# Programación Funcional

## Tema 1: Lenguaje Haskell.

### 1. Programación funcional, 1er ejemplo:

Para empezar veremos un ejemplo de la implementación del Quick Sort en Haskell:

```
qsort::Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y < x] ++ [x] ++ qsort [y | y<-xs, y >= x]
```

La ventaja principal es que los programas son más cortos, lo que implica menos tiempo de desarrollo. Pero la principal desventaja es que esta clase de lenguajes es menos eficiente (porque normalmente son interpretados).

Lenguajes funcionales:
+ Estrictos / impacientes: Lisp, Scheme, ML (SML, CAML), Erlan.
+ Perezosos: Miranda, Gofer, Clean, Haskell.

La diferencia con un lenguaje imperativo como C, es que no se cambia el estado del programa dentro de las funciones, los parametros siempre valen los valores dados inicialmente. Los resultados siempre se dan en lo que devuelve la función en base a sus argumentos.

### 2. Ventajas de la programación funcional:

+ Fundamentos teóricos muy claros, concisos y útiles.
+ Nivel de abstracción muy alto (cercano a los lenguajes de especificación). Mayor productividad y mayor facilidad de mantenimiento.

### 3. Desventajas:

+ Menor eficiencia en tiempo de ejecución y memoria usada.
+ Poco extendido, lo que implica menos recursos y utilidades.
+ Cambios de estado no fácilmente expresables.

### 4. Aplicaciones típicas:

+ Prototipado.
+ Aplicaciones de Inteligencia Artificial.

### 5. ¿Por qué tiene ventajas la programación funcional?

+ Transparencia referencial (no hay efectos laterales).
+ Polimorfismo. Una misma función puede valer para cualquier tipo.
+ Inferencia de tipos automática.

+ Orden superior. Por ejemplo, tenemos que hacer dos funciones tales que:

```
mas1 [] = []
mas1 (x:xs) = x+1 : mas1 xs
cuadr [] = []
cuadr (x:xs) = sqr x : cuadr xs
```

  Como son muy similares podríamos hacer lo siguiente:

```
map f [] = []
map f (x:xs) = f x : map f xs
mas1 xs = map (+1) xs
cadr xs = map sqr xs
```

  Esto como hacer plantillas de funciones.
+ La pereza del lenguaje.

### 6. Para finalizar:

En 1977 John Backus recibe el premio Turing, y publica el paper: “Can programming be liberated from the Von Neuman Style? A functional style and its Algebra of Program.” Después de esto a finales de los 80 nace el lenguaje Haskell. La última versión es Haskell98, que salió a la luz en el 98-99.

La programación funcional y la programación lógica deriva de lo que se llama la programación declarativa. Este tipo de programación implica declarar lo que debe hacerse, y no como ha de hacerse.

## Tema 2: Tipos predefinidos. Expresiones y valores. Funciones.

### 1. Tipos predefinidos:

+ Típos básicos:
  + **Int**: Enteros de tamaño limitado (parecido al int de C).
  + **Integer**: Enteros de tamaño “ilimitado” (como una lista de bytes creciente a medida que se va necesitando).
  + **Float**, **Double**: Reales con coma flotante.
  + **Char**: Caracteres ASCII.
  + **Bool**: Booleano (True o False).

  (Nota: Haskell es case sensitive al igual que en C, distinguiendo entre mayúsculas y minúsculas.)
+ Notación de mayúsculas y minúsculas: (Para la primera letra.)
  + **Mayúsculas**: Nombres de los tipos, los elementos de los tipos enumerados (ejemplo: True, False).
  + **Minúsculas**: Nombres de funciones, nombres de los parametros.
+ Tipos compuestos:
  + **Tuplas**: Si $a_1$, ... , $a_n$ son tipos válidos, entonces ($a_1$, ... , $a_n$) representa una n-tupla cuya 1ª componente es $a_1$, y la n es $a_n$. Esto es equivalente a las struct de C/C++. Ejemplos: (Bool, Int), (Int, Int, Int), (Char, Int, (Int, Char), Bool) o () que es parecido al void de C.
  + **Listas**: Si *a* es un tipo válido, entonces [*a*] representa listas de elementos de tipo *a*. Ejemplos:

    [Integer] → [1, 127, -3]

[Char] → [‘h’, ‘o’, ‘l’, ‘a’]
Notación especial para [Char] → “hola”
(Al ser muy usado hay un sinónimo de [Char], que es String, para ser usado este tipo de datos fácilmente.)
Se pueden combinar las listas con las tuplas, y al revés:
[(Bool, [Int], (Char, [[Integer]]), [String])]
Pero en estos casos se suelen usar sinónimos para facilitar la tarea de los sufridos programadores (equivale a usar los typedef de C).

+ **Funcionales**: Si *a* y *b* son tipos válidos, entonces (*a*->*b*) representa el tipo de las funciones que reciben como entrada algo de tipo *a* devuelve algo de tipo *b*. (Nota: Las funciones pueden usarse como parámetro, guardarse en estructuras de datos, etcétera.)

## 2. Expresiones. Funciones predefinidas:

+ Una expresión denota un valor de un cierto tipo: 2 * 5 + 4. Eso da 19 que está en los tipos Int, Integer, Float y Double.
+ Las expresiones pueden contener variables en el sentido matemático pero no en el imperativo. No varía durante la ejecución el valor de estas.
+ Las expresiones pueden usar operadores y funciones:
    + **Aritméticos**: + - * / `div` `mod`
        + / solo se usa con operandos reales.
        + div y mod se usa con operandos enteros.
        + El resto pueden ser usados por cualquier tipo numérico, pero ambos operandos han de ser del mismo tipo.
    + **Booleanos**: && ||
    + **Comparaciones**: < <= > >= == /=
    + **Para listas**:
        + elemento : lista  3 : [4, 5, 6] = [3, 4, 5, 6]
        + lista ++ lista  [1, 2] ++ [3, 4] = [1, 2, 3, 4]
+ Funciones predefinidas:
    + **Booleanas**: not
    + **Numéricas**: abs, negate, div, mod, round, floor, ceiling, cos, sin, tan, acos, asin, atan, etcétera...
        - div 7 2 = 7 `div` 2
        - floor 5.3 = 5  round 5.3 = 5
        - ceiling 5.3 = 6  round 5.7 = 6
        - round 5.5 = 6  round 4.5 = 4
        - Un estandar del IE[3] dice que round en igualdad de distancia, se elegirá el par para no tener una desviación muy elevada en los cálculos.
    + **Ordinales**: succ, pred, fromEnum, toEnum.
        - fromEnum ‘a’ = 97
        - toEnum 97 = ¿?
        - En ese caso no devuelve nada, porque no sabe que tipo de dato devolver, pero podemos usar algo como (toEnum97):”dios”, para que nos devuelva “adios”. En este caso Haskell infiere que el tipo de datos que estamos utilizando son caracteres.
    + **Tuplas**:
        - fst (3, “hola”, 4.5) = 3
        - snd (3, “hola”, 4.5) = “Hola”

+ **Listas**:
  - length [1, 8, 3] = 3
  - take 3 [1, 8, 9, 2, 5] = [1, 8, 9]
  - head [7, 8, 1] = 7

+ Niveles de prioridad:
  0) Parentesis
  1) Funciones
  2) **  ^ (exponenciación de floats y enteros, 2 ^ 3 = 8)
  3) *  /  `div`  `mod`
  4) +  -
  5) ++  :
  6) ==  <=  <  >  >=  /=
  7) &&
  8) ||
  9) Otros...

## 3. Definición de nuevas funciones:

+ La declaración de una nueva función tiene dos partes:
  + Declaración del tipo de la función (opcional).
  + Declaración del comportamiento.

+ Ejemplos:

```
mas1::Int -> Int
mas1 x = x + 1

f x = x + 1
```

En este caso el compilador asigna el siguiente tipo por inferencia al analizar el contenido de la función: (f::Num a => a -> a)

```
esBisiesto::Int -> Bool
esBisiesto x = ((mod x 4) == 0) && ((mod x 100) /= 0) || ((mod x 400) == 0)

aMayus::Char -> Char
aMayus c = toEnum (fromEnum c + diferencia)
   where diferencia = fromEnum 'A' - fromEnum 'a'
```

Nota sobre los comentarios: Para poder meter comentarios en nuestro código tenemos las dos siguientes formas:

```
-- Comentario de una sola linea.
{- Comentario de
  varias lineas. -}
```

+ Definición de funciones por casos:

```
mayor::Int -> Int -> Int
mayor x y = if x > y then x else y
```

En Haskell el if-then-else equivale al operador :? de C, ya que es un operador, no una instrucción, así que siempre tiene que devolver un valor. Otra forma de hacer lo mismo y mejor es con guardas:

```
mayor x y
   | x > y = x
   | otherwise = y
```

Podemos tener tantas guardas como necesitemos en nuestra función. El sangrado uniforme es algo muy relevante en Haskell, por ello es un punto en el que hay que tener mucho cuidado. Otra forma de hacer funciones por casos es usando un case:

```
f x y z = case z of
            0 -> x * y
            1 -> x + y
            otherwise -> x - y
```

Por último, la cuarta forma que tenemos es con ajuste de patrones:

```
f x y 0 = x * y
f x y 1 = x + y
f x y n = x - y
```

Esto lo que hace es que Haskell analice cada función, y la primera que encuentre donde pueda encajar nuestra llamada a dicha función, será donde tendremos que entrar. Otros ejemplos varios de funciones son:

```
factorial::Integer -> Integer
factorial 0 = 1
factorial 1 = 1
factorial x = x * factorial (x - 1)

fib::Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

-- Equivale a la función repeat
repetir::Int -> a -> [a]
repetir 0 x = []
repetir n x = x : repetir (n - 1) x

-- Equivale a la función sum
sumatorio [] = 0
sumatorio (x:xs) = x + sumatorio xs

-- Equivale a la función length
longitud::[a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

-- Equivale a la función reverse
alReves::[a] -> [a]
alReves [] = []
alReves (x:xs) = (alReves xs) ++ [x]
```

```
-- Equivale a la función take
coger::Int -> [a] -> [a]
coger 0 xs = []
coger n [] = []
coger n (x:xs) = x : coger (n - 1) xs

-- Equivale a la función drop
saltar::Int -> [a] -> [a]
saltar 0 xs = xs
saltar n [] = []
saltar n (x:xs) = saltar (n - 1) xs

-- Equivale a la función splitAt
partir::Int -> [a] -> ([a], [a])
partir 0 xs = ([], xs)
partir n [] = ([], [])
partir n (x:xs) = (x : izq, der)
   where (izq, der) = partir (n - 1) xs

   partir 3 [1, 2, 3, 4, 5, 6]
      partir 2 [2, 3, 4, 5, 6]
         partir 1 [3, 4, 5, 6]
            partir 0 [4, 5, 6]
            ([], [4, 5, 6]) = (izq, der)
         (3 : [], [4, 5, 6])
      (2 : [3], [4, 5, 6])
   (1 : [2, 3], [4, 5, 6])
([1, 2, 3], [4, 5, 6])

-- Al poner otra letra, a y b pueden ser tipos distintos.
-- Equivale a la función zip
cremallera::[a] -> [b] -> [(a, b)]
cremallera [] ys = []
cremallera xs [] = []
cremallera (x:xs) (y:ys) = (x, y) : cremallera xs xy

   cremallera [1, 2, 3] “hola”
   [(1, ‘h’), (2, ‘o’), (3, ‘l’)]

-- Equivale a la función unzip
anticremallera::[(a, b)] -> ([a], [b])
anticremallera [] = ([], [])
anticremallera (a, b):xs = (a:as, b:bs)
   where (as, bs) = anticremallera xs

-- Ordenación por inserción.
inserta::Int -> [Int] -> [Int]
{- Suponemos que la lista está ordenada -}
inserta x [] = [x]
inserta x (y:ys)
   | x <= y = x : y : ys
```

```
    | otherwise = y : inserta x ys
insertSort::[Int] -> [Int]
insertSort [] = []
insertSort (x:xs) = inserta x (insertSort xs)
```

(Nota: Para hacer que esta función opere con cualquier tipo e dato ordenado, se tendría que poner lo siguiente:

```
        inserta::Ord a => a -> [a] -> [a]
        insertSort::Ord a => [a] -> [a]
```

Con esto le indicamos a Haskell que a es un tipo de dato ordenado, es decir, que se puede establecer un orden comparando diferentes elementos de este conjunto de datos.)

```
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = juntar (mergeSort xs1) (mergeSort xs2)
   where (xs1, xs2) = splitAt ((length xs) `div` 2) xs
juntar [] ys = ys
juntar xs [] = xs
juntar (x:xs) (y:ys)
   | x <= y = x : juntar xs (y:ys)
   | otherwise = y : juntar (x:xs) ys
```

## 4. Definición de operadores binarios en Haskell:

Primero vamos a diferenciar lo que sería un operador puesto de forma infija o puesto de forma prefija:

| | | |
|---|---|---|
| + Infijo: | a + b | a `div` b |
| + Prefijo: | (+) a b | div a b |

Esto hace que en haskell podamos usar los operadores de dos formas distintas, algo que podrá sernos útil para aplicar cálculos al aplicar el orden superior. Así que para definir un operador nuevo tenemos que hacer:

```
infixr 5 -+-
(-+-)::Int -> Int -> Int
x -+- y = x * y - 3
```

La palabra clave *infixr* nos indica que este operador va a ser un operador infijo que se opera por la derecha (se calcula primero lo que hay en el operador derecho, y luego lo que hay en el izquierdo). El siguiente valor hace referencia a la prioridad del operando, y lo siguiente es el símbolo con el que vamos a representar nuestro operando.

Esto es igual que las funciones salvo por cosas como:
+ Que el nombre va entre paréntesis para utilizar la notación prefija.
+ Que tenemos la opción de proporcionar la precedencia y decir como se asocia.
(Nota: Por defecto se asume infixl 9 al definir operadores.)

Así que las diferentes asociaciones que podemos indicar son:
+ *infixl*: Infijo de izquierda a derecha.
+ *infixr*: Infijo de derecha a izquierda.

Respecto a los niveles de prioridad tenemos que el menor es 9 y el mayor es 1, como ya se comentó en el apartado sobre operadores. Por último un ejemplo más sobre como crear un operador nuevo:

```
(&+/)::Int -> Int -> Int
a &-/ 0 = a + 3
a &-/ b = a * b
```

## 5. Mejorando la eficiencia: parametros acumuladores:

Como mejorar la complejidad de algo como:

```
alReves::[a] -> [a]
alReves [] = []
alReves (x:xs) = (alReves xs)++[x]
```

Ya que la concatenación con ++ de por sí, tiene una complejidad de O(n), y como vamos a ejecutar alReves n veces, tenemos una complejidad final de $O(n^2)$. Para evitar esto tenemos los parámetros acumuladores:

```
alReves::[a] -> [a]
alReves xs = invierte xs []
invierte::[a] -> [a] -> [a]
invierte [] ys = ys
invierte (x:xs) ys = invierte xs (x:ys)

invierte            [1, 2, 3]       []
   invierte         [2, 3]          1:[]
      invierte      [3]             2:[1]
         invierte   []              3:[2, 1]
            [3, 2, 1]
```

De este modo la complejidad se reduce a O(n), ya que la operación : es constante de complejidad O(1). Otro ejemplo al que podemos aplicar lo mismo es fibonacci:

```
fib::Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

El coste de esta función es exponencial, de $O(2^n)$. Esto es algo aberrante y para mejorar esto y evitar los cálculos innecesarios que no paran de repetirse, haríamos:

```
fibonacci::Int -> Int
fibonacci n = fib n 1 1
fib::Int -> Int -> Int -> Int
fib 0 x y = y
fib n x y = fib (n-1) (x+y) x

fib          5       1       1
```

| | | | |
|---|---|---|---|
| fib | 4 | 2 | 1 |
| fib | 3 | 3 | 2 |
| fib | 2 | 5 | 3 |
| fib | 1 | 8 | 5 |
| fib | 0 | 13 | 8 |

## 6. Clases de tipos:

Haskell es más “quisquilloso” en cuestiones de tipos que otros lenguajes.

+ ¿Por qué?
    + En otros lenguajes funcionales no es así. Parcialmente, independiente de la programación funcional.
    + El motivo es que proporciona un sistema muy potente y general para utilizar clases de tipos.
+ Motivación:
    + Algunos operadores queremos que valgan para distintos tipos de datos, para realizar la sobrecarga e operadores.
    + Suma de enteros o de reales: +
    + La sobrecarga se limita a operadores numéricos predefinidos, y a operadores de comparación. Sobrecarga ad-hoc.
    + Es obligatorio que el programador declare el tipo de las variables, de modo que el compilador sepa que concreción del operador debe usar.
+ Haskell:
    + Podemos sobrecargar cualquier cosa: polimorfismo paramétrico.
    + El programador no está obligado a declarar tipos, el compilador debe inferirlos automáticamente.
    + Solución general: Uso de “clases de tipos” que engloban todos aquellos tipos que permiten realizar ciertas operaciones.
    + Más potencia y complejidad.
    + Más cómodo para cosas complejas (más incómodo para operaciones básicas).
+ Clases:
    + Clases numéricas:
        - **Eq**: Pertenecen todos los tipos para los que puedan aplicarse comparaciones por igualdad → == /=

        (Ejemplo: Para poder ver el tipo de una función podemos usar el comando :t del interprete de Haskell.

        ```
        >:t (==)
        (==)::Eq a => a -> a -> Bool
        ```

        Así podemos ver que el operador == recibe dos operandos de la clase Eq, y devuelve un valor booleano.)
        - **Ord**: Pertenecen todos los tios con orden → < <= > >=
        - **Num**: Todos los tipos numéricos (ComplexFloat, Rational, Int, ...) → + - * negate abs
        - **Fractional**: Todos los fraccionales → /
            - **Floating**: Trigonométricas, logaritmos, exponencial, ...
        - **Integral**: Todos los enteros (Int, Integer)
        - **Enum**: Todos los tipos que pueden enumerarse →succ, pred, toEnum, fromEnum.
        - **Show**:Todos los tipos que pueden convertirse en cadenas de caracteres.

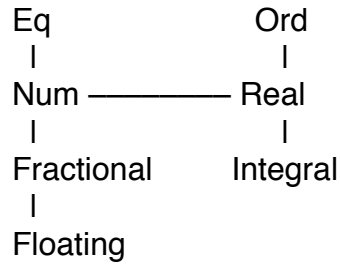
show::Show a => a -> String
show 3.7 → “3.7” = [‘3’, ‘.’, ‘7’]
show True → “True”

- **Read**: Todos los tipos que pueden obtenerse a partir de cadenas de caracteres.

read::Read a => String -> a
read “True” → True
read “3.7” → 3.7

(Nota: Hay muchas más clases predefinidas. Todo esto no solo sirve para tipos predefinidos, los nuevos tipos que creemos también pueden pertenecer a clases de tipos. Además se pueden crear nuevas clases de tipos, y hacer que los tipos que deseemos (predefinidos o nuevos) pertenezcan a las nuevas clases.)

+ Jerarquía:




## 7. Definición de nuevos tipos:

Hay dos formas de crear nuevos tipos en Haskell:
+ Renombramiento (tipos sinónimos).
+ Tipos algebraicos (nuevos constructores).

a) Renombramiento:
+ No se crea un “nuevo” tipo.
+ Solo da un nombre a un tipo ya existente:
  type Corto = ([Int], String)
+ El nombre debe empezar en mayúscula.
+ El nuevo tipo y su definición pueden usarse indistintamente.
+ Realmente no estamos creando un tipo nuevo.
  f::Corto -> Int
  f (xs, ys) = sumatorio xs

El tipo Corto, realmente es tratado en función a su estructura, por lo que el tipo sigue siendo: f::([Int], String) -> Int

b) Tipos algebraicos:
+ Se crean nuevos tipos (distintos a todos los ya existentes):
  data MiTipo = definición...
+ Tipos enumerados:

data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo

  + Cada constructor de los tipos tiene que tener la 1ª letra en mayúsculas.
  + Todos los constructores usados deben ser distintos dentro de la definición, y también deben ser distintos de los usados en otras definiciones de tipos dentro del mismo módulo.

```
esLaborable::Dias -> Bool
esLaborable Sabado = False
esLaborable Domingo = False
esLaborable dia = True

esLaborable x = (x /= Sabado) && (x /= Domingo)
(Nota: Si la definición de días indica la clase de tipo:
data Dias = Lunes | ... | Domingo deriving (Eq, Ord, Enum, Show,
Read))
```

+ Tipos con variantes:

  data EntOChar = Entero Int | Caracter Char

  + Valores válidos: Entero 3 Caracter 'a'
  + Valores inválidos: Entero 'c' Caracter 4
  + Anólogo a los registros con campos variantes de pascal.
  + Todos los constructores han de ser distintos, y con la primera letra en mayúsculas.

```
contInt::EntOChar -> Bool
contInt (Entero x) = True
contInt (Caracter x) = False
(Ojo: EntOChar != (Int, Char).)
```

  + Al igual que antes, al declarar el tipo podemos derivar automáticamente su pertenencia a ciertas clases básicas.

```
data EntOChar = Entero Int | Caracter Char
    deriving (Eq, Ord, Show, Read)
(No podemos derivar de Enum, porque no es una enumeración.)

data MiDato = LE [Int] | LC [Char]
longitud (LE xs) = length xs
longitud (LC xs) = length xs

data MiReg = R Int [Bool] String
(MiReg != (Int, [Bool], String)
sacaTexto::MiReg -> String
sacaTexto (R n ts xs) = xs
```

+ Tipos algebraicos recursivos:

→ ¿Qué es una lista de enteros?

```
data ListaInt = LstIntVacia | LstIntDato Int ListaInt
        → LstIntDato 3 (LstIntDato 5 LstIntVacia)

long::ListaInt -> Int
long LstIntVacia = 0
long (LstIntDato x xs) = 1 + long xs

ListaInt != [Int]
(Los tipos son distintos, aunque en el fondo representen lo mismo.)
```

→ ¿Podemos definir un tipo lista paramétrica de forma similar?

```
data Lista a = LstVacia | LstDato a (Lista a)
suma::Lista Int -> Int
```

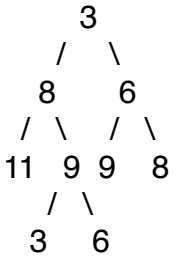
```
suma LstVacia = 0
suma (LstDato x xs) = x + suma xs
```

→ Podemos reescribir todo lo visto para [a].
(De hecho, [a] está predefinido más o menos de este modo.)

→ Podemos hacer definiciones similares para árboles:

```
data ArbBinInt = ABIHoja Int | ABINodo Int ArbBinInt ArbBinInt
```

Ejemplo:



```
inOrden::ArbBinInt -> [Int]
inOrden (ABIHoja x) = [x]
inOrden (ABINodo x izq der) = (inOrden izq) ++ [x] ++ (inOrden der)
```

→ Lógicamente podemos definir cualquier tipo de arbol:

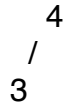
```
data ArbBin a = ABNulo | ABNodo a (ArbBin a) (ArbBin a)
```

Con esto tenemos un arbol clásico en el que sus extremos son vacíos.

```
inOrden::ArbBin Int -> [Int]
inOrden ABNulo = []
inOrden (ABNodo x izq der) = (inOrden izq) ++ [x] ++ (inOrden der)
```

→ Para tener un arbol completamente genérico:

```
data ArbGen a = AGNulo | AGNodo a [ArbGen a]
```



En esta definición AGNulo no se utiliza realmente:
AGNodo 4 [AGNodo 3 [], AGNodo 5 []]

→ Definición de un arbol de busqueda:

```
data Ord a => ArbBsq a = Nulo | Hoja a | Nodo a (ArbBsq a) (ArbBsq a)

esVacio::Ord a => ArbBsq a -> Bool
esVacio Nulo = True
esVacio arbol = False

busca::Ord a => ArbBsq a -> a -> Bool
busca Nulo x = False
busca (Hoja dato) x = (dato == x)
busca (Nodo dato izq der) x
        | dato > x = (busca izq x)
        | dato < x = (busca der x)
        | otherwise = True

meter::Ord a => ArbBsq a -> a -> ArbBsq a
meter Nulo x = Hoja x
meter (Hoja y) x
```

```
        | x < y = Nodo y (Hoja x) Nulo
        | otherwise = Nodo y Nulo (Hoja x)
    meter (Nodo y izq der) x
        | x < y = Nodo y (meter izq x) der
        | otherwise = Nodo y izq (meter der x)
```

(Nota: Los constructores tienen que ser nombres únicos incluso entre los tipos distintos del mismo programa, como por ejemplo:

```
    data Ord a => ArbBus a = ABNulo | ABHoja a |
                                ABNodo a (ArbBus a) (ArbBus a)
    data ArbBin a = A2Nulo | A2Hoja a | A2Nodo a (ArbBin a) (ArbBin a)
    data ArbGen a = AGNulo | AGNodo a [ArbGen a]
```

De otro modo podría haber conflicto de tipos en las funciones que manejaran estos tipos de datos distintos.)

```
    eliminar::Ord a => ArbBsq a -> a -> ArbBsq a
    eliminar Nulo x = Nulo
    eliminar (Hoja y) x
        | x == y = Nulo
        | otherwise = Hoja y
    eliminar (Nodo y iz de) x
        | x < y = Nodo y (eliminar iz x) de
        | x > y = Nodo y iz (eliminar de x)
        | esVacio iz = de
        | otherwise = Nodo maxIz (eliminar iz maxIz) de
            where maxIz = maximo iz

    maximo::Ord a => ArbBsq a -> a
    maximo (Hoja x) = x
    maximo (Nodo x iz de)
        | esVacio de = x
        | otherwise = maximo de

    profundidad::Ord a => ArbBsq a -> Int
    profundidad Nulo = 0
    profundidad (Hoja x) = 1
    profundidad (Nodo x iz de) = 1 + max (profundidad iz, profundidad de)
```

→ Otro tipo recursivo: Representación de expresiones aritméticas.

```
    data Op = Sum | Res | Mul | Div | Mod
    data Expresion = Const Int | Expr Expresion Op Expresion

    eval::Expresion -> Int
    eval (Const x) = x
    eval (Expr exp1 op exp2) = aplica op (eval exp1) (eval exp2)

    aplica::Op -> Int -> Int -> Int
    aplica Sum x y = x + y
    aplica Res x y = x - y
    aplica Mul x y = x * y
    aplica Div x y = x `div` y
```

```
aplica Mod x y = x ‘mod` y
```

# Tema 3: Orden superior.

## 1. Funciones de orden superior:

La idea principal es que las funciones son “ciudadanos de 1ª clase”, es decir, podemos usarlas como cualquier otro tipo:
- Pasarlas como parámetros de otra función.
- Devolverlas como resultado de otra función.
- Almacenarlas en estructuras de datos.
- Etcétera...

Algunas funciones de orden superior predefinidas son: map, filter, zipWith, dropWhile, span, la familia fold, y la familia scan.

```
map::(a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
> map sqrt [9, 25, 1]
      [3, 5, 1]

filter::(a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
   l f x = x : filter f xs
   l otherwise = filter f xs
> filter even [1, 2, 3, 4]
      [2, 4]

zipWith::(a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
→ productoEscalar xs ys = sum (zipWith (*) xs ys)

dropWhile::(a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
   l p x = dropWhile p xs
   l otherwise = x:xs

span::(a->Bool) -> [a] -> ([a], [a])
span p [] = ([], [])
span p (x:xs)
   l p x = (x:iz, dr)
   l otherwise = ([], x:xs)
   where (iz, dr) = span p xs
```

## 2. Funciones fold:

```
and::[Bool] -> Bool
and [] = True
and (x:xs) = x && or xs

or::[Bool] -> Bool
or [] = False
or (x:xs) = x || or xs

length::[a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

concat::[[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss

> concat [[1, 2, 3], [4, 5], [6, 7], []]
[1, 2, 3, 4, 5, 6, 7]
```

Como se puede ver, existe un patrón entre algunas operaciones y por ello podemos utilizar el orden superior para crear funciones genéricas. Para hacer eso sustituimos el constructor de la lista por un nuevo operador, y el [] por un nuevo caso base.

+ El primer parametro es una función que recibe dos parametros de tipo a y b, y devuelve un resultado del mismo tipo que b.
+ El segundo parametro es el resultado del caso base.
+ El tercer parametro es la lista que vamos a procesar.
+ Y por último indicamos que el resultado a devolver es de tipo b.

```
foldr::(a->b->b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

La función foldr está ya creada en haskell, con lo que no hace falta implementarla de nuevo. Con esta función tendremos un esquema muy reutilizable:

```
> sum [3, 7, 2]
> foldr (+) 0 [3, 7, 2]

> and [...]
> foldr (&&) True [...]

> or [...]
> foldr (||) False [...]

> length [...]
> foldr incr 0 [...]
      where incr x y = 1 + y

> concat [...]
```

```
> foldr (++) [] [...]
```

Sin embargo el esquema anteriormente explicado no siempre es lo que buscamos o necesitamos. Por ejemplo, tenemos la lista [2, 3, 7] y deseamos obtener el número 237. Esto nos llevaría a realizar el cálculo siguiente ((0 * 10 + 2) * 10 + 3) * 10 + 7 = 237.

```
numero xs = numeroc 0 xs
numeroc acum [] = acum
numeroc acum (x:xs) = numeroc (10 * acum + x) xs
```

Para reproducir este esquema, que en vez de asociarse por la derecha, es por la izquierda, tenemos la función foldl:

```
foldl::(a->b->b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

La diferencia principal con respecto a foldr, es que el parametro e es el acumulador de la función, con lo que la función para transformar la lista en un número quedaría como:

```
numero xs = foldl inc 0 xs
   where inc x y = 10 * x + y
```

Pero estas dos funciones siguen sin cubrir todos los casos posibles:

```
maximo::Ord a => [a] -> a
maximo [X] = X
maximo (x:xs) = max x (maximo xs)
```

La mayor diferencia que podemos apreciar es que en el caso base lo que tenemos es una lista de un solo elemento, ya que una lista vacia no tiene sentido aplicarle esta función, ya que al no tener elementos, no existe un máximo. Ello nos lleva a tener una nueva función fold:

```
foldr1::(a->a->a) -> [a] -> a
foldr1 f [X] = X
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Del mismo modo tendremos su versión para asociar por la izquierda:

```
foldl1::(a->a->a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

En esta función llamamos a foldl, porque el funcionamiento va a ser el mismo, lo único que hay que hacer es indicar como caso base el primer elemento de la lista.

## 3. Otros ejemplos de orden superior no predefinidos:

¿Como puedo hacer pruebas automáticamente para ver si mi función es como la función-solución dada por el profesor?

```
tester::Eq b => (a -> b) -> (a -> b) -> [a] -> Bool
tester f g xs = and comparados
    where trasf = map f xs
          trasg = map g xs
          comparados = zipWith (==) trasf trasg
```

```
>tester f1 f2 [0, 3, 8, 23, 9, 16, 100]
```

(Nota: No se puede comparar que dos funciones son iguales, pero sí puedo probarlas automáticamente para muchos casos.)

Ahora veamos en qué casos fallamos, para ello miraremos cuales dan resultados distintos, y los meteremos en una lista para saber cuales son exactamente:

```
tester2::Eq b => (a -> b) -> (a -> b) -> [a] -> Bool
tester2 f g xs = map snd malos
    where trasf = map f xs
          trasg = map g xs
          comparados = zipWith (/=) trasf trasg
          compyEnt = zip comparados xs
          malos = filter fst compyEnt
```

Otro ejemplo es el divide y vencerás genérico en Haskell:

```
dv::(a -> Bool) -> (a -> b) -> (a -> [a]) -> (a -> [b] -> b) -> a -> b
dv trivial solve split combine prob
    | trivial prob = solve prob
    | otherwise = combine prob subSols
          where subProbs = split prob
                subSols = map (dv trivial solve split combine) subProbs
```

Esto puede hacerse con cualquier otro esquema visto en MTP como el de ramificación y poda, backtracking, minimax, alphabeta.

Visto ya como se programa normalmente en la programación funcional, es hora de ver como se simula el while y el repeat en Haskell. No obstante, esta no es la forma natural de programar en este lenguaje, y es mucho menos eficiente.

```
while::(a -> Bool) -> (a -> a) -> a -> a
while test body state
    | test state = while test body (body state)
    | otherwise = state
```

```
repeat::(a -> Bool) -> (a -> a) -> a -> a
repeat test body state
    | test result = result
    | otherwise = repeat test body result
```

```
        where result = body state
```

Ahora un ejemplo de como implementar el algoritmo del minimax con podas alphabeta:

```
minimax::Int -> (a->[a]) -> (a->Valor) -> ([Valor]->Valor) -> ([Valor]->Valor) -> a -> Valor
minimax depth expandir evaluar peor mejor problema
    | (depth == 0) || (null siguientes) = evaluar problema
    | otherwise = mayor (map (minimax (depth-1) expandir evaluar mejor peor)
                    siguientes)
        where siguientes = expandir problema
```

Esta función nos da el valor de la mejor jugada. Si queremos la jugada concreta, hay que modificar el primer nivel del esquema, como ocurría con tester2.

## 4. Listas intensionales:

El orden superior nos trae ventajas como la reutilización y programas más cortos. ¿Podemos usar notaciones que acorten más depterminados cómputos? La respuesta es sí, con las listas intensionales (comprension lists):
+ Son una notación compacta para manipular listas (sólo listas).
+ Se traducen automáticamente al uso de funciones predefinidas (map, filter, concat).

Ejemplos:

```
[f x | x <- xs] → map f xs

[(x, y) | x <- xs, y <- ys] → Este ejemplo no hace lo mismo que la función zip,
   sino una combinación de todas las parejas de elementos de un conjunto con
   el otro: x = [1, 2], y = [3, 4], resultado = [(1, 3), (1, 4), (2, 3), (2, 4)]
[(x, y) | x <- [1, 3], y <- [2, 4, 5]] = [(1, 2), (1, 4), (1, 5), (3, 2), (3, 4), (3, 5)]

[x | x <- xs, p x] → filter p xs
[x | x <- [1..10], even x] = [2, 4, 6, 8, 10]
[f x | x <- xs, p x] → map f (filter p xs)
(Nota: Con [1..10] creamos una lista con los valores del 1 al 10.)

Ahora recordaremos el ejemplo del Quick Sort que vimos al principio:
qsort::Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y < x] ++ [x] ++ qsort [y | y<-xs, y >= x]
```

Ejemplos más complejos:

```
[(x, y) | x <- [1..3], y <- [1..x]] = [(1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3)]

El 2º generador depende del 1º, por lo que podemos poner tantos generadores y
condiciones como sea precisa.

[(x, y) | x <- [1..4], even x, y <- [x+1..4], odd y] = [(2, 3)]
        Si de la lista x = [1, 2, 3, 4], x es par, entonces
            Si de la lista y = [x+1..4], y es impar, entonces
```

Generar la lista [(x, y)]

¿Devolverá lo mismo si las condiciones se ponene al final?
[(x, y) | x <- [1..4], y <- [x+1..4], even x, odd y] = [(2, 3)]

Sí devuelve lo mismo, sin embargo la segunda forma es más ineficiente que la primera, ya que computa casos que luego serán descartados al final. Otros ejemplos de mismo resultado, pero se tarda más de una forma que de otra:

+ En el 1er caso: genero x's, podo x's, genero y's, podo y's. (Solo genero las parejas con la x "buena").
+ En el 2º caso: genero x's, genero y's, podo x's, podo y's. (Genero todas las parejas, y luego son filtradas).

(Nota: Se puede comprobar el número de reducciones de una función con el comando ":s +s" en el interprete hugs.)

La sintaxis completa:

[<expresión> | <gen_cond>, ...]

<gen_cond> =
Generador: <variable> <- <lista>
<patrón>
Introduce variables nuevas.

Restricción: <expresión booleana>
Restringe las expresiones resultantes

+ Una variable no se puede introducir más de una vez.
+ Un generador sólo puede depender de variables introducidas por generadores anteriores.
+ Una restricción solo puede depender de las variables introducidas por generadores anteriores.

Más ejemplos:

Todos los divisores positivos de un natural n:

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

¿Es primo un número natural n?

```
primo n = divisores n == [1, n]
```

Otra forma posible de hacer lo mismo:

```
primo n = [] == [d | d <- [2..raiz], n `mod` d == 0]
        where raiz = floor(sqrt(fromInteger n))
```

(Observación: primo 386 no necesita generar todos los divisores. En cuanto se encuentra el 2 el compilador deteca que [] != [2, _] y no sigue buscando. ¡No hace falta vigilar cuando deja de buscar!)

El problema de las 8 reinas:

```
ochoreinas::[[Int]]
ochoreinas = queens 8
```

```
queens::Int -> [[Int]]
queens 0 = [[]]
queens n = [previo ++ [nuevo] | previo <- queens (m-1), nuevo <- [1..8],
              noAtaques previo nuevo]

noAtaques::[Int] -> Int -> Bool
noAtaques p n = and [comprueba (i, j) (m, n) | (i, j) <- zip [1..] p]
      where m = length p + 1

comprueba (i, j) (m, n) = (j /= n) && (i + j /= m + n) && (i - j /= m - n)
```

¿Como podemos traducir las listas intensionales a otras funciones predefinidas?

Caso 1: [expresion | x <- xs]

```
map f xs
      where f x = expresion
```

Caso 2: [expresion | x <- xs, y <- ys, ...]

```
concat [[expresion | y <- ys, ...] | x <- xs]
```

Caso 3: [expresion | x <- xs, condicion, ..]

```
[expresion | x <- filter p xs, ...]
      where p x = condicion
```

## Tema 4: Entrada/Salida (y otros problemas).

- El resultado de una función sólo depende de los datos de entrada:
  - Ventaja: No hay efectos laterales.
  - Desventaja: No hay efectos laterales.
- Supongamos una función getChar para leer un caracter de la entrada estándar ¿Cual es su tipo?

  1er intento:

  getChar::Char

  (No necesita ningún dato de entrada, solo uno de salida.)

  Problema: Si llamo varias veces a getChar siempre leería lo mismo, por lo que estaría mal su implementación.

  2º intento:

  Necesitamos algún parámetro, ¿pero cuál? ¿De qué depende el valor que leeremos? De lo que se le pase por la cabeza al usuario, esto es algo impredecible. Una posible solución a esto sería usar como parametro “el estado del universo”:

  getChar::Universo -> Char

  Cada vez que usemos getChar puede devolver cosas distintas si el estado del universo cambia. Problemas:
  - ¿Como representamos el universo?
  - ¿Como cambia el universo con nuestras acciones?

3er intento:
getChar::Universo -> (Char, Universo)

Objetivo: Las funciones que necesitan interactuar con el universo, lo tienen como entrada y como salida. Es decir, dependen del universo y modifican el universo. El principal problema de todo esto es tener que ir arrastrando parametros de más por todo nuestro programa.

```
leerLinea::Universo -> (String, Universo)
leerLinea universo = if letra == '\n' then ("", uni2) else (letra:letras, uni3)
        when (letra, uni2) = getChar uni
                (letras, uni3) = leerLinea uni2
```

Este ejemplo no funciona en el interprete exactamente, pero ejemplifica como funcionaría la función de leer una linea con este sistema. Todo este sistema resulta un poco pesado, y la idea sería dejarlo a un lado, pero haciendo las cosas más sencillas.

4º intento:
En este último intento nuestra intención es ocultar la manipulación del parámetro ficticio. Para ello utilizaremos lo siguiente:

```
getChar::IO Char
```

Esto representa un nuevo tipo paramétrico "IO a", que indica que devuelve un valor de tipo "a", pero recordándonos que el cómputo necesita hacer E/S y que por tanto no es simplemente algo de tipo "a".

- ¿Qué operaciones tiene Haskell de tipo "IO a"?
  - getChar::IO Char → Lee un caracter del teclado.
  - getLine::IO String → Lee una cadena del teclado.
  - readFile::String -> IO String → Lee el contenido de un fichero de texto.
  - putChar::Char -> IO () → Escribe un caracter en pantalla.
  - putStr::String -> IO () → Escribe una cadena en pantalla.
  - putStrLn::String -> IO () → Escribe una cadena en pantalla con salto de linea.
  - writeFile::String -> String -> IO () → Escribe una cadena en un fichero.
  - print::Show a => a -> IO () → Escribe un dato en pantalla.

- ¿Como combinamos unas operaciones con otras?
  - (>>=)::IO a -> (a -> IO b) -> IO b
  - (>>)::IO a -> IO b-> IO b
  - return::a -> IO a

  Ejemplo:

```
ejemplo = getLine >>= putStrLn
-- Esto primero lee una cadena del teclado, y luego la escribe por pantalla.

verFich = putStr "Mete el nombre del fichero: " >> getLine >>= readFile >>= putStr
-- Funciona y hemos mejorado, pero sigue siendo pesado y engorroso de ver.
```

- ¿Como podemos hacer lo anterior de forma más cómoda?

Con lo que Haskell llama la notación do. Para ver como funciona veremos un ejemplo anterior, pero con esta nueva notación:

```
verFich = do putStr “Mete el nombre del fichero: “
             nombre <- getLine
             texto <- readFile nombre
             putStr texto
```

Esto viene a ser como la notación de los lenguajes imperativos, con <- en vez de =, pero en realidad, esto es traducido a las operaciones de concatenación de instrucciones de entrada/salida:

+ Por ello todo esto sigue siendo puramente funcional.
+ Se pueden seguir haciendo las mismas cosas que en cualquier programa funcional: pasar funciones como parámetro, etcétera.
+ Ninguna operación tiene tipo IO Char -> Char, ni nada parecido. Es decir, una vez que entramos en el mundo de la E/S no podemos olvidar que hemos hecho E/S.

Ejemplos:

```
insiste::(String -> String) -> IO ()
insiste f = do putStrLn “Dame una frase (vacía para acabar): “
               linea <- getLine
               if linea /= “” then do putStrLn (f linea)
                                      insiste f
                               else return ()
```

La última linea es necesaria, porque el operador if-then-else, siempre tiene que devolver un valor en caso de ser falsa la condición. Así que se opta por devolver lo que en Haskell representaría un “void”. Otro ejemplo sería un menú:

```
menu::IO ()
menu = do putStrLn “1: Dar vuelta a frases”
          putStrLn “2: Pasar frases a mayúsculas”
          putStrLn “3: Dar la longitud de las frases”
          putStrLn “4: Salir”
          opcion <- getLine
          case head (opcion) of
               ‘1’: do insiste reverse
                       menu
               ‘2’: do insiste (map toUpper)
                       menu
               ‘3’: do insiste (show . length)
                       menu
               ‘4’: return ()
```

En la 2ª opción al enviar como función (map toUpper), lo que estamos haciendo es configurar la función map, de tal modo que lo único que hace falta pasarle de más es el último parámetro.

Dado un fichero, alinear cada una de sus líneas a la derecha:

```
justificaDer::Int -> String -> String
justificaDer ancho texto = map (justifLinDer ancho) lineas
    where lineas = lines texto
```

```
justifLinDer::Int -> String -> String
justifLinDer ancho linea = blancos ++ linea
        where numblancos = max 0 (ancho - length linea)
                blancos = replicate numblancos ' '

-- replicate numblancos ' ' = [' ' | x <- [1..numblancos]]

justificaFichero = do putStr "Dame el nombre del fichero: "
                      fichero <- getLine
                      putStr "Dame el ancho máximo de la linea: "
                      anchoTexto <- getLine
                      texto <- readFile fichero
                      let ancho = read anchoTexto
                          textoFormateado = justificaDer ancho texto
                          ficheroSalida = fichero ++ ".formateado"
                      in writeFile ficheroSalida textoFormateado
```

- Restricciones en el uso de la notación para la E/S:
  - Cada línea debe contener una acción E/S.
  - Todas las líneas empiezan con el mismo sangrado (o más sangrado en las líneas inferiores).
  - La cláusula "where" puede usarse con las mismas restricciones que en el resto de los programas. Eso significa que no puede usarse en el "where" ninguna variable creada por las acciones dentro del do:

```
funcionES = do linea <- getLine
                  putStr nuevaLinea
          where nuevaLinea = reverse linea -- Esto es totalmente incorrecto.
```

  - La cláusula "let" puede utilizarse para resolver estas situaciones:

```
funcionES = do linea <- getLine
                  let nuevaLinea = reverse linea
                  in putStr nuevaLinea
```

Sintaxis del let:

```
let variable = valor
    ...
in sentencia
```