**Vídeos - FLIP - Tema 3 Paradigma funcional**

[**Introducción**](https://media.upv.es/player/?id=046aeb30-9200-11e6-806a-45b4f471ec59)

[**Algunas características distintivas**](https://media.upv.es/player/?id=211c7ff0-9200-11e6-806a-45b4f471ec59)

**Parte 1**

1.- [Tipos funcionales](https://media.upv.es/player/?id=3fd8b1c0-9200-11e6-806a-45b4f471ec59). [Tipos algebraicos](https://media.upv.es/player/?id=5f415030-9200-11e6-806a-45b4f471ec59)

2.- Tipos predefinidos

[El tipo Char](https://media.upv.es/player/?id=826521e0-9200-11e6-806a-45b4f471ec59)

[El tipo "tupla"](https://media.upv.es/player/?id=9f8aae20-9200-11e6-806a-45b4f471ec59)

**Ejercicio 1.**

1. Define una función que, dados dos números enteros, los devuelva en una tupla ordenados de menor a mayor.
2. Define una función siglet :: Char -> Char que transforme cada letra del alfabeto en la letra siguiente, dejando invariantes los restantes caracteres. Suponemos que debe cumplirse siglet ‘Z’ = ‘A’ y siglet ‘z’ = ‘a’.
3. Supongamos que se representa una fecha como una tupla (d,m,a) de números enteros, correspondientes al día, mes y año respectivamente. Define una función que calcule la edad de un individuo en años, dadas la fecha de su nacimiento y la fecha actual.
4. Sean sigma y pi las funciones especificadas como sigue:

                  sigma f a b = f a + f (a+1) + f (a+2) + ... + f b

                  pi f a b = f a \* f (a+1) \* f (a+2) \* ... \* f b

 construye definiciones recursivas ejecutables para las dos funciones, incluyendo declaraciones de tipo.

[El tipo predefinido String](https://media.upv.es/player/?id=bb028a60-9200-11e6-806a-45b4f471ec59)

[Los tipos predefinidos numéricos](https://media.upv.es/player/?id=d9b5bb80-9200-11e6-806a-45b4f471ec59)

[El tipo predefinido "lista"](https://media.upv.es/player/?id=fa097d40-9200-11e6-806a-45b4f471ec59)

**Ejercicio 2.** Define mediante listas intensionales las funcionas sigma y pi mencionadas en el ejercicio anterior.

**Ejemplos.** A continuación podéis ver algunos ejemplos de funciones típicas definidas mediante listas intensionales:

    map f xs = [f x | x <- xs]  --aplica f a todos los elementos de la lista xs

    filter p xs = [x | x <- xs, p x]  --devuelve una lista con los elementos

                                      --de la lista xs que cumplen p

    repetido y xs = length [ () | x <- xs, y == x]

              --devuelve el número de repeticiones del elemento y en la lista xs

    divisores n = [ i | i <- [1..n], n `mod` i == 0]

              --devuelve el número de divisores de un número n dado

    pertenece y xs = not (null [ () | x<-xs, y == x])

              --devuelve True si el elemento y pertenece a la lista xs

**Ejercicio 3.** Define una función elimDups :: [Int] -> [Int] que elimine de una lista los elementos duplicados adyacentes. Por ejemplo: elimDups [1,2,2,3,3,3,1,1]=[1,2,3,1]

**Ejercicio 4.**Define dos funciones  
    any, all :: (a -> Bool) -> [a] -> Bool  
que satisfagan las siguientes especificaciones  
    any p xs ⇔ existe x en xs tal que p x  
    all   p xs ⇔ para todo x en xs se cumple p x

**Ejercicio 5.** Define una función que compruebe si una lista de enteros está ordenada.

**Ejercicio 6.** Define una función que cree una lista de n copias de x (Nota: usa listas intensionales).

Operaciones sobre listas

1.- Propiedades de una lista

* Longitud de una lista: length :: [a] -> Int

length [ ] = 0  
length (x:xs) = 1 + length xs

* ¿Lista vacía?: null :: [a] -> Bool

null [ ] = True  
null (x:xs) = False

2.- Combinación de listas

* Concatenación: (++) :: [a] -> [a] -> [a]

[ ] ++ xs    = xs  
(x:xs) ++ ys = x:(xs ++ ys)

* Aplanamiento: concat :: [[a]] -> [a]

concat [ ] = [ ]  
concat (xs:xss) = xs ++ concat xss

* Combinación: zip :: [a] -> [b] -> [(a,b)]

zip [ ] xs  = [ ]  
zip (x:xs) [ ]      = [ ]  
zip (x:xs) (y:ys) = (x,y) : zip xs ys

Ejemplo: zip [1, 2, 3] [”a”, ”b”, ”c”] = [(1,”a”), (2,”b”), (3,”c”)]  
                donde zip :: [Int] -> [String] -> [(Int, String)]

3.- Acceso a componentes de una lista

* Cabeza de una lista: head :: [a] -> a

head (x:xs) = x

* Último de una lista: last :: [a] -> a

last [x] = x      
last (x:xs) = last xs

* Acceso posicional: (!!) :: [a] -> Int -> a

(x:xs) !! 0 = x       
(x:xs) !! (n) = xs !! (n-1)

5.- Transformación de listas

* Inversión: reverse :: [a] -> [a]

reverse [ ] = [ ]  
reverse (x:xs) = reverse xs ++ [x]

* Aplicación de una función a sus elementos:

map :: (a->b) -> [a] -> [b]  
map f [ ] = [ ]  
map f (x:xs) = f x : map f xs

4.- Acceso a sublistas de una lista

Principio de una lista: init :: [a] -> [a] -- todos menos el último elemento  
Resto de una lista: tail :: [a] -> [a] -- todos menos el primer elemento

Ejemplo: averiguar la posición de un elemento en una lista

Vamos a definir una función posicion que indique la posición que ocupa un elemento en una lista.  
        position :: a -> [a] -> Int  
        : position ”b”  [”a”, ”b”, ”c”]      
        2  
**Sugerencia:** marcamos cada elemento de la lista con su posición, con lo que así es muy sencillo localizar cuál es la posición del elemento buscado:  
        [”a”, ”b”, ”c”]    ----->    [(”a”,1), (”b”,2), (”c”,3)]

Para ello, usamos la función zip:

    zip [”a”, ”b”, ”c”]  [1, 2, 3]= [(”a”,1), (”b”,2), (”c”,3)]

Aunque no sabemos cuántos elementos tiene la lista xs,  sí sabemos que serán tantos como length xs:

    zip xs [1.. length xs]

Por último, podemos seleccionar la posición en la que aparece el elemento que estamos buscando así:

    position x xs = [pos | (x’,pos) <- zip xs [1..], x’ == x]

    position ”b” [”a”, ”b”, ”c”] = [2]

Por tanto, la solución completa es la siguiente:

    position :: a -> [a] -> Int  
    position x xs = head [pos | (x’,pos) <- zip xs [1..], x’ == x]

donde la función head nos devuelve el primer elemento de una lista.

Ejemplo: calculando la longitud de un camino

Vamos a definir una función que calcule la longitud de un camino representado por la secuencia de puntos del mismo. Concretamente, usamos los siguientes tipos de datos para su representación:  
  type Point = (Float,Float)  
  type Path = [Point]

Por ejemplo, dado el siguiente camino:  
  examplePath = [p,q,r,s]

tendríamos la siguiente longitud:  
  pathLength = distance p q + distance q r + distance r s

Dos funciones útiles que vamos a emplear son las siguientes:

  init [p, q, r, s] = [p, q, r]  
  tail [p, q, r, s] = [q, r, s]

Y, para combinarlas, usaremos la función zip:

  zip …      = [(p,q), (q,r), (r,s)]

La solución completa será por tanto:

pathLength :: Path -> Float  
pathLength xs = sum’ [distance p q | (p,q) <- zip (init xs) (tail xs)]

sum’ :: [Float] -> Float  
sum’ [] = 0.0  
sum’ (x:xs) = x + sum’ xs

distance :: Point -> Point -> Float  
distance (p1,p2) (q1,q2) = sqrt (sqr(p1 -  q1) + sqr(p2 - q2))

Ejemplo: ordenación de listas

El algoritmo más sencillo consiste en la inserción en una lista ordenada:

*insertar x [ ] = [x]*

*insertar x (y:ys)*

*|x<=y = (x:y:ys) -- [x]++(y:ys)*

*|otherwise = y : (insertar x ys)*

*inorder [ ] = [ ]*

*inorder (x:xs) = insertar x (inorder xs)*

Sin embargo, existen formas más eficientes de ordenar una lista, como por ejemplo el algorithm mergeSort. En este caso, se trata de dividir la lista en dos partes iguales, ordenar cada parte, y poner juntas las dos mitades ordenadas:

*mergeSort [] = []*

*mergeSort [x] = [x]*

*mergeSort xs | size > 0 =*

*merge (mergeSort front) (mergeSort back)*

*where size = length xs `div` 2*

*front = take size xs*

*back = drop size xs*

*merge a@(x:xs) b@(y:ys)*

*| x <= y = x : merge xs b*

*| otherwise = y : merge a ys*

*merge [ ] ys = ys*

*merge xs [ ] = xs*

[3.- Polimorfismo: coerción, genericidad y sobrecarga en Haskell](https://media.upv.es/player/?id=1e858bf0-9201-11e6-806a-45b4f471ec59)

**Nota:** En el vídeo anterior, hay una errata en el último ejemplo. Concretamente, esto

**instance (Eq a) => Eq (Tree a)**

**Void == Void = True  
  (Branch x l1 r1) == (Branch y l2 r2) = (x==y) && (l1 == l2) && (r1==r2) \_ == \_ = False**

**...**

debería ser:

**instance (Eq a) => Eq (Tree a) where**

**Void == Void = True  
  (Branch x l1 r1) == (Branch y l2 r2) = (x==y) && (l1 == l2) && (r1==r2) \_ == \_ = False**

**...**

(faltaba la palabra reservada **where**)

### Ejercicios

Consideremos el tipo Nat que hemos definido anteriormente:

    data Nat = Zero | Suc Nat

Se pide:

1. sobrecargar los operadores aritméticos (+ y \*) de la clase Num para poder usarlos para sumar y multiplicar naturales;
2. Sobrecargar show de la clase Show para mostrar los naturales como el número que representan, por ejemplo Zero como 0, Suc Zero como 1, …;
3. sobrecargar los métodos necesarios de Enum para poder definir listas aritméticas con valores de tipo Nat. Por ejemplo: [Zero..Suc (Suc Zero)] daría [Zero, Suc Zero, Suc (Suc Zero)];
4. sobrecargar < de la clase Ord para poder comparar valores del tipo Figure según el valor de su área;
5. sobrecargar show de la clase Show para visualizar un círculo con su radio entre paréntesis (e.g., (2.5)) y un rectángulo con sus lados entre corchetes separados por una coma (e.g., [1.5,2.5]).

**Parte 2**

[4.- Modelo operacional](https://media.upv.es/player/?id=682b6080-b7e0-11e7-9aae-59f0880c7499)

### Ejercicio

Dado el siguiente programa funcional:

insertar x [ ] = [x]

insertar x (y:ys)  
  |x<=y = (x:y:ys)            
  |otherwise = y : (insertar x ys)

inorder [ ] = [ ]  
inorder (x:xs) = insertar x (inorder xs)

indica la secuencia de reducción de la expresión: inorder [2,6,1] tanto con la estrategia de evaluación perezosa como con la estrategia de evaluación impaciente.

[Evaluación](https://media.upv.es/player/?id=e07efff0-b7e6-11e7-9aae-59f0880c7499)

**Parte 3**

[5.- Funciones anónimas y composición de funciones](https://media.upv.es/player/?id=b88a6640-bb5d-11e7-9aae-59f0880c7499)

[6.- Iteradores y compresores](https://media.upv.es/player/?id=f4617790-bd6e-11e7-9aae-59f0880c7499)

[El esquema de Mapreduce](https://media.upv.es/player/?id=55d4f870-bb5e-11e7-9aae-59f0880c7499)