2
    if Array.length v1 <> Array.length v2
        then (raise (Invalid_argument "sprod"))      →   lanza una
            excepción si los vectores v1 y v2 tienen longitudes diferentes
    else begin
        let p = ref 0.0 in
        for i = 0 to Array.length v1 - 1 do
            p := !p +. v1.(i) *. v2.(i)      →   la exclamación devolvía el
                valor de p (porque p es una variable no un float)
        done;
        !p
    end;;
# sprod [|1.; 2.; 3.|] [|0.; 1.5;2.|];;
- : float = 9.
```

- (otra implementación de la función sprod:)

```
let sprod v1 v2 =
    try
        Array.fold_left (+.) 0.0 (Array.map2 ( *. ) v1 v2)
    with Invalid_argument _ -> raise (Invalid_argument "sprod");;      →   la
        excepción sirve para los vectores con diferente longitud como el
        invalid_argument de la función sprod anterior
```

STRUCTS / REGISTROS

```
type persona = {nombre: string; edad: int};;
    # let pepe = {nombre = "Pepe"; edad = 57};;      →   le da valores
    val pepe : persona = {nombre = "Pepe"; edad = 57}
    # let maria = {edad = 21; nombre = "María"};;
    val maria : persona = {nombre = "María"; edad = 21}
    # maria.edad;;      →   devuelve el contenido del campo edad
    - : int = 21
```

```
let mas_viejo p =    →    devuelve la persona que se escriba pero con 1 año más
    {nombre = p.nombre; edad = p.edad + 1};;
    # mas_viejo pepe;;
    - : persona = {nombre = "Pepe"; edad = 58}
type persona = {nombre: string; mutable edad: int};;    →    la palabra
    mutable permite modificar el valor
    # maria.edad <- 19;;
    - : unit = ()
    # maria.edad;;
    - : int = 19
let envejece p =
    p.edad <- p.edad + 1;;
let p = ref 0.0;;
    # p.contents;;
    - : float = 0.;;
    # p.contents <- 100.;;
    - : unit = ()
    # p.contents;;
    - : float = 100.
```

DEFINICIONES ALFA REF

```
type 'a ref = {mutable contents: 'a};;
let {} v = v.contents;;
let {:=} v x = v.contents <- x;;
```

ORIENTACIÓN A OBJETOS

```
let c = object
    val mutable n = 0
    method turno =
        n <- n + 1;
        n
    method reset =    →    modifica el valor del objeto
        n <- 0
end;;
val c : < reset : unit; turno : int > = <obj>    →    el tipo de un objeto depende
    de los métodos y del tipo de estos
    # c#turno;;
    - : int = 1
    # c#turno + c#turno;;
    - : int = 5
    #  c#turno + (c#reset; c#turno);;
    - : int = 3
let doble c =
    2 * c#turno;;    →    se aplica a cualquier tipo de objeto con tal de que
        tenga un método turno de tipo int
```

```
        # doble c;;
        - : int = 6
        # c#turno;;
        - : int = 4
let c2 = object
        method turno = 2 * c#turno
        method reset = c#reset
end;;          →    devuelve un objeto que tiene un método turno de tipo int y un
                    método reset del mismo tipo que el método reset del argumento
        # c2#turno;;
        - : int = 10
        # c2#turno;;
        - : int = 12
        # c2#turno;;
        - : int = 14
class counter = object
        val mutable n = 0
        method turno =
                n <- n + 1;
                n
        method reset =
                n <- 0
end;;
type counter = <turno: int; reset: unit>;;
class counter_w_set = object
        inherit counter as super
        method set i =
                n <- i
end;;
class counter_wSet = object      →    otra definición similar a counter_w_set
        inherit counter
        method set ini = n <- ini
end;;
class counter_wInit n0 = object (this)
        inherit counter_wSet
        initializer this#set n0
end;;
        # let c = new counter_wInit 27;;
        val c : counter_wInit = <obj>
        # c#next;;
        - : int = 28
```

```
class counter_wMax n0 max = object (this)      →      otra clase counter para que los
                contadores tengan un valor máximo (como los turnos en un
                supermercado, que al llegar a 100 reinician)
        inherit counter_wInit n0 as super      →      super permite invocar los
                métodos que hereda de la clase counter_wInit (superclase)
        method reset = this#set n0
        method next =
                let nx = super#next in
                if nx <= max then nx
                else (this#reset; super#next)
end;;
        # let cm = new counter_wMax 8 12;;
        val cm : counter_wMax = <obj>
        # cm#next;;
        - : int = 9
        # cm#next;;
        - : int = 10
        # cm#next;;
        - : int = 11
        # cm#next;;
        - : int = 12
```

COLAS (FIFO)

```
class ['a] queue = object (self)      →      self es lo mismo que this
        val mutable front = []
        val mutable back = []      →      back es la cola de la lista, que se coloca al
                        revés para que sea más cómodo añadir elementos, es decir la
                lista está formada por 2 colas posicionadas (cabeza-cola-cola-cabeza)
        method enqueue (e: 'a) =
                back <- e::back
        method dequeue = match front, back with
                h::t, _ -> front <- t; Some h
              | [], [] -> None
              | [], _ -> front <- List.rev back;
                        back <- [];
                        self#dequeue
end;;
        # let q = new queue;
        val q : '_weak1 queue = <obj>      →      '_weak1 significa que el tipo de la
                cola es desconocido (no se sabe si es una cola de int, string, char...)
        # q#enqueue 'a';;
        - : unit = ()
        # q;;
        - : char queue = <obj>
```

```
let addlist l q =          →      añadir una lista de valores a una cola q (para no tener
                                  que añadir valor a valor)
        List.iter (fun e -> q#enqueue e) l;;          →      List.iter aplica una función a
                                  cada uno de los valor de la lista
        # addlist ['e';'i';'o';'u'] q;;
        - : unit = ()
        # q#dequeue;;
        - : char option = Some 'a'
let rec drain q =          →      drenar / vaciar la cola
        match q#dequeue with      →      se aplica hasta que q devuelva None (es
                                  decir, que esté vacía)
                None -> []
              | _ -> drain q;;
```