

SESIÓN 3

INTRODUCCIÓN WINHUGS



Universidad de Huelva

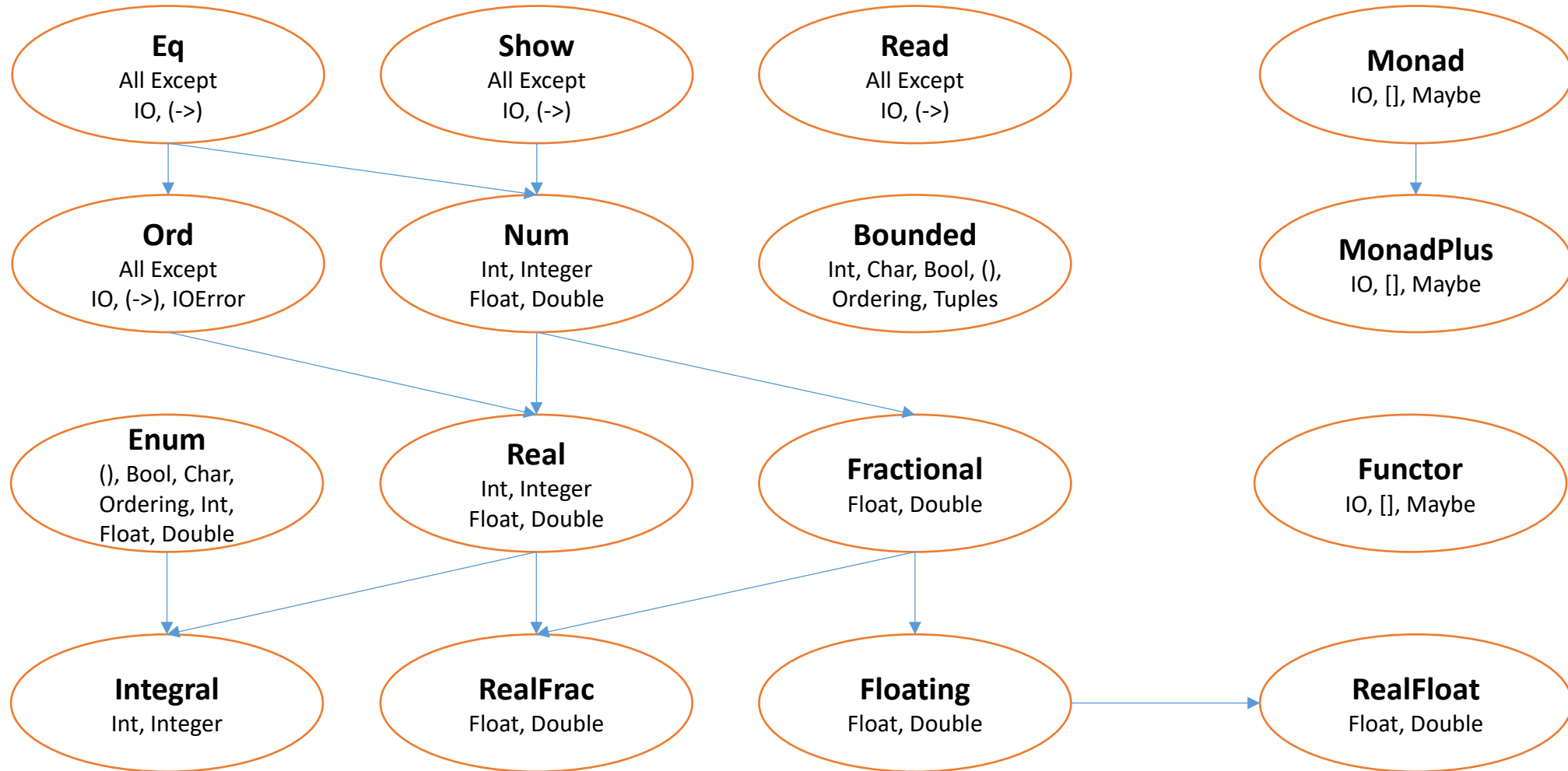
REPASO: SISTEMA DE TIPADO

Haskell es un lenguaje **fuertemente tipado**

Haskell utiliza **inferencia de tipos**

Haskell utiliza **tipos de datos jerárquicos (type classes)**

- Conjunto de tipos que comparten un comportamiento común, definido por una serie de funciones o métodos. Permiten especificar restricciones sobre qué tipos se pueden usar ciertas funciones.



Clase de Tipos Eq

Propósito: Define la igualdad y desigualdad entre valores.

Métodos:

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

Usos: Permite comparar valores para verificar si son iguales o diferentes

Eq
All Except
IO, (->)

Clase de Tipos Show

Propósito: Convierte un valor a su representación en cadena de texto.

Métodos:

`show :: a -> String`

Usos: Permite mostrar valores como cadenas, útil para depuración y visualización

Show

All Except
IO, (->)

Clase de Tipos Ord

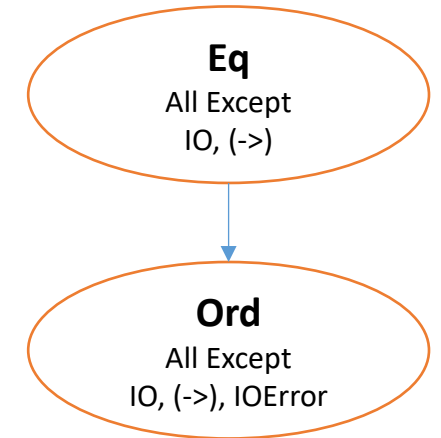
Propósito: Proporciona comparaciones de orden entre valores (menor, mayor)..

Herencia: Eq

Métodos:

$(<), (<=), (>), (>=) :: a \rightarrow a \rightarrow \text{Bool}$

Usos: Permite ordenar listas y comparar valores



Clase de Tipos Num

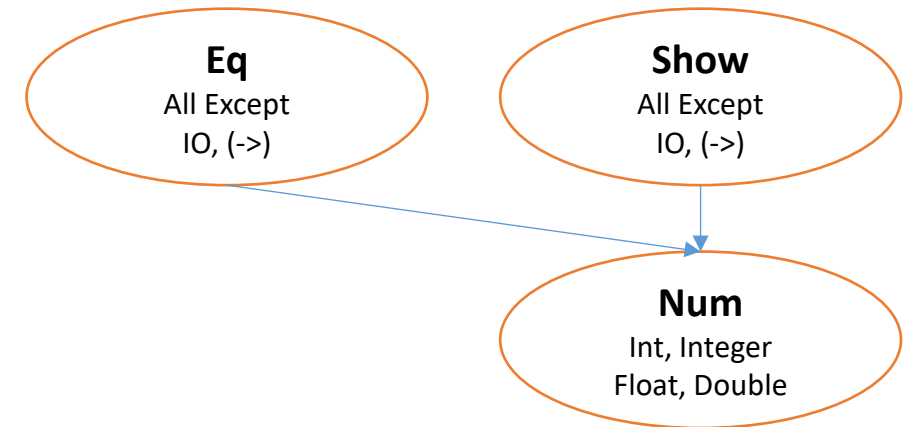
Propósito: Proporciona operaciones numéricas básicas (suma, multiplicación, etc.).

Herencia: Eq y Show

Métodos:

$(<), (<=), (>), (>=) :: a \rightarrow a \rightarrow \text{Bool}$

Usos: Permite ordenar listas y comparar valores



Clase de Tipos Enum

Enum

(), Bool, Char,
Ordering, Int,
Float, Double

Propósito: Define sucesión y predecesión, es decir, permite generar valores consecutivos.

Métodos:

`succ :: a -> a` (siguiente valor)

`pred :: a -> a` (valor anterior)

`toEnum :: Int -> a` (convierte desde un entero) `fromEnum :: a -> Int` (convierte a un entero)

`enumFrom :: a -> [a]` (genera una lista a partir de un valor)

Usos: Se utiliza en tipos que tienen una secuencia natural, como Int, Char, o Bool.

Clase de Tipos Real

Real
Int, Integer
Float, Double

Propósito: Representa números reales, que pueden convertirse en números racionales.

Herencia: Num

Métodos:

toRational :: a -> Rational (convierte un número real en una fracción racional)

Usos: Se usa para tipos que pueden ser tratados como números reales y que pueden convertirse en fracciones racionales.

Clase de Tipos Real

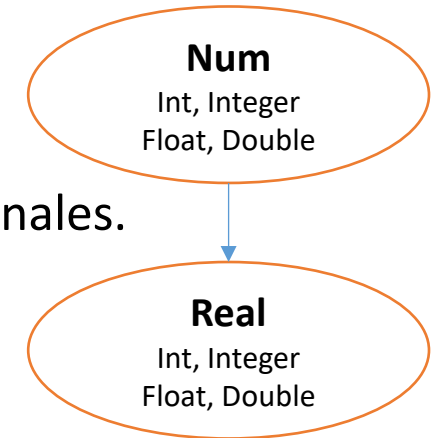
Propósito: Representa números reales, que pueden convertirse en números racionales.

Herencia: Num

Métodos:

toRational :: a -> Rational (convierte un número real en una fracción racional)

Usos: Se usa para tipos que pueden ser tratados como números reales y que pueden convertirse en fracciones racionales.



Clase de Tipos Fractional

Propósito: Define operaciones para números fraccionarios (decimales)

Herencia: Num

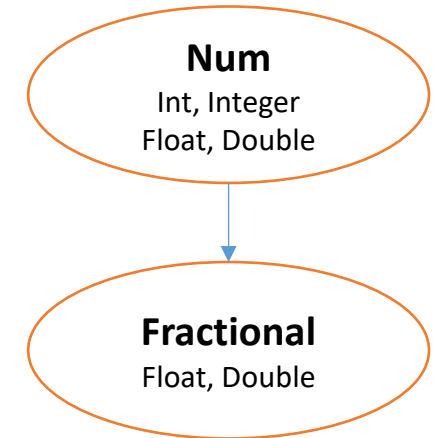
Métodos:

$(/) :: a \rightarrow a \rightarrow a$ (división)

$\text{recip} :: a \rightarrow a$ (inverso multiplicativo)

$\text{fromRational} :: \text{Rational} \rightarrow a$ (conversión desde un número racional)

Usos: Se utiliza en tipos que permiten divisiones y cálculos fraccionarios, como Float y Double



Clase de Tipos Integral

Propósito: Representa números enteros, proporcionando operaciones enteras

Herencia: Real y Num

Métodos:

`quot :: a -> a -> a` (cociente entero)

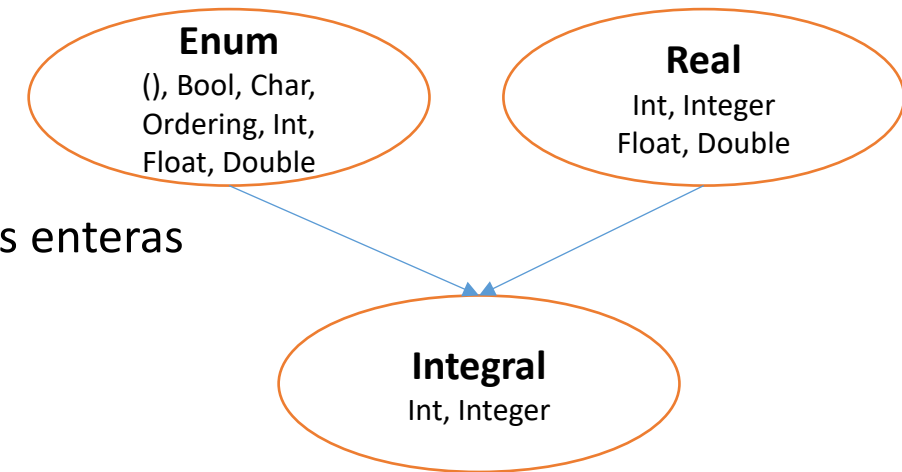
`rem :: a -> a -> a` (residuo entero)

`div :: a -> a -> a` (división entera)

`mod :: a -> a -> a` (módulo)

`toInteger :: a -> Integer` (convierte a Integer)

Usos: Se utiliza en tipos que permiten divisiones y cálculos fraccionarios, como Float y Double



Haskell es un lenguaje **fuertemente tipado**

Haskell utiliza **inferencia de tipos**

Haskell utiliza **tipos de datos jerárquicos (type classes)**

Operaciones lógicas: (&&), (||), not

Operaciones aritméticas: (+), (-), (*), (/), (^), (^^), (**)

Otras funciones incluidas en el Prelude:

- odd
- even
- gcd
- lcm
- subtract
- abs
- signum

DESCONECTA INTERNET

Retos evaluables: USAR SOLO LO VISTO HASTA AHORA, SIN INTERNET.


- **Ejercicio 1:** calcular la media de 3 *parámetros*
- **Ejercicio 2:** usando lo anterior, crear una función que indique si el resultado es par o impar (usa cualquiera de las funciones del prelude para pares e impares).
- **Ejercicio 3:** calcular el volumen de una esfera ($V = \frac{4}{3} \pi r^3$). (EN HASKELL EL NÚMERO PI SE DENOTA “pi”)
- **Ejercicio 4:** definir una funcion `maximoDouble` que devuelva el mayor de los numero pasados por parámetros de 5 valores y cuyo resultado sea del tipo *Double*



```
1 -- Definir la función media  
2 media :: Fractional a => a -> a -> a -> a  
3 media x y z = (x + y + z) / 3
```



```
1 -- Definir la función media
2 media :: Fractional a => a -> a -> a -> a
3 media x y z = (x + y + z) / 3
4
5 -- Definir la función media
6 mediaPar :: RealFrac a => a -> a -> a -> Bool
7 mediaPar x y z = even (round ((x + y + z) / 3))
```



```
1 -- Definir la función volumenEsfera
2 volumenEsfera :: Floating a => a -> a
3 volumenEsfera r = (4/3) * pi * r^3
```



```
1 -- Definir la función maximoDouble sin listas
2 maximoDouble :: Double -> Double -> Double -> Double -> Double -> Double
3 maximoDouble a b c d e = max a (max b (max c (max d e)))
```

SESIÓN 3: nuevos contenidos

¿Qué vamos a ver?

Listas: operadores, funciones básicas

Funciones sobre listas

Tuplas

Funciones sobre tuplas

3.2 LISTAS

Listas

Una lista es una colección ordenada de elementos del mismo tipo.

Haskell permite definir las listas como tipos compuestos a partir de cualquier otro tipo de dato.

Para **describir** listas se utilizan corchetes. Por ejemplo, el tipo `[Int]` representa una lista de valores de tipo `Int`.

En Haskell, las cadenas (*String*) son en realidad listas de caracteres `[Char]`.

Haskell define un valor especial que es la lista vacía, que se denota `[]`.

Listas

Se pueden definir listas como literales de cualquier tipo separados por coma y entre corchetes. **[1, 2, 3, 4]**.

Las cadenas son en realidad listas de caracteres. **“hola” => ['h','o','l','a']**.

Se pueden definir listas sobre tipos enumerados utilizando puntos suspensivos. **[1 .. 5] => [1,2,3,4,5]**.

Los números reales también son enumerados con +1.0 para el siguiente. **[1.0 .. 2.5] => [1.0, 2.0, 3.0]**.

Se puede modificar el incremento utilizando los dos primeros elementos de la lista. Por ejemplo, **[1.0, 1.25 .. 2.0]** genera la lista **[1.0, 1.25, 1.5, 1.75, 2.0]**. Por ejemplo, **[5, 4 .. 1]** genera **[5,4,3,2,1]**.

Operadores sobre listas

(:) : Permite crear una lista con un primer elemento y el resto de la lista. Por ejemplo, **1 : [2,3,4,5]** genera la lista **[1,2,3,4,5]**.

(++) : Permite concatenar dos listas. Por ejemplo, **[1,2,3] ++ [4,5]**.

(!!) : Obtiene el elemento i-ésimo de la lista. Por ejemplo, **[3..10]!!2 == 5**.

En realidad, el tipo de dato lista es una estructura con dos campos: el primer elemento y el resto de la lista.

Internamente, la lista **[1,2,3,4,5]** se representa como **1:2:3:4:5:[]**. ¿POR QUÉ?

Funciones básicas sobre listas:

head :: [a] -> a: Devuelve el primer elemento de una lista. [1..10]:[20..30]??

```
Hugs> head [1,2,3,4]
```

```
1 :: Integer
```

tail :: [a] -> [a]: Devuelve el resto de una lista. Genera error sobre la lista vacía.

```
Hugs> tail [1,2,3,4] -- Hugs> tail "Hola"
```

```
[2,3,4] :: [Integer] -- "ola" :: [Char]
```

```
Hugs> tail []
```

```
Program error: pattern match failure: tail []
```

```
Hugs>
```

Funciones básicas sobre listas:

length :: [a] -> Int : Devuelve la longitud de la lista.

```
Hugs> length [1,2,3,4]
```

```
4 :: Int
```

```
Hugs> length []
```

```
0 :: Int
```

null :: [a] -> Bool : Verifica si la lista está vacía.

```
Hugs> null [1,2,3,4,5]
```

```
False :: Bool
```

```
Hugs> null []
```

```
True :: Bool
```

Funciones básicas sobre listas:

last :: [a] -> a : Obtiene el último elemento de la lista.

```
Hugs> last [1,2,3,4,5]  
5 :: Integer
```

init :: [a] -> [a] : Obtiene la lista completa excepto el último elemento.

```
Hugs> init [1,2,3,4,5]  
[1,2,3,4] :: [Integer]
```

elem :: a -> [a] -> Bool : Verifica si un elemento pertenece a una lista.

```
Hugs> elem 4 [1,2,3,4,5]  
True :: Bool
```

Funciones básicas sobre listas:

notElem :: a -> [a] -> Bool : Verifica si un elemento no pertenece a una lista.

```
Hugs> notElem 4 [1,2,3,4,5]
```

```
False :: Bool
```

```
Hugs> notElem 8 [1,2,3,4,5]
```

```
True :: Bool
```

WINHUGS: comprobar las siguientes funciones y operadores

Operadores: (:), (++), (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

¿hemos probado a definir listas de listas?

¿Cómo obtengo el elemento 3.0 de [[2.5, 3.0], [4.8, 10.69, 9.12], []]?

- `Hugs> ([[2.5, 3.0], [4.8, 10.69, 9.12], []] !! 0) !! 1`
- `3.0 :: Double`

WINHUGS: comprobar las siguientes funciones y operadores

Operadores: (:), (++), (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

Implementar una función que elimine el primero y el ultimo de una cadena

```
[1,2,3,5]
```

```
Main> init (tail [1,2,3,5])
```

```
[2,3] :: [Integer]
```

Funciones sobre cadenas:

lines :: String -> [String] : Trocea las líneas de una cadena con saltos de linea.

```
Hugs> lines "aa\nbb\nbb"  
[ "aa" , "bb" , "bb" ] :: [String]
```

unlines :: [String] -> String : Une las cadenas con el salto de linea.

```
Hugs> unlines [ "aa" , "bb" , "cc" , "dd" , "ee" ]  
"aa\nbb\ncc\ndd\nnee\n" :: [Char]
```

Funciones sobre cadenas:

words :: String -> [String] : Trocea las palabras de una cadena.

```
Hugs> words "aa bb cc \t dd \n ee"  
[ "aa" , "bb" , "cc" , "dd" , "ee" ] :: [String]
```

unwords :: [String] -> String : Une las cadenas con el espacio.

```
Hugs> unwords [ "aa" , "bb" , "cc" , "dd" , "ee" ]  
"aa bb cc dd ee" :: [Char]
```

Funciones sobre cadenas:

and :: [Bool] -> Bool : Verifica que todos los elementos son True.

```
Hugs> and [(1==1),True,True,True]
```

```
True :: Bool
```

```
Hugs> and [(1==1),True,False,True]
```

```
False :: Bool
```

```
Hugs> and [(1==2),True,True,True]
```

```
False :: Bool
```

Funciones sobre cadenas:

or :: [Bool] -> Bool : Verifica que alguno de los elementos es True.

```
Hugs> or [(1==1),False,False,False]
```

```
True :: Bool
```

```
Hugs> or [(1==2),False,False,False]
```

```
False :: Bool
```

Funciones sobre cadenas:

any :: (a -> Bool) -> [a] -> Bool : Devuelve True si algún elemento de una lista verifica una función.

```
Hugs> any (pred(5)==) [0,1,2,3,4,5]
```

```
True :: Bool
```

```
Hugs> any (pred(5)>) [0,1,2,3,4,5]
```

```
True :: Bool
```

```
Hugs> any (pred(9)>) [0,1,2,3,4,5]
```

```
True :: Bool
```

```
Hugs> any (pred(9)<) [0,1,2,3,4,5]
```

```
False :: Bool
```

Funciones sobre cadenas:

all :: (a -> Bool) -> [a] -> Bool : Devuelve True si todos los elementos de una lista verifican una función.

```
Hugs> all (<10) [1,3,5,7,9]
```

```
True :: Bool
```

```
Hugs> all (==1) [1,1,0,1,1]
```

```
False :: Bool
```

```
Hugs> all even [2,4,6,8,10]
```

```
True :: Bool
```

WINHUGS

Operadores: (:) , (++) , (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

lines, unlines, words, unwords, and, or , any, all

Realizar ejemplos con Winhugs

Funciones sobre listas de números:

sum :: [Num] -> Num : Calcula el sumatorio de los elementos.

```
Hugs> sum [1,2,3,4]
```

```
10 :: Integer
```

product :: [Num] -> Num : Calcula el producto de los elementos.

```
Hugs> product [1,2,3,4]
```

```
24 :: Integer
```

```
.
```

Funciones sobre listas de números:

maximum :: [Ord] -> Ord : Calcula el máximo de los elementos.

```
Hugs> maximum [3,2,6,4,1,2,3]
```

```
6 :: Integer
```

```
Hugs> maximum "Feroz"
```

```
'z' :: Char
```

```
Hugs> maximum ['a','b','c']
```

```
'c' :: Char
```

Funciones sobre listas de números:

minimum :: [Ord] -> Ord : Calcula el mínimo de los elementos.

```
Hugs> minimum [3,2,6,4,1,2,3]
```

```
1 :: Integer
```

```
Hugs> minimum "Feroz"
```

```
'F' :: Char
```

```
Hugs> minimum ['a','b','c']
```

```
'a' :: Char
```

Funciones que generan listas:

repeat :: a -> [a] : Genera una lista ilimitada repitiendo el elemento.

```
Hugs> take 4 (repeat 3)
```

```
[3,3,3,3] :: [Integer]
```

```
Hugs> take 6 (repeat 'A')
```

"AAAAAA" :: [Char]

```
Hugs> repeat 3
```

¿?

[illegible]

Funciones que generan listas:

replicate :: Int -> a -> [a] : Genera una lista repitiendo n veces el elemento.

```
Hugs> replicate 3 5
```

```
[5,5,5] :: [Integer]
```

```
Hugs> replicate 5 "aa"
```

```
["aa","aa","aa","aa","aa"] :: [[Char]]
```

```
Hugs> replicate (succ 8) 4
```

```
[4,4,4,4,4,4,4,4] :: [Integer]
```

Funciones que generan listas:

cycle :: [a] -> [a] : Genera una lista ilimitada repitiendo la lista inicial.

```
Hugs> take 10 (cycle [1,2,3])  
[1,2,3,1,2,3,1,2,3,1] :: [Integer]
```

```
Hugs> take 10 (cycle "ABC")  
"ABCABCABCA" :: [Char]
```

Funciones que generan listas:

iterate :: (a -> a) -> a -> [a] : Aplica reiteradamente una función a partir de un valor inicial generando una lista ilimitada.

```
Hugs> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512] :: [Integer]
Hugs> take 10 (iterate (+3) 34)
[34,37,40,43,46,49,52,55,58,61] :: [Integer]
Hugs> take 10 (iterate (succ) 34)
[34,35,36,37,38,39,40,41,42,43] :: [Integer]
```

WINHUGS

Operadores: (:) , (++) , (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

lines, unlines, words, unwords, and, or , any, all

sum, product, maximu, mínimo, repeat, replicate, cycle, iterate

Realizar ejemplos con Winhugs

Funciones que transforman listas:

map :: (a -> b) -> [a] -> [b] : Aplica una función a los elementos de una lista.

```
Hugs> map abs [-1,-3,4,-12]
```

```
[1,3,4,12] :: [Integer]
```

```
Hugs> map reverse ["abc","cda","1234"]
```

```
["cba","adc","4321"] :: [[Char]]
```

```
Hugs> map (recip . negate) [1,4,-5,0.1]
```

```
[-1.0,-0.25,0.2,-10.0] :: [Double]
```

Funciones que transforman listas:

reverse :: [a] -> [a] : Devuelve la lista en sentido inverso.

```
Hugs> reverse [1..5]
```

```
[5,4,3,2,1] :: [Integer]
```

```
Hugs> reverse ["a","b","c"]
```

```
["c","b","a"] :: [[Char]]
```

Funciones que reducen listas:

filter :: (a -> Bool) -> [a] -> [a] : Selecciona los elementos de una lista que verifican una cierta función.

```
Hugs> filter (>5) [1,2,3,4,5,6,7,8]
```

```
[6,7,8] :: [Integer]
```

```
Hugs> filter odd [3,6,7,9,12,14]
```

```
[3,7,9] :: [Integer]
```

```
Hugs> filter (\x -> length x > 4) ["aaaa", "bbbbbbbbbbbbbb", "cc"]
```

```
["bbbbbbbbbbbbbb"] :: [[Char]]
```

Funciones que reducen listas: funciones de plegado (reducir a un valor)

¿Cuál es el patrón de funciones recursivas de plegado?

- Si el argumento es la lista vacía, se devuelve cierto **valor base** (correspondiente al **caso base** de la definición).
- En otro caso, se opera, mediante cierta *función u operador* (**plegador**), la cabeza de la lista y una llamada recursiva con la cola de la lista.

Funciones que reducen listas: funciones de plegado

¿Cuál es el patrón de funciones recursivas de plegado?

```
sumar :: [Integer] -> Integer  
sumar [ ] = 0  
sumar (x : xs) = x + sumar xs
```

Caso base

Cabecera

Plegador

Cola

Funciones que reducen listas: funciones de plegado

¿Cuál es el patrón de funciones recursivas de plegado?

funcion [] = z

funcion (x : xs) = x op (funcion xs)

z representa el valor base

op el plegador

Funciones que reducen listas: funciones de plegado -> foldr

foldr :: (a -> b -> a) -> a -> [b] -> a : A partir de un operador binario y un valor inicial, reduce una lista aplicando sucesivamente el operador de derecha a izquierda. $(a \text{ op } (b1 \text{ op } (b2 \text{ op } (\dots \text{ op } bn)))$

¿Cómo definiríamos nuestra función suma usando foldr?

```
sumar :: [Integer] -> Integer  
sumar = foldr (+) 0
```

Funciones que reducen listas: funciones de plegado -> foldr ¿cómo funciona?

```
foldr (+) 0 [1, 2, 3]
```

```
foldr (+) 0 (1 : (2 : (3 : [ ])))
```

```
=> (+) 1 (foldr (+) 0 (2 : (3 : [ ])))
```

```
=> (+) 1 ( (+) 2 (foldr (+) 0 (3 : [ ])))
```

```
=> (+) 1 ( (+) 2 ( (+) 3 (foldr (+) 0 [ ])))
```

```
=> 1 + (2 + (3 + 0))
```


Funciones que reducen listas:

foldr :: (a -> b -> a) -> a -> [b] -> a : A partir de un operador binario y un valor inicial, reduce una lista aplicando sucesivamente el operador de derecha a izquierda. $(a \text{ op } (b1 \text{ op } (b2 \text{ op } (\dots \text{ op } bn)))$

`= foldr (\x y -> 2*x + y) 4 [1, 2, 3]`

Realizar u seguimiento:

```
foldr (\x y -> 2*x + y) 4 [1, 2, 3]
```

```
foldr (\x y -> 2*x + y) 4 (1:2:3:[ ])
```

```
foldr (\x y -> 2*x + y) 1
```

```
    (foldr 4 (\x y -> 2*x + y) (2 : 3:[ ]))
```

```
foldr (\x y -> 2*x + y) 1
```

```
    (foldr (\x y -> 2*x + y) 2
```

```
        (foldr (\x y -> 2*x + y) 4 ((3:[ ]))))
```

```
foldr (\x y -> 2*x + y) 1  
  (foldr (\x y -> 2*x + y) 2  
    (foldr (\x y -> 2*x + y) 3  
      (foldr (\x y -> 2*x + y) 4 ([]) ) ) ) )
```

Hemos llegado al caso base, a partir de ahora... Resolveremos las expresiones de derecha a izquierda, por eso es un plegado a la derecha, hasta llegar a función más superior. ¿Cómo?

```
foldr (\x y -> 2*x + y) 1  
  (foldr (\x y -> 2*x + y) 2  
    (foldr (\x y -> 2*x + y) 3  
      (foldr (\x y -> 2*x + y) 4 ([]) ) ) )
```

Hemos llegado al caso base, a partir de ahora... Resolveremos las expresiones de derecha a izquierda, por eso es un plegado a la derecha, hasta llegar a función más superior. ¿Cómo?

```
foldr (\x y -> 2*x + y) 1  
    (foldr (\x y -> 2*x + y) 2  
        (foldr (\x y -> 2*x + y) 3 4 ) )
```

```
foldr (\x y -> 2*x + y) 1  
    (foldr (\x y -> 2*x + y) 2 10 )
```

```
foldr (\x y -> 2*x + y) 1 14
```

Resultado: 16

Funciones que reducen listas:

foldl :: (a -> b -> a) -> a -> [b] -> a : A partir de un operador binario y un valor inicial, reduce una lista aplicando sucesivamente el operador de izquierda a derecha. $((a \text{ op } b_1) \text{ op } b_2) \text{ op } b_3) \dots$

```
(\x y -> 2*x + y) 4 [1,2,3]
= foldl (\x y -> 2*x + y) 4 [1, 2, 3]
= foldl (\x y -> 2*x + y) ((\x y -> 2*x + y) 4 1) [2, 3]
= foldl (\x y -> 2*x + y) ((\x y -> 2*x + y) 9 2) [3]
= foldl (\x y -> 2*x + y) ((\x y -> 2*x + y) 20 3) []
= 43
```

Funciones que reducen listas:

foldl vs foldr:

```
Hugs> foldl (\x y -> 2*x + y) 4 [1,2,3]  
43 :: Integer
```

```
Hugs> foldr (\x y -> 2*x + y) 4 [1,2,3]  
16 :: Integer  
Hugs>
```

Funciones que reducen listas:

foldr1 :: (a -> a -> a) -> [a] -> a : Es similar a la función *foldr* pero tomando como valor inicial el primer elemento de la lista. $(b1 \text{ op } (b2 \text{ op } (\dots \text{ op } b_n))$

foldl1 :: (a -> a -> a) -> [a] -> a : Es similar a la función *foldl* pero tomando como valor inicial el primer elemento de la lista. $((b1 \text{ op } b2) \text{ op } b3) \dots$

Funciones que reducen listas:

scanr :: (a -> b -> a) -> a -> [b] -> [a] : Es similar a la función *foldr* pero genera una lista con los valores intermedios.

```
Hugs> scanr (+) 5 [1,2,3,4]
```

```
[15,14,12,9,5] :: [Integer]
```

```
Hugs> scanr (&&) True [1>2,3>2,5==5]
```

```
[False,True,True,True] :: [Bool]
```

scanr1 :: (a -> a -> a) -> [a] -> [a] : Es similar a *foldr1* generando la lista con los valores intermedios.

Funciones que reducen listas:

scanl :: (a -> b -> a) -> a -> [b] -> [a] : Es similar a la función *foldl* pero genera una lista con los valores intermedios. El último valor de la salida de **scanl** es el mismo valor que devuelve *foldl*.

```
Hugs> scanl (/) 64 [4,2,4]  
[64.0,16.0,8.0,2.0] :: [Double]
```

```
Hugs> scanl max 5 [1,2,3,4,5,6,7]  
[5,5,5,5,5,5,6,7] :: [Integer]
```

scanl1 :: (a -> a -> a) -> [a] -> [a] : Es similar a *foldl1* generando la lista con los valores intermedios.

Funciones que recortan listas:

take :: Int -> [a] -> [a] : Devuelve los n primeros elementos de la lista.

```
Hugs> take 5 [1,2,3,4,5,6,7]  
[1,2,3,4,5] :: [Integer]
```

drop :: Int -> [a] -> [a] : Elimina los n primeros elementos de la lista.

```
Hugs> drop 5 [1,2,3,4,5,6,7,8,9,10]  
[6,7,8,9,10] :: [Integer]
```

Funciones que recortan listas:

takeWhile :: (a -> Bool) -> [a] -> [a] : Devuelve los primeros elementos de una lista que verifican una función.

```
Hugs> takeWhile (<3) [1,2,3,4,5]  
[1,2] :: [Integer]
```

dropWhile :: (a -> Bool) -> [a] -> [a] : Elimina los primeros elementos de una lista que verifican una función.

```
Hugs> dropWhile (\x -> 6*x < 100) [1..20]  
[17,18,19,20] :: [Integer]
```

WINHUGS

Operadores: (:) , (++) , (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

lines, unlines, words, unwords, and, or , any, all

sum, producto, maximu, mínimo, repeat, replicate, cycle, iterate

map, filter, reverse, foldl, foldr, foldl1, foldr1, scanl, scanr, scanl1, scanr1, take, drop, takeWhile, dropWhile

Realizar ejemplos con Winhugs

EJEMPLOS

map (*2) [1, 2, 3, 4] Resultado: [2, 4, 6, 8]

filter even [1, 2, 3, 4, 5] Resultado: [2, 4]

reverse [1, 2, 3, 4] Resultado: [4, 3, 2, 1]

foldr (-) 2 [9, 2, 3, 4] Resultado: 8

foldl (-) 2 [9, 2, 3, 4] Resultado: -16

foldr1 (-) [1, 2, 3, 4] Resultado: -2

foldl1 (+) [1, 2, 3, 4] Resultado: -8

EJEMPLOS

scanl (-) 1 [1, 2, 3, 4] Resultado: [1, 0, -2, -5, -9]

scanr (-) 1 [1, 2, 3, 4] Resultado: [-1, 2, 0, 3, 1]

scanl1 (-) [1, 2, 3, 4] Resultado: [1, -1, -4, -8]

scanl1 (-) [1, 2, 3, 4] Resultado: [-2, 3, -1, 4]

take 3 [1, 2, 3, 4, 5] Resultado: [1, 2, 3]

drop 3 [1, 2, 3, 4, 5] Resultado: [4, 5]

takeWhile (< 4) [1, 2, 3, 4, 5] Resultado: [1, 2, 3]

EJERCICIOS

Crear una función que devuelva los valores mayores que 50 y menores que 100 de una lista infinita que comienza por 10 con incrementos de 10.

```
Hugs> take (10) [10,20..]  
[10,20,30,40,50,60,70,80,90,100] :: [Integer]  
Hugs> drop 5 (take (10) [10,20..])  
[60,70,80,90,100] :: [Integer]  
Hugs> takeWhile (<100) (drop 5 (take (10) [10,20..]))  
[60,70,80,90] :: [Integer]
```

¿Y si queremos los 80<?.

¿Y si queremos los 80<?.

```
Hugs> takeWhile (>80) (drop 5 (take (10) [10,20..]))  
[] :: [Integer]
```

```
Hugs> takeWhile (80>) (drop 5 (take (10) [10,20..]))  
[60,70] :: [Integer]
```

```
Hugs> takeWhile (>80) (reverse (drop 5 (take (10) [10,20..])))  
[100,90] :: [Integer]
```

Obtener los numeros divisibles por el parámetro que indiquemos de una lista de 100 números.

```
getlist :: Integral a => a -> [a]
```

```
Main> getlist y = filter (\x -> mod x y == 0) [1..100]
```

```
Main> getlist 10
```

```
[10,20,30,40,50,60,70,80,90,100] :: [Integer]
```

```
Main> getlist 5
```

```
[5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100] :: [Integer]
```

Implementar

- *Los siguientes ejercicios y*
- *diseñar 5 ejemplos nuevos utilizando combinación de las funciones vistas*

Fecha de entrega: hasta el 24 de Octubre

Nombre del fichero: Apellido1-Apellido2-Nombre-Practica1

EJERCICIOS COMUNES

cambia_el_primero(a,b): cambia el primer valor de la lista b por el valor de a

cambia_el_n(a,n,b): cambia el valor de la posición n de la lista b por el valor de a

get_mayor_abs(a): devuelve el mayor número en valor absoluto de la lista a

num_veces(a,b): devuelve la cantidad de veces que aparece el valor a en la lista b

palabras_mayores_n(n,a): devuelve una lista con las palabras mayores que n

EJERCICIOS COMUNES

es_palindroma palabra: comprueba si “palabra” es palíndroma

esprimo(x): Devuelve si el número introducido es primo o no

sumparesimp [x..z]: suma los pares y resta los impares de una lista

show_foldr_suma_n (n): muestra por pantalla los pasos del foldr (+) [1,2..n]

palindromas []: comprueba son palíndromas todas las palabras de una lista