

PRÁCTICA 3

INTRODUCCIÓN WINHUGS



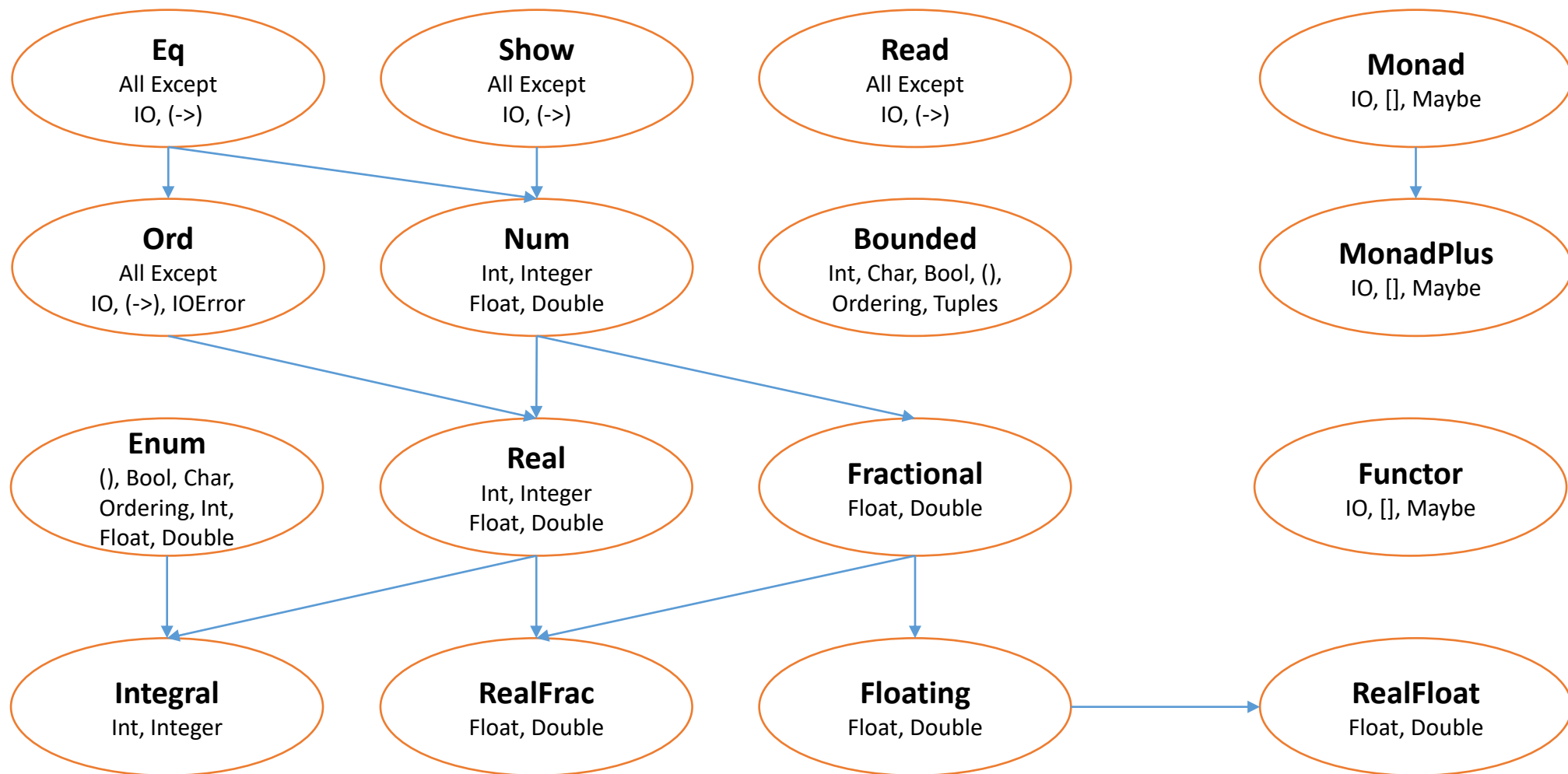
Universidad de Huelva

SISTEMA DE TIPADO

Haskell es un lenguaje **fuertemente tipado**

Haskell utiliza **inferencia de tipos**

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



Haskell es un lenguaje **fuertemente tipado**

Haskell utiliza **inferencia de tipos**

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

Operaciones lógicas y aritméticas

Breve introducción a listas

¿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:[]**.

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]  
[2,3,4] :: [Integer]  
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

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

Funciones

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

Realizar ejemplos con Winhugs

Funciones sobre cadenas:

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

```
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
```

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

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 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:

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:

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]`

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

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

...

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:

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

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

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 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 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

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] :: [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

- *5 ejercicios siguientes*
- *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 posicion n de la lista a por el valor de a

Get_mayor_abs(a): devuelve el mayor numero 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