resumenteoriabibliaPP.pdf

imluciacosta

Paradigmas de Programación

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

Paradigmas de la Programación

Programación en OCaml

1. **Expresiones**

- **Definiciones:** let
  Comienza índice de valores en el manual (index of values).
- **Excepción ->** error de ejecución
- **Comentarios:** (* … *)
- Los **nombres de los valores** siempre empiezan por minúscula

Char.code
↓ └→ Minúscula porque es el valor
Mayúcula porque es el módulo

- **Unit = () ->** es el tipo más sencillo de dato, sólo tiene ese valor de tipo unit
  *** similar al void en C#**

- **Producto cartesiano:**
  (2, 5) -> int x int * escrito sin paréntesis es igual
  (2, 'A') -> int x char

2. **Tipos**

- Unit
- Function
- Int
- Float
- Double
- Producto cartesiano
- Definición (da nombre a un valor)
  <id> = nombre que se le pone al valor

  let <id> = <expr> -> cualquier expresión válida en OCaml

- Si escribimos 'Ledit Ocaml' en el terminal, abriremos el editor de OCaml, el cual funciona igual que el compilador interactivo, solo que tenemos la posibilidad de acceder a comandos o frases que hemos puesto con anterioridad usando "↑".

3. Sentencia IF-THEN-ELSE

if <b> then <e1> else <e2> -> tipo α

<b> -> expresión correcta en booleano
<e1> -> expresión correcta de tipo α
<e2> -> expresión correcta de tipo α
} Ambas han de ser del mismo tipo

- **<b1> || <b2>: bool ->** if <b1> then TRUE
  else <b2>
  OPERADOR 'OR' LÓGICO

- **<b1> && <b2>: bool ->** if <b1> then <b2>
  Else FALSE
  OPERADOR 'AND' LÓGICO

(*) LAZY: no evalúan la segunda.

4. Funciones

Function <id> -> <e> *Las funciones son valores

Ejemplo:

(Function x -> 2*x) (2+1) //de tipo int

Función de a en b Argumento válido para la función
Tiene que ser de tipo α

***evaluación de lager and lazy**

La palabra **'let'** sirve para definir funciones. La definición de funciones se puede abreviar:

```
let <f> = function <x> -> <e>
          ↓
let <f><x> = <e>
```

ejemplo: **función ABS**

```
let abs = function n -> if n >= 0 then n else -n;;
          ↓
let abs n = if n >= 0 then n else -n;;
```

También se pueden definir **funciones recursivas** introduciendo **'rec'** antes del nombre de la función.

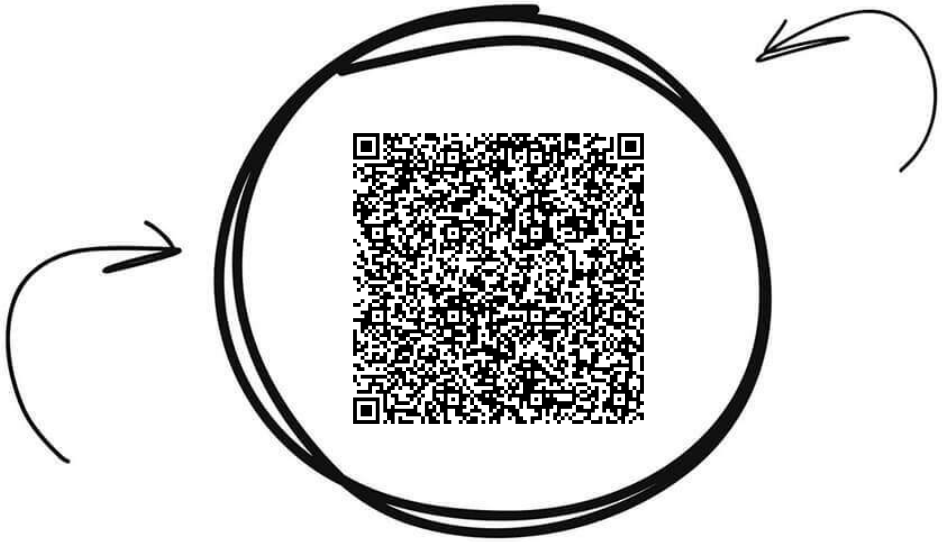
```
let rec fact = function n -> if n = 0 then 1 else n * fact(n-1);;
```

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

- **List.lenght ->** n° de elementos de una lista

- **Primeros n elementos:**
```
#let rec primeros n = function
      if n>0 then primeros (n-1)@[n]
      else [];;
```

- **Char.uppercase ->** convierte un carácter de letra minúscula a mayúscula

```
#let upper = function n ->
      if int_of_char(n)>96 && int_of_char(n)<123
      then char_of_int(Char.code(n) - 32)
      else n;;
```

→ val code : char -> int
Return the ASCII code of the argument.

Otros ejemplos mejores…

```
#let upper = function c ->
      let n = int_of_char(c)
      if n>96 && n<123
            then char_of_int(n - 32)
      else c;
```

```
#let upper = function c ->
      if c>= 'a' && c<='z'
            then char_of_int(c - 32)
      else c;
```

***Podemos hacer también Char.lowercase**

- **#[];;**
  - 'a list -> **'a es un tipo polimorfo, es decir, puede funcionar en cualquiera de los tipos**

- **function x -> x;;**
  ***función identidad:** es polimorfa, sirve para cualquier tipo

```
#function (x:int) -> x;;
```
-> También es la función identidad, pero sólo válida para enteros

***fst -> polimorfa**

- `let v = function f -> f0;;` } int -> 'a
  (f0;; aplica a 0)

  - `val v = (int -> 'a) -> 0`
    (int -> 'a) = f ; 0 = f(0)

  * v se aplica a funciones de int -> 'a

- let s = function x -> (function y -> x + y);;
  - val s = int -> (int -> int)
    #s(7) 1 = 8

    * Podemos representarla de una forma más abreviada:

    #let s x = function y -> x + y
    O
    #let s x y = x + y;;

¡OJO! No existen funciones de varios argumentos, sólo uno:

```
let suma = function p -> fst(p) + snd(p);;
```

tipo (int x int) -> int

Hay dos formas:

1. **Forma clásica:** función que devuelve una función
2. **Forma Hagkell Curry: ***
   #(+) 3 4;;

   (+): función a la curry. Todos los operadores binarios igual
   (^): concatenación de strings
   (@): concatenación de listas: 'a list -> 'a list -> 'a list

* otra forma de implementar la suma:
#let suma(a,b) = a + b

RELLENOS PARA TACOS PEKIS BRUTALES.

5. Listas

```
#[1; 2; 3]
#[1; 3; 5; 7]   } mismo tipo
                  Son secuencias, listas de enteros
```

- int list = [1; 2; 3]

- Para dos listas ser iguales, han de ser del mismo tipo y tener los mismos elementos en el mismo orden.

Hay listas de diferentes tipos de datos. Si existe el tipo de dato "α", existirá "α list". Son listas **homogéneas**: todos los elementos son del mismo tipo.

Se pueden solicitar dos cosas de una lista:
- La **cabeza** de la lista (head)
- La **cola**, que es todo menos la cabeza
-

En una "int list" la cabeza es un "int" y la cola es una "int list":
List.hd lista;; -> head
List.tl lista;; -> tail

* La lista es un valor, no se puede añadir porque sería otra lista, se puede **concatenar**: [1; 2; 3]@[5; 7]

Para referirse a la cabeza y la cola:
<h>: α
<t>: α list } La lista tiene como cabeza <h> y como cola <t>

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

```
#L1 = [5; 9]
#L2 = tl L1
```
- inte list = [9]

```
#L3 = tl L2
```
- int list = [] -> lista vacía

```
#hd L4 -> error: no hay
```

6. Patrones

Un **patrón** es una estructura que tiene forma de algún tipo de valor y que se pueden usar nombres para representar parte de ese valor.

- **Pattern Matching:** comparación de patrones.

```
#fst(x, y)
```

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

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

Una sola definición asigna a varios valores.

#let <pattern> = <exp>

Tienen que concordar en tipo con la forma del patrón

```
#let p = 1, 2;;
#let x, y = p;;       //x = 1; y = 2 ;
#let _, z = p;;       //'_' no es un nombre, no puede ser asociado con nada
                      //z = 2;
#let x,y = p, x;;     //x = (1, 2); y = 1
```

3::2::[]
3::(2::[])
'::' : constructor de listas

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

¡OJO! El pattern matching, cuando se vaya a aplicar, puede coger valores que no coinciden como por ejemplo "[]".
Abría que arreglarlo así:

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

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

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

Cuando no llega con un patrón se hace con varios, '|' se usa para separar patrones.

```
#let rec length = function
      (_::t) -> 1 + lenght(t)
     |[] -> Ø;;
```

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

```
function <p1> -> <e1>
| <p2> -> <e2>
| <p3> -> <e3>
.
.
.
| <pn> -> <en>
```

```
#let tl = function
       _::t  -> t
      |[]    -> []

#let hd = function
       h::_  -> h
      |[]    -> raise (Failure "hd");; //muestra error si se pasa []
```

- Failure "tl", Failure "hd", Division_by_zero … Son **valores de tipo exn: error de ejecución.**

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

***raise se aplica a un valor de exn -> 'a**

```
#let falsetl l = try List.tl l with
Failure "tl" -> [];;
```

```
try <e> with
| <p2> -> <e2>
| <p3> -> <e3>
.
.
.
| <pn> -> <en>
```

Donde <e> -> expr puede provocar un error

7. Factorial

```
#let rec fact n = if n = 0 then 1
                  else n * fact (n-1)*    //no tiene sentido para los números
                                          negativos, lo arreglaremos con
                                          "raise"
```

Solamente hay que añadirle al código:

if (n<0) then raise (Invalid_argument "fact")

Pero de esta manera hemos perdido eficiencia. en cada iteración calculará si n<0. basta con comprobarlo una sola vez fuera de la recursividad:

```
#let rec fact n =
if (n<0) then raise (Invalid_argument "fact")
else let rec funciton
     if n = 0 then 1
     else n * fact (n-1);;
```

O también:

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

//como no hemos implementado la función de forma recursiva, la llamada a fact
utilizará la función que teníamos antes. *
```

También lo podemos hacer con pattern_matching:

```
#let rec fact = fuction
      0 -> i
     |n -> n * fact (n-1);;
```

Otros ejemplos de funciones…

```
#let rec sum_to = function
      0 -> 0
     |n -> n + sum_to(n-1)
```

```
#let rec last = function
      h::[] -> h
     |1_::t -> last t
     |raise (Failure "last");;
```

Este algoritmo devuelve el último α de la lista:

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

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

Desarrollo del algoritmo de factorial:

```
fact 3
3 * fact 2
3 * (2 * fact 1)
3 * (2 * (1 * fact 0))
3 * (2 * (1))
3 * (2)
6
```

Va aumentando el espacio ocupado en la pila.

La función last podría definirse también:

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

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

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

```
     f      i
fact 3(fact  2)
     6(fact  1)
```

Ahora tenemos otra forma de calcularlo con recursividad terminal:

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

```
#let sum_to n =
      let rec aux (f, i) =
            if i > n then []
            else aux (f+i, i+1)
in aux (0, n);;

#let rec primeros n =
      if n <= 0 then []
      else primeros (n-1)@[n];;  //Para números muy grandes el tiempo se prolonga
            ↓                    demasiado
#let rec from_to m n =
      if m > n then []
      else m::from_to (m+1) n;;
```

Ahora primero se está mejor implementado con el from_to. Es mucho más rápido, pero no es una recursividad terminal.

Mas ejemplos de recursividad terminal:

Módulo list

```
let rec sumList = function
       [] -> Ø
      |h::t -> h + sumlist (t);;
```

El funcionamiento es:

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

La version terminal es:

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

```
let rec rev  = function
      [] -> []
     |h::t -> rev t@[h];;
```

En vez de poner @, se podría haber puesto el operando módulo list List.append. No es una función recursiva terminal. Su solución es darle la vuelta a la lista:

Reverse [1; 2; 3] -> [3; 2; 1]

La implementación terminal sería:

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

li es como un acumulador. Ahora sí que es una función recursiva terminal, la cual funciona de la siguiente manera:

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

8. Ordenación por selección

Para ello necesitamos dos funciones:

```
Función minl    -: 'a list -> 'a
Función remove  -: á -> a´list -> 'a list
```

```
let rec ordenación_por_seleccion = function
      [] -> []
     |l  -> let m -> min l in
            let resto -> remove ml in
            m :: ordenacion_por_seleccion resto;;
```

No se le puede dar el valor de h::t porque ya lo hacen todas las otras funciones, necesitamos la lista completa.
El problema que nos plantea esta función es que tiene complejidad cuadrática, ya que minl recorre la lista entera y remove, aunque haga la lista cada vez más pequeña, también la recorre entera.

```
let minl (h::t) =
     List.foldleft min h t;;
```

Nos da un aviso conforme no está declarado para la lista vacía, pero no hace falta, ya que dicha comprobación se hace en ordenación por selección antes de la llamada a la función.

Otra posible implementación, esta vez terminal, es:

```
let minl = function
      h::[] -> h
     |h1::h2::t -> minl (min h1 h2 :: t)
```

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

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

Esta no es una función terminal.

9. Fibonacci

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

Esta es la función no terminal, que tiene complejidad exponencial. En el TGR III esta implementado en una forma muy completa.

Sys.time()	-: unit -> float

No es funcional. devuelve el tiempo en segundos que lleva consumido un proceso.

Ahora haremos la función que dé el máximo de una lista. será de la forma:
maxl : 'a list -> 'a

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

Max es una función predefinida.

10. Funciones de ordenación

-: 'a list -> 'a list

Existe una función predefinida en OCaml, perteneciente al módulo Random, que coge valores aleatorios. Si se le pasa un valor n, el número aleatorio estará dentro del rango [0:n-1]. Si implementamos una función recursiva con esta otra función, podemos obtener una lista de valores aleatorios:

```
random list: -: int -> int -> int list

let rec ramdomList r n
      if (n > 0) then
            Random.int(n)::randomList r (n-1)
      else [];;
```

No es terminal.

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

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

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

#List.map (crono selc)(List.map (ramdomList max_int) [1000; 2000; 4000; 8000])

map aplica una función a cada 1 de los elementos de una lista, dando como resultado otra lista.
debemos mejorar lo de no hacerlo en dos pasos el minl y el resto. Coge una lista y pone ambos en un par 'a list -> 'a x 'a list
En el caso del map de arriba, se devuelve una lista cuyo primer elemento es una lista de 1000 elementos, el segundo una lista de 2000, el tercero una lista de 4000 y, el último, una lista de 8000 elementos. Después, con cada una de ellas hace crono y nos da una lista de cuatro valores, resultado de cuánto tiempo tarda en ordenarse la lista de 1000, 2000, 4000 y 8000 elementos.

11. Ordenación por selección

La que se le pasó a una lista ordenada y un valor a ordenar, y que devuelve una lista también ordenada. Para ello, crearemos la función insert:

```
insert -: 'a -> 'a list -> 'a list

let rec insert v =
       [] -> [v]
      |h::t -> if v < h then v::L
               else…
```

12. Ordenación por fusion

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

Va de 'a list -> 'a list
La ordenaremos de menora mayor.

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

```
reparte: 'a list -> ('a list x 'a list)
```

```
merge: ('a list x 'a list) -> 'a list
```

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

Función merge

```
let rec merge = function
       ([], l)|(l,[]) -> l
      |h1::t1, h2::t2 -> if (h1 < h2) then h1::merge(t1, h2::t2)
                        else h2::merge (h1::t1, t2);;
```

Se podría haber escrito cambiando los argumentos:

```
merge (l1, l2) = match l1, l2 with *
```

Función reparte

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

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

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

13. Creación de tipos

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

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

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

Cada constructor está asociado a un tipo que ya existe. Son funciones que insertan cada 1 de los tipos en 1 solo. Es obligatorio que el constructor empieza por mayúscula. Utilizándose:

```
Ch 'a';; -> -: Charint = Ch 'a'
Ch 'a' = Ch 'b';; -> false
In 8;; -> -: Charint = In 8
In 8 = Ch 'a' -> false
```

Los constructores juegan un papel muy importante ya que se pueden utilizar en el pattern mattching.

No se pueden hacer las siguientes comparaciones porque son tipos diferentes:

```
In 0 = 0;;
Ch 'a' = 'a';;
```

Trabajaremos ahora con funciones sobre este tipo:

INT_OF_CHARINT:

Función que lleva el tipo charint a:

- CHAR -> código ASCII
- INT -> int

```
let int_of_charint = function
      In n -> n;
     |Ch c-> int_of_char (c);;
```

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

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

```
L 3;; -> -: dint = L 3;;
R 5;; -> -: dint = R 5;;
R 3;; -> -: dint = R 3;;
R 3 = L 3;; -> false. Es distinto porque ha sido declarado con otro
                      constructor.
R 3 = R 5;; -> false. Es distinto porque ha sido declarado con otro
                      con otro int.
```

Es un tipo que para cada int tiene dos valores.

Función DINTORD

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

```
let dintord di1, di2 = match di1, di2 with
      L _, R _ -> true
     |R _, L _ -> false
     |R m, R n -> m <= n
     |L m, L n -> m <= n;;
```

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

- Los tipos también para hacer copias de los básicos:

```
type int2 =  D of int;;
```

- Ahora vamos crear un tipo que represente los números naturales:

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

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

```
Z ();; -> es el único que no se puede hacer con este constructor.

S(Z())      ┐ Los valores que se pueden declarar
S(S(Z()))   ┘ con S son infinitos

Z es un 0
S es el siguiente nat…
```

Función NATORD

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

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

En la declaración del tipo, es mejor poner solamente Z, en vez de Z of Unit. El valor "Unit" ya se le asigna por defecto. Son constructores constantes.

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

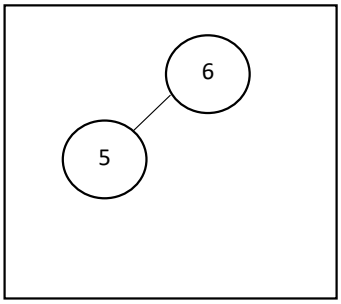
Cada constructor constante de un valor tiene cuatro valores (palo). Recuerda los tipos enumerados.

14. Árboles Binarios

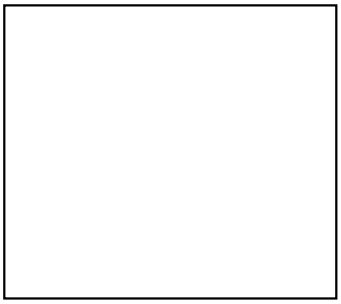
Comenzaremos definiendo el tipo:

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

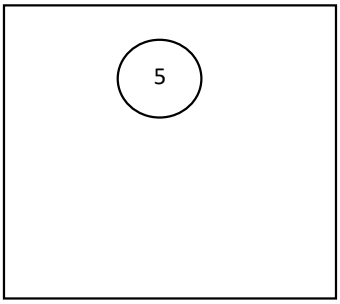
```
let v = Vacío;;
let  h5 = Nodo (5, Vacío, Vacío);;
let a1 = Nodo (6, h5, Vacío);;
```



a1



v



h5

Función Raíz

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

```
let raíz = function
      Vacio          -> raise (Failure "raíz")
     |Nodo (r, _, _) -> r;;
```

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

```
exception no_exite_raiz;;
Vacio -> raise no_esxiste_raiz
```

Función Altura

```
Let rec altura = function
      Vacio -> 0
     |Nodo (_, i, d) -> 1 + max (altura(i), altura d);;
```

ÁRBOLES BINARIOS POLIMÓRFICOS

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

Ahora los árboles pueden ser de cualquier tipo. 'a es un parámetro de tipo. Con esto estamos definiendo miles de tipos, 1 para cada tipo de 'a. De de esta manera:

```
Node (3, Empty, Empty) -> inttree
Node ('a', Empty, Empty) -> chartree
```

Función Crear

Función que nos permite crear un árbol con 'x' raíz, y que tiene las ramas vacías:

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

Función ROOT

Función idéntica a la "raíz" implementada antes, pero adaptada a este tipo de datos:

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

Función NNDOS

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

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

ÁRBOL

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

Ahora no se puede tener una rama o un árbol vacío.

ÁRBOLES CON X RAMAS

```
type 'a gtree =
  Vacío
| Nodo of ('a * 'a gtreelist);;
```

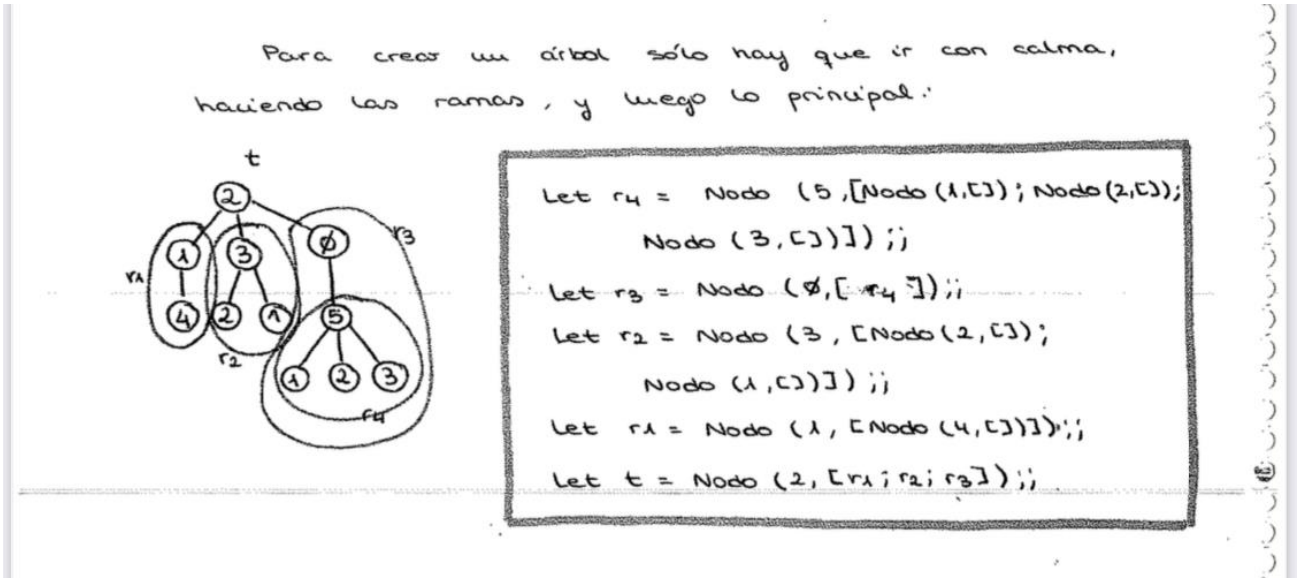
Ahora el número de ramas puede ser mayor que dos. También podemos quitar el constructor vacío ya que ahora no hace falta considerarlo.

Para un árbol de este tipo:

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

Función que permite crear árboles que solo tienen raíz:

```
let ar x = Nodo (x, []);;
```



Función nnodos

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

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

15. LOGO interpreter

Luego es un lenguaje de programación de alto nivel, en parte funcional en parte estructurado. Es conocido por su característica más explotada, los "gráficos tortuga". Esto consiste en dar instrucciones a una tortuga virtual, que es un curso gráfico usado para crear dibujos.

Nuestra práctica consiste, pues, en crecer un intérprete de logo utilizando para ello el módulo Graphics. La conversión del movimiento según las instrucciones se realizará en el plano XY.

INSTRUCCIONES

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

Para todo esto necesitamos:

- Un TAD que represente las instrucciones.
- Posibilidad de pasarle cualquier expresión aritmética que tenga como resultado un valor numérico, no solo constantes.
- Tener una función print que nos muestre la solución de una expresión aritmética.
- Posibilidad de concatenación de instrucciones.
- Un tipo de dado que nos indique el estado.
- Una función ejecutar: estado -> inst -> nuevo estado
- Otro tipo de dato para los argumentos. Pero hay que tener en cuenta que, aparte de las constantes, pueden ser expresiones aritméticas. Por lo que:
  - Un caso para constantes
  - Un caso para la suma
  - Un caso para la resta
  - …

Es una declaración de tipo recursiva, ya que, por ejemplo en la suma, no se van a sumar dos float. Si no que se van a sumar dos expresiones aritméticas. De hecho, en el tipo que queremos para "instrucción" vamos a usar este TAD y podemos ver que la instrucción "repetir" va a llevar una expresión aritmética y una instrucción, por lo que también va a ser recursivo este tipo. Instr puede ser una secuencia de instrucciones.

Para implementar eso, utilizaremos el módulo graphics de OCaml.

Para abrir el módulo es necesario poner:

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

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

```
α rev: int -> int ref
```

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

```
(!): 'a ref -> 'a
(:=): 'a ref -> 'a -> unit
Ref: 'a -> 'a ref
```

No tiene que devolver ningún valor. Efecto colateral fuerte.

Con este nuevo concepto, podemos realizar el programa del "turno" propuesto unas páginas antes:

```
let turno() =                    Muestra el valor de incrementado, por lo que sí
     i := ¡i + 1;                inicializamos i=ref 0, nunca mostrará el 0
     ¡i;;

Let turno() =                    Esta función, en cambio, muestra el valor actual de
     let r = ¡i in               i y luego lo incrementa. Aquí si se mostraría el 0.
     i := ¡i + 1;
r;;
```

```
let i = ref 0;;
     -: int ref {contents = 0}
!i;;
     -: int = 0
(:=) i 1;;
!i;;
     -: int = 1
i := 2;;
!i;;
     -: int = 2
```

Podríamos también definir una función de reseteo, que simplemente tendría que poner el valor de i a 0:

```
let reset() = i := 0;;
```

16. Programación imperativa. Bucles

Lo que se va a dar a continuación, viene en el manual, en el capítulo de expresiones (6, 7), porque se corre supone a la parte dos -> Part II, The OCaml Language

BUCLE WHILE

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

Si la expresión Guyana da como resultado false, se devuelve Unit. sida verdadera, true, se pasa a evaluar <e>.
En caso de que la expresión boliviana no dependa de nadie, nos encontramos ante un bucle vacío o un bucle infinito. lo que realmente importa es el cambio que realiza <e>.

BUCLE FOR

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

- <e1> y <e2> son dos expresiones de tipo int. Se comienza evaluándolos en orden creciente (ambos inclusive).

```
let <i> = V in <e>
```

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

Factorial usando programación imperativa

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

¡f es el valor de f. La i no es ningún valor de referencia, por eso no llena ninguna admiración.

Fibonacci usando la programación imperativa

```
let fib n =
     let i = ref 0                    //con ref se crean e inicializan variables
     and f = ref 0
     and u = ref 1 in while (!i < n) do
          i := !i + 1;
          f := !f + !u;
          u := !f - !u;
     done;
!f;
```

Función SWAP

SWAP: ('a ref * 'a ref) -> unit

- Intercambia valores entre dos variables de referencia

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

Almacenamos el valor en una variable auxiliar, de manera que no nos la carguemos al intentar intercambiarla.

¡OJO!

```
let x = ref 0;;        -> int ref = ´contents = 0}
!x;;                   -> 0
let z = x;
!z;;                   -> int = 0
Z := !z + 1;;
!z;;                   -> int = 1
!x;;                   -> int = 1

!x;;                   -> int = 1
```

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

Para el estado de la tortuga (LOGO), necesitamos una tupla que contenga sus características, como pueden ser: las coordenadas, la posición del lápiz (LEVANTADO O NO)…

17. TIPO REGISTRO

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

- El nombre de los campos debe ir en minúscula

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

let nombre {apellido = _; edad = _; nombre = n} = n;;

nombre p;;
     -: String = "Jose M."

let nombre {nobre = n} = n;;

p.nombre;;
```

Veamos que podemos obtener el valor de 1 de los campos del registro de varias maneras diferentes. Aunque, quizás, la más cómoda sea la última.

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

```
let s = "hola";;
s[i];;          -> carácter en la posición i
s[0];;          -> 'h'
s[0] <- 'k'     -> sustituye el carácter en la posición 0 por 'k'
```

18. Operaciones de entrada/salida

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

- in_channel
- out_channel

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

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

Ahora expondremos diferentes funciones relacionadas con estos tipos:

- **pint_char ->** Es la función más básica, imprime por pantalla un solo carácter
- **print_String ->** Envía los strings sin salto de línea
- **print_endline ->** Envía los caracteres del string + '\n'
- **print_int ->** Envía los caracteres, no el número
- **print_float ->** Idem qué print_int
- **print_newline ->** envió un salto de línea con vuelco de buffer. Podríamos definirlo como un print_endline del string vacío.

Cualquiera de esas funciones puede ser implementada a partir de la función "print_char".

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

Si ejecutamos esta función el compilador interactivo, tenemos que tener en cuenta que se comparte la salida de nuestro código con la del compilador. La función PRINT_ENDLINE que hemos definido es igual a la del módulo Pervasives.

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

Si un programa está constantemente realizando operaciones de entrada/salida, eso va a perjudicar mucho el rendimiento general.
La gestión realizada por el sistema operativo es transparente para el programa. Esto es relevante en el sentido de que, cuando se manda imprimir, por ejemplo, no tiene que aparecer en ese momento. Aparecerá, realmente cuando el sistema operativo mande la señal de volcarse.

Si está ocupado con otras cosas, puede tardar bastante. Pero, hay momentos en los que necesitamos que sea en ese mismo instante. Esto es posible, forzando a que todo aquello que esté pendiente sea mandado al dispositivo; es decir, forzando a que vuelque el buffer del sistema operativo.

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

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

LECTURA DE LA ENTRADA ESTÁNDAR (TECLADO)

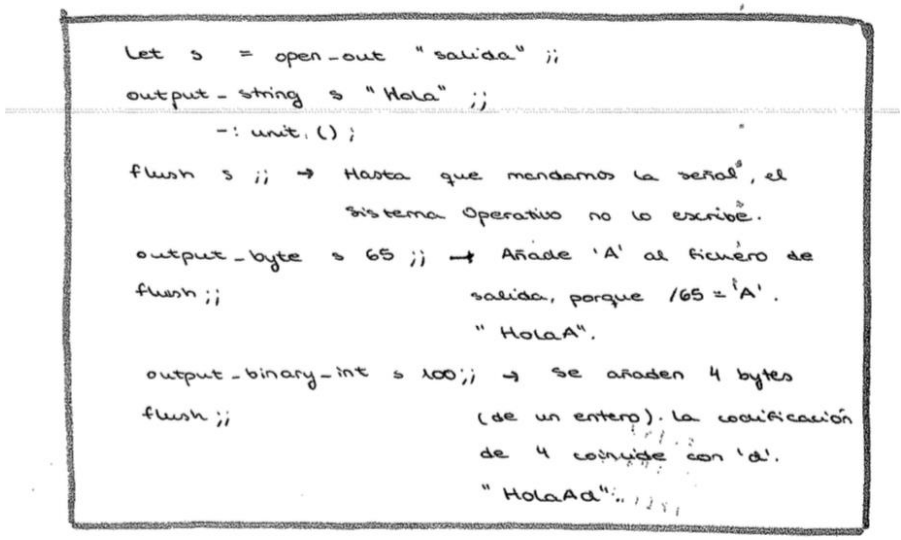
- Lectura básica, leemos un carácter (un byte). Hasta que se dé el retorno de carga, el proceso no estará listo. Los caracteres no están listos ni disponibles hasta darle a *enter*.
- **read_line ->** es una aplicación sobre unt, que nos devuelve el string que se haya introducido hasta el primer salto de línea (o hasta que termine la entrada estándar).
- **read_int ->** lee y aplica "int_of_string"
- **read_float ->** lee y aplica "float_of_string"

Estas dos últimas funciones no son muy útiles, de que tienen una alta probabilidad de fallo.

OPERACIONES DE LA SALIDA ESTÁNDAR

Valor de tipo **out_channel.** Eso nos da "open_out". Se le pasa un string nos devuelve un tipo out_channel, que estará asociado al archivo. Es decir, open_out quería un archivo en el disco con el nombre que le digamos para escribir en él. Si ya existe, se machaca al existente. En caso de no tener permisos, se producen un error (una excepción): Sys.error. Coloca un puntero al principio del fichero para empezar a leer.

- **close_in ->** no es tan importante el cerrarlo como es en la salida, por qué estamos leyendo, no escribiendo ni modificando nada, pero debemos cerrarlo igual
- **input_byte ->** lee un valor y lo devuelve como int
- **input_char ->** lee un valor y lo devuelve como char
- **input_line ->** le como string desde el chat en el que está posicionado hasta el primer salto de linea
- **input_line ->** lee valores de cualquier tipo
- **read_line ();; ->** queda en espera de la entrada hasta que se le dé al enter
- **input_binary_int**
- **seek_in**
- **pos_in**
- **stdin**



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

```
lec rec output_string_list s = function
       [] -> unit
      |h::t -> output_string s (h^"\n");
           Output_string_list s t;;
```

Tipo: output_channel -> string List -> unit

```
let l = ["uno"; "dos"; " "; "cuatro"; "5"];;

let s open_out "lista";;

output_string_list s l;;

close_out s;;
```

Si ahora miramos con cat el contenido de la lista:

```
uno
dos

cuatro
5
```

19. Programación orientada a objetos

CLASE

Es la que define como es, cómo se crea y cómo se comporta un objeto. Es el modelo a partir del cual creamos las instancias del objeto. Define:

- **Variables de instancia:** Atributos. Contienen los datos asociados a esa instancia y representan sus propiedades.
- **Métodos:** Son operaciones de manipulación de dichos datos asociados.

OBJETO

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

SINTAXIS

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

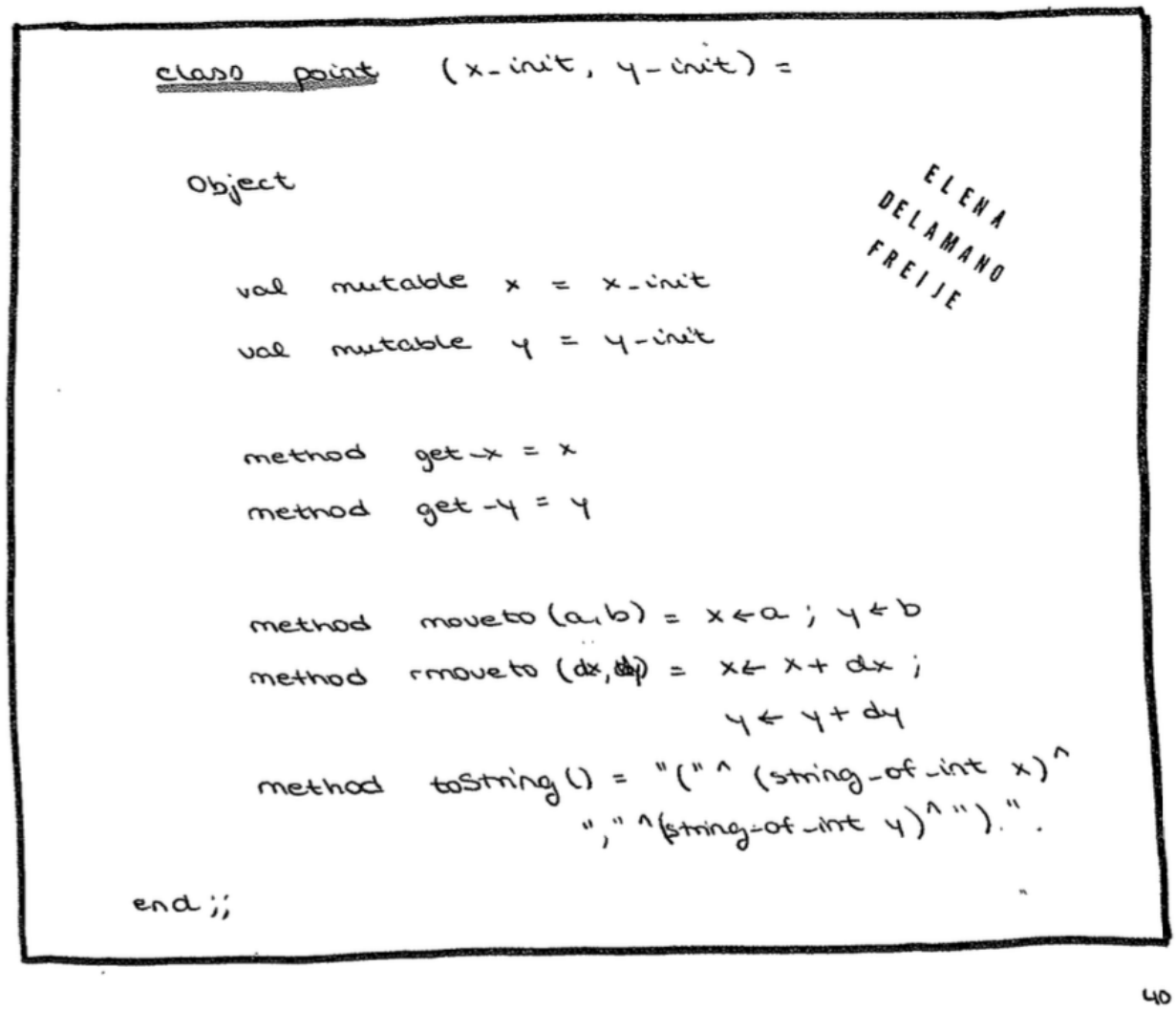
```
class nombre_clase p1…pn =
      Object [(alias)]

      val [mutable] nombre_var1 = expr1
      .
      .
      .
      val [mutable] nombre_varm = exprm

      method nombre_metodo1 pm1…pmn = expr1
      .
      .
      .
      method nombre_metodom pm1…pmn = exprm
```

[mutable] -> opcional. Puede variar su valor. Si no, sería constante.

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

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

1. Los atributos son siempre privados, es decir, no son visibles fuera de la clase. Si queremos que así sea tenemos que definir métodos getters y setters.
2. Debemos tener cuidado con la anotación de manejo de los atributos (¡, :=, ref)
3. Cuando No hay argumentos en un método por convenio se tiene que: sii es un getter, no se pasa nada, pero si es un setter, hay que forzarlo pasándole unit '()'.



El tipo del objeto aquí no es point.

No es de tipo point ya que no es nominal, sino estructural. Realmente, el tipo de o1 es:

```
get_x : int
get_y : int
moveto : int * int -> unit
rmoveto : int * int -> unit
toString : unit -> string
```

El tipo se define por las características de la clase, no pero el nombre que se le asigna, solo que en OCaml se pone este nombre como abreviatura de todas esas características.

TIPO DE OBJETO: Es la lista de los métodos y sus respectivos tipos, los atributos no.
OCaml es un lenguaje multiparadigma. El tipo no es la clase.
Para poder crear instancias de una clase:

```
let p1 = new nombre_clase p1…pn
```

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

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

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

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

Para llamar a los métodos utilizamos "#".

```
id_objeto #nombre_metodo pm1…pmn
```

El tipo de funciones que se definen para la creación de objetos, como es el caso del 'make_point', se llaman **funciones factoría.**
Una vez tenemos una instancia del objeto, podemos manipular dicha instancia como cualquier otra definición del lenguaje. Por ejemplo, podemos almacenarlo en listas, compararlo… Es igual que cualquier otra "variable".

Respecto a la comparación de objetos, hay algo que tener en cuenta: un objeto puede ser igual a otro física o estructuralmente. Para la igualdad física se utiliza el operador '=' o '<>', pero compara las direcciones de memoria. Para la igualdad física se utilizan '==' y '!='.

En en caso de querer una igualdad estructural entre 2 objetos, debemos definir un método "equals" que haga dicha función.

RELACIONES

1. **AGREGACIÓM**

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

val x = _____ **o** **val x = [| ____ , ____ , ____ |]**

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

```
class arista2 (p1, p2) =
     object
          val vertices = (p1, p2)    -> getters con fst y snd
     end;;
```

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

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

El problema que podemos encontrarle a estas implementaciones es que cambia el tipo. Así se nos permitirá cualquier cosa en el par, por lo que será un par '('a, 'b)', Porque el motor de inferencia de tipos de OCaml no es capaz de detectar nada que le indique qué tipo es sí o sí son del mismo.
Ambos son del mismo tipo 'a solo en caso del array o la lista. El problema viene dado en el par.

2. HERENCIA

La subclase hereda los mismos atributos o variables de instancia y métodos que la superclase, pudiendo:

- Añadir nuevos atributos
- Modificar atributos o métodos ya existentes

Para unificar la herencia:

```
class nombre_subclase p1…pn =
      object
      inherit nombre_superclase pp1…ppn as alias
            val …
            method …
      end;;
```

Para el alias, habitualmente, se usa super.
En cuanto a añadir atributos y métodos, no hay problema. Donde sí que los podemos encontrar es al modificarlos. Cuando modificamos tenemos que tener presente ciertas cosas:

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

¿Cómo funciona el mecanismo de herencia?

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

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

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

REFERENCIAS

- **No es posible llamar a tus propios métodos, a menos que se utilices el alias (this/self)**
- **En ocasiones, también se indica un identificador del tipo:**

```
class class nombre_clase p1…pn =
     object (self:'self)
```

Referencia a si mismo y a "su propio tipo"

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

DELAYED BINDING

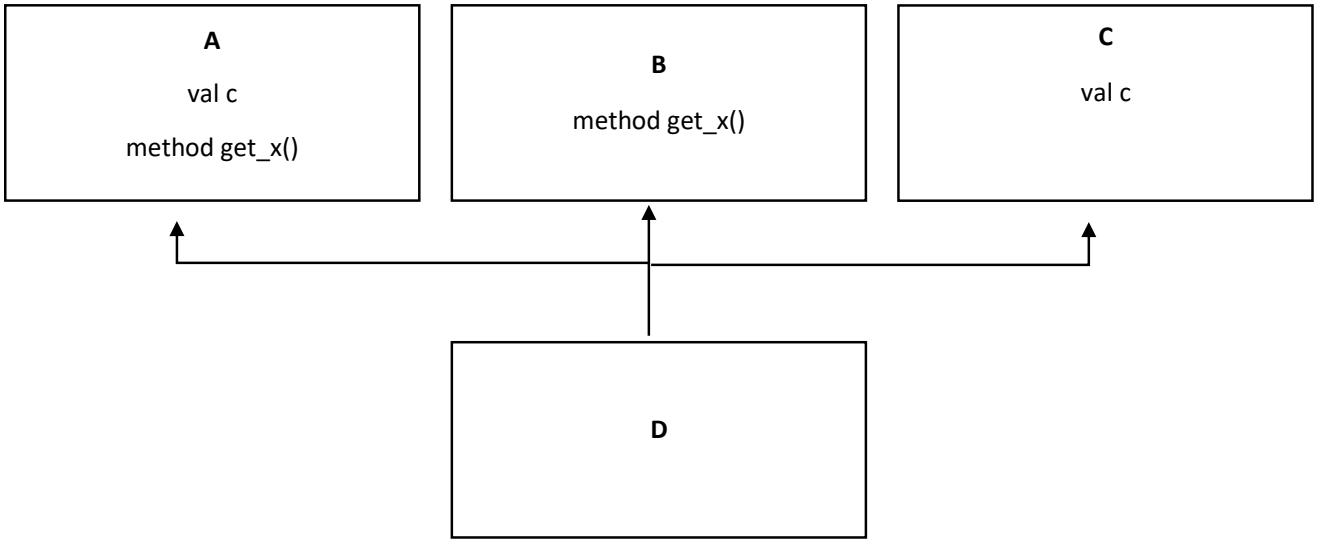
En OCaml, el método, código o expresión concreto a llamar es determinado en tiempo de ejecución. No es definido a priori.

HERENCIA MÚLTIPLE

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

PROBLEMA 1: ¿Qué sucede cuando hay métodos o atributos con el ismo nombre entre padres?

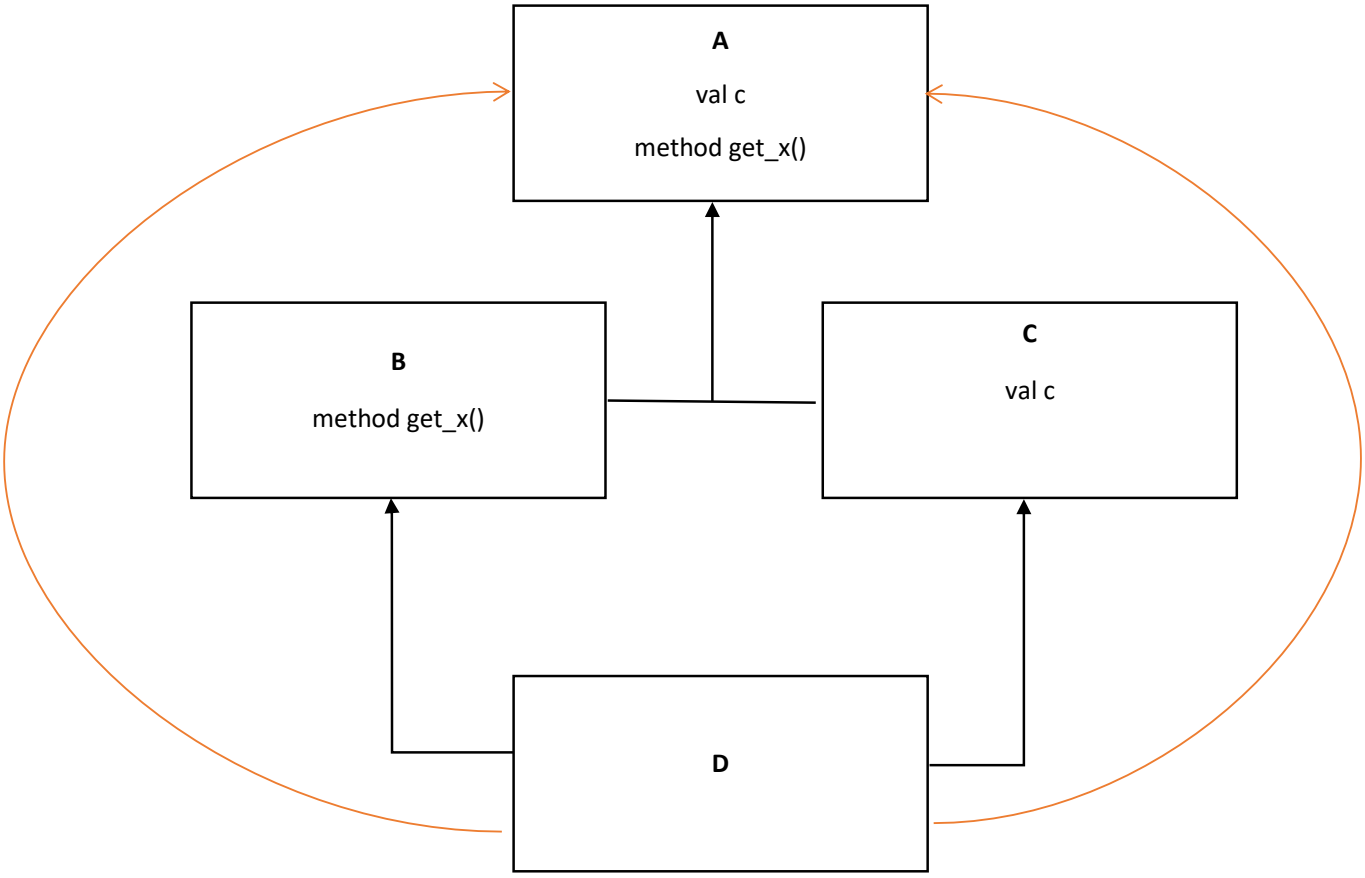
- Deben coincidir en tipo
- Se hereda de la última superclase en el inherit (coincide con el concepto de inclusión textual)



```
class nombre_clase p1…pn =
      object
            inherits A,B,C
            .
            .
            .
      end;;
```

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

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



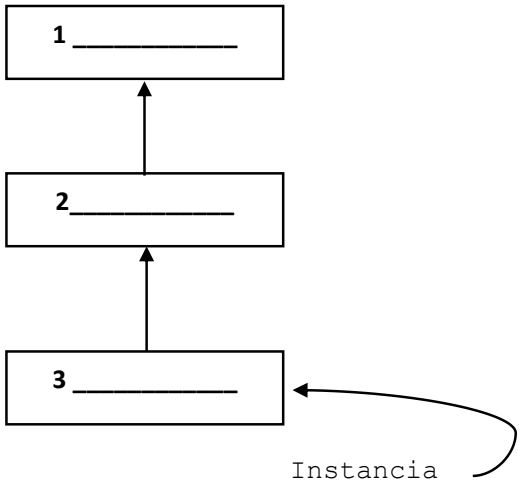
Indirectamente, D hereda de A. Esto sucede cuando varios padres tienen antepasados comunes. Se aplica la misma solución propuesta en el problema 1. Seguiremos viendo la herencia múltiple más adelante.

INICIALIZACIÓN

```
class nombre_clase p1…pn =
     object
          inicializer expr
          .
          .
          .
     end;;
```

El 'inicializer' es similar al constructor:

1. se ejecuta una sola vez, automáticamente durante la creación del objeto
2. es lo primero que se evalúa inmediatamente después de que haya sido creado
3. puede emplear tanto métodos como atributos del objeto
4. si existe una jerarquía, se evaluarán todos los inicializers desde la superclase más general hacia abajo



Otra alternativa es usar "let…in" en la definición de la clase. Pero, sin embargo, este se ejecutará antes de la creación del objeto, por lo que no se puede acceder a los métodos y atributos del objeto.

VISIBILIDAD

Tanto de atributos como de métodos-

- **ATRIBUTOS ->** 'privada', es decir, sólo se puede acceder desde eñ propio objeto.
- **MÉTODOS ->** por defecto es 'pública', pero se puede hacer privada mediante:

```
method private nombre_metodo p1…pn = expr.
```

¡IMPORTANTE! 'private no se corresponde con el de JAVA, es más similar al protected, ya que:

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

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

CLASES, MÉTODOS Y ATRIBUTOS VIRTUALES

- **MÉTODO ABSTRACTO:** aquel que no está implementado, del que sólo desarrollamos su interfaz.
  Sintaxis:

```
method virtual nombre_metodo p1…pn : tipo
```

- **ATRIBUTO ABSTRACTO:** aquel que no está inicializado, si no que solamente está declarado su tipo. Sintaxis:

```
val [mutable] virtual nombre_atributo : tipo
```

- **CLASES ABSTRACTAS:** son aqurllas que tienen algún método o atirbuto abstracto, hay que declararlas como tal. Sibtaxis:

```
class virtual nombre_clase p1…pn0 =
     object
     .
     .
     .
     end;;
```

Al declarar un método como abstracto no se ponen los parámetros, ya que se implican al inicializar el tipo:

int -> int -> char -> String

Donde los parámetros son los subrayados.

Una subclase virtual no se puede instanciar directamente.

Con 'initializer' se puede acceder a los atributos y métodos de la clase, mientras que con una inicialización normal no.

¿CÓMO COMBINAR LAS CLASES ABSTRACTAS Y LA HERENCIA MÚLTIPLE?

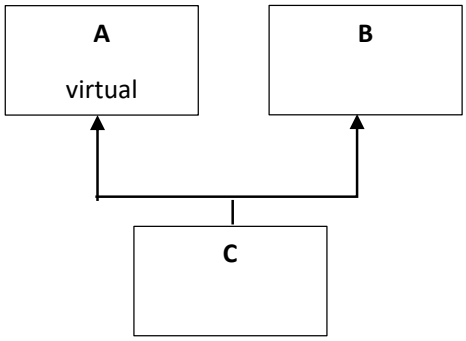
PROBLEMA 1 -> Teníamos métodos y atributos de igual nombre en varias clases de manera simultánea.

PROBLEMA 2 -> Si había herencia repetida

PROBLEMA 3 -> Subclases virtuales

Para solucionar estos problemas, se pueden considerar dos métodos.

CASO 1:



Encargar a la subclase de implementar métodos o atributos virtuales. Para observar mejor este caso:

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

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

CASO 2:

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

A tiene un método virtual A1 : tipo A1
B implementa A1, método virtual : tipo A1

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

20. MANEJO DEL POLIMORFISMO A NIVEL DE OBJETO

Para ello comenzaremos hablando de las **clases parametrizadas:**

Objetivo:

Permitir contenidos polimórficos y su manejo. Puede verse como una evolución de los tipos parametrizados. Debe que quedar claro que en el momento en el que se cree una instancia de ese objeto, quedará ligada al tipo concreto que se le haya pasado en la creación.

SINTAXIS

```
class ['a, `b, …] nombre_clase p1…pn =
      object
      .
      .
      .
end;;
```

Los corchetes son 'reales', obligatorios, no implican opcionalidad. Un nombre de tipo polimórfico para cada parámetro que queramos. Ejemplo:

```
class ['a, `b, …] par (X0 : 'a)(Y0 : 'b) =
      object
            val x = X0
            val y = Y0

            method fst = x
            method snd = y
end;;
```

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

```
class ['a ] par (X0 : 'a)(Y0 : 'a) =
```

Ahora pasamos los atributos a mutable:

```
class [`a, `b, …] par (X0 : `a)(Y0 : `b) =
      object
            val mutable x = X0
            val mutable y = Y0

            method fst = x
            method snd = y

            method set_fst x1 = x <- x1
            method sey_snd y1 = y <- y1

end;;
```

```
let p1 = new par "hola" 1;;

p1 #fst;;
      -: String = "hola"
p1 #snd;;
      -: int = 1
p1 #set_fst "adios";;

p1 #fst;;
      -: String = "adios"
p1 #set_snd 2;;

p1 #snd;;
      -: int = 2
p1 #set_fst 545;;
      -  ERROR
```

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

¿Qué sucederá si ahora incluimos herencia?

21. CLASES PARAMETRIZADAS Y HERENCIA

OPCIÓN 1:

Podemos mantener su carácter polimórfico.

```
class [`c, `d ] ppair (xx : `c)(yy : `d) =
      object
            inherit [`c, `d ] par xx yy

            method to_pair() = (x,y)
end;;
```

En el inherit se le indica cual va a ser la `a (`c) y cuál la `b (`d). Luego ya se encargará él de ligarlo.

OPCIÓN 2:

Especializar la clase completamente a unos tipos concretos, es decir, que deje de ser polimórfica. Si queremos que ppar sea solo para enteros y floats, vamos a crear una subclase que sirva solo para esos tipos:

```
class int_float_par xx yy =
      object
            inherit [int, float] par xx yy
end;;
```

OPCIÓN 3:

Especializar la clase parcialmente a unos tipos concretos, concretizar alguno de los tipos polimórficos, pero no todos:

```
Class [`a] int_x temperatura_par xx (yy = `a) =
      object
            inherit [int, `a] par xx yy
end;;
```

22. OPENTYPES

Tipos abiertos, concepto que se los aplica a las clases objeto. Ocaml tiene un tipado estático, es decir, antes de evaluar una expresión comprueba su tipo. Si este tipo es correcto, se compila, si no da error. Los tipos que hemos visto hasta ahora son cerrados.

Los tipos abiertos son mucho más permisivos: el tipo de sociedad OA la clase a la que pertenece un objeto también es denominado "notación". Podemos solicitar el tipo entre '<>' sus nombres y tipos.

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

El doble punto recibe el nombre de "elipsis", se corresponde con un significado similar al etcétera o ... en nuestro lenguaje. En OCaml, se refiere a que tiene el método get_x y alguno más. Es decir, que al menos tiene ese método, pudiendo tener solo ese o alguno a mayores.

LIMITACIÓN

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

```
F(x:#point) = x #get_x;;
```