

# SESIÓN 4

## INTRODUCCIÓN WINHUGS



Universidad de Huelva

# 4.1 REPASO Y EJEMPLOS P1

### WINHUGS

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

#### **Funciones**

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

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

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

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

**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 a por el valor de b

***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

### **WINHUGS: lo que puedo usar**

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

#### **Funciones**

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

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

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

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

### WINHUGS: ejercicio 1 en clase

**unico:** elimina los elementos repetidos

```
"a"++"j"++"l"++"fqdfnqhfchwqnfqwefwni"
```

```
"a"++"j"++"l"++"f"++"q"++"dnhcwnwewni"
```

```
...
```

```
"ajlfqdnhcwei"
```

## WINHUGS: ejercicio 1 en clase

**unico:** elimina los elementos repetidos

```
24
25  -- guarda en una lista todos los elementos diferentes de l
26
27  unico :: (Eq a) => [a] -> [a]
28
29  unico [] = []
30
31  unico l = (head l):(unico (filter (\x -> x /= (head l)) l))
32
33  -- unico "aaaaaajallfqldfnlqhfcldwqnfqwjefljwni" -> "ajlfqdnhcwei"
34
```



### WINHUGS: ejercicio en clase 2

**ordena\_menor\_a\_mayor:** ordena una lista de menor a mayor

```
Main> ordena_menor_a_mayor_insercion [1,5,8,3,4,9,0]
```

```
[0,1,3,4,5,8,9] :: [Integer]
```

```
Main> ordena_menor_a_mayor "asdfasdf"
```

```
"aaddffss" :: [Char]
```

## WINHUGS: ejercicio en clase 2

`ordena_menor_a_mayor`: ordena una lista de menor a mayor

```
ordena_menor_a_mayor :: (Ord a) => [a] -> [a]

ordena_menor_a_mayor [] = []
ordena_menor_a_mayor (cab:resto) = insertar cab (ordena_menor_a_mayor resto)

insertar :: (Ord a) => a -> [a] -> [a]
insertar x lista = takeWhile (<= x) lista ++ [x] ++ dropWhile (<= x) lista
```

¿Qué vamos a ver?

**Tuplas**

**Funciones sobre tuplas**

**Creación de funciones**

**Ejercicios**

## 4.2 TUPLES

Una tupla es un tipo de datos formado por una secuencia ordenada de elementos con una estructura y tamaño fijo.

En Haskell las tuplas se representan entre paréntesis y separadas por coma. Por ejemplo, **(1, True)**.

Una 2-tupla es una pareja de valores **(a, b)**. Una 3-tupla es un trío de valores **(a, b, c)**.

Son utilizadas cuando sabes exactamente cuántos valores tienen que ser combinados y su tipo depende de cuantos componentes tengan y del tipo de estos componentes.

Las tuplas pueden contener una combinación de valores de tipos iguales o de **distintos tipos**

`[(1,2),(8,11),(4,5)]`   `[(1,2,8),(8,11,5),(4,5,0)]`   `[(0,False),(1,True)]`   `[(0,'Novalido'),(1,'Valido')]`

## Funciones sobre tuplas:

**`fst :: (a,b) -> a`** : Obtiene el primer valor de una 2-tupla.

```
Hugs> fst(1,2)+fst(2,1)
```

```
3 :: Integer
```

**`snd :: (a,b) -> b`** : Obtiene el segundo valor de una 2-tupla.

```
Hugs> fst(1,2)+snd(2,1)
```

```
2 :: Integer
```

## Funciones sobre tuplas o que devuelven tuplas :

**splitAt** :: Int -> [a] -> ([a], [a]) : Divide una lista en un primer trozo de  $n$  elementos y un segundo trozo con el resto.

```
Hugs> splitAt 2 ['a','b','c','d']
```

```
( "ab" , "cd" ) :: ([Char],[Char])
```

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

```
( [1,2,3,4,5],[6,7,8,9,10] ) :: ([Integer],[Integer])
```

```
Hugs> splitAt 2 ['a','b','c']
```

```
( "ab" , "c" ) :: ([Char],[Char])
```

## Funciones sobre tuplas o que devuelven tuplas:

**span :: (a -> Bool) -> [a] -> ([a], [a])** : Divide una lista en un trozo con los primeros elementos que verifican una función y el resto de la lista.

```
Hugs> span (>"a") ["b", "c", "a"]  
(["b", "c"], ["a"]) :: ([[Char]], [[Char]])  
Hugs> span (<3) [1,2,4,5,6]  
([1,2], [4,5,6]) :: ([Integer], [Integer])  
Hugs> span (<"b") ["b", "c", "a"]  
([], ["b", "c", "a"]) :: ([[Char]], [[Char]])
```



## Funciones sobre tuplas o que devuelven tuplas :

**break** :: (a -> Bool) -> [a] -> ([a], [a]) : Divide una lista en un trozo con los primeros elementos que no verifican una función y el resto de la lista.

```
Hugs> break (<"b") ["b", "c", "a"]  
(["b", "c"], ["a"]) :: ([[Char]], [[Char]])  
Hugs> break (<"b") ["a", "b", "c"]  
([], ["a", "b", "c"]) :: ([[Char]], [[Char]])
```

### Funciones que empaquetan listas en tuplas:

**zip :: [a] -> [b] -> [(a, b)]** : Toma dos listas y genera una lista emparejando los elementos de ambas. Si las listas no tienen la misma longitud, los elementos que sobran se pierden.

```
Hugs> zip [1,2,3] [9,8,7]
[(1,9),(2,8),(3,7)] :: [(Integer,Integer)]
Hugs> zip "abcde" [True,False]
[('a', True),('b', False)] :: [(Char,Bool)]
Hugs> zip [] ["ab", "cd"]
[] :: [(a,[Char])]
```

### Funciones que empaquetan listas en tuplas:

**zip3** :: [a] -> [b] -> [c] -> [(a, b, c)] : Es similar a *zip* pero tomando tres listas y generando 3-tuplas.

```
Hugs> zip3 [3,5,7,9] [2,4,6] [1,0,6,8]
```

```
[(3,2,1),(5,4,0),(7,6,6)] :: [(Integer,Integer,Integer)]
```

```
Hugs> zip3 [3,5,7,9] [2,4] [1,0,6,8]
```

```
[(3,2,1),(5,4,0)] :: [(Integer,Integer,Integer)]
```

## Funciones que modifican listas de tuplas:

**zipWith** :: (a -> b -> c) -> [a] -> [b] -> [c] : Es similar a *zip*, pero en vez de generar tuplas aplica la función sobre la pareja y almacena el resultado.

```
Hugs> zipWith (+) [3,5,4] [2,1,5,8]  
[5,6,9] :: [Integer]
```

```
Hugs> zipWith (:) "eje" ["emp","los"]  
["eemp","jlos"] :: [[Char]]
```

**zipWith3** :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d] : Es similar a *zipWith* pero trabajando sobre tres listas

### Funciones que empaquetan listas:

**unzip** :: [(a, b)] -> ([a], [b]) : Transforma una lista de parejas en una pareja de listas.

```
Hugs> unzip [(1,2),(2,3),(3,4)]  
([1,2,3],[2,3,4]) :: ([Integer],[Integer])
```

**unzip3** :: [(a, b, c)] -> ([a], [b], [c]) : Transforma una lista de tripletas en una tripleta de listas.

```
Hugs> unzip3 [(1,2,3),(2,3,4),(3,4,5)]  
([1,2,3],[2,3,4],[3,4,5]) :: ([Integer],[Integer],[Integer])
```

**Realizar ejemplos en WINHUGS**

**`fst, snd, splitAt, span, break, zip, zip3, zipWith, unzip, unzip3`**

# 10 MINUTOS

**¿Sabrías calcular el determinante de  $[(4,5),(4,7)]$ ?**

## 4.3 CREACIÓN DE FUNCIONES

## Orden de prioridad

La notación **infix** indica que el operador es infijo, es decir, que se escribe entre los operandos.

```
Hugs> 1 + 2
```

```
3 :: Integer
```

```
Hugs> + 1 2
```

```
ERROR - Syntax error in expression (unexpected token)
```

```
Hugs> (+) 1 2
```

```
3 :: Integer
```

```
Hugs> (<5) `map` [1,2,4,5,6]
```

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



## Orden de prioridad

En el caso de encadenar operadores no se define ninguna prioridad.

```
Hugs> 1 == 1 == 1
```

```
ERROR - Ambiguous use of operator "(==)" with "(==)"
```

```
Hugs> (1 == 1) == 1
```

```
ERROR - Cannot infer instance
```

```
*** Instance      : Num Bool
```

```
*** Expression   : (1 == 1) == 1
```

## Orden de prioridad

El resultado de evaluar la primera expresión es True. El error viene dado por la definición del operador “==”

```
Hugs> :info ==  
infix 4 ==  
(==) :: Eq a => a -> a -> Bool  -- class member
```

El operador se define para dos elementos del mismo tipo, que pertenecen a la clase Eq (equiparable).

## Orden de prioridad

La notación **Infixl** indica que, en caso de igualdad de precedencia se evaluará primero la izquierda.

```
Hugs> 1 - 2 - 1
```

```
-2 :: Integer
```

```
Hugs> (1 - 2) - 1  -- equivalente
```

```
-2 :: Integer
```

Si se quiere interpretar algo diferente es necesario utilizar los paréntesis

```
Hugs> 1 - (2 - 1)
```

```
0 :: Integer
```

## Orden de prioridad

La notación **Infixl** indica que, en caso de igualdad de precedencia se evaluará primero la izquierda.

$$x_1 \ x_2 \ \cdots \ x_{n-1} \ x_n \Rightarrow (\dots (x_1 \ x_2) \ \cdots \ x_{n-1}) \ x_n$$

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

```
43 :: Integer
```

```
(2 * (2 * (2 * 4 + 1) + 2) + 3)
```

```
(2 * (2 * 9 + 2) + 3)
```

```
(2 * 20 + 3)
```

```
43
```

## Orden de prioridad

La notación **Infixr** indica que, en caso de igualdad de precedencia se evaluará primero el operador que está más a la derecha.

```
Hugs> 2 ^ 1 ^ 2
```

```
2 :: Integer
```

```
Hugs> 2 ^ (1 ^ 2) -- equivalente
```

```
2 :: Integer
```

```
Hugs> (2 ^ 1) ^ 2
```

```
4 :: Integer
```

```
Hugs>
```

## Orden de prioridad

La notación **Infixr** indica que, en caso de igualdad de precedencia se evaluará primero el operador que está más a la derecha.

$$x_1 x_2 \cdots x_{n-1} x_n \Rightarrow x_1 (x_2 \cdots (x_{n-1} x_n) \dots) \Rightarrow \text{foldr } (\backslash x y \rightarrow 2 * x + y) 4 [1,2,3]$$

$$(2 * 1 + (2 * 2 + (2 * 3 + 4)))$$

$$(2 * 1 + (2 * 2 + (10)))$$

$$(2 * 1 + (14)) = 16$$

## Orden de prioridad

El operador que mayor precedencia tiene es la composición de funciones “.”.

```
Hugs> succ . pred 4  
ERROR - Cannot infer instance  
*** Instance      : Enum (b -> a)  
*** Expression   : succ . pred 4
```

Error debido a que primero se realiza la operación `pred 4` (de resultado 3). Después se intenta realizar la composición del número 3 con la función `succ`. Para poder realizar una composición de funciones son necesarias dos funciones. No es posible componer una función con un número. De esta forma, ponemos de manifiesto que entre un operador (sea cual sea) y una función, primero se evaluará la función.

## Orden de prioridad

El operador que mayor precedencia tiene es la composición de funciones “.”.

```
Hugs> (succ . pred) 4
```

```
4 :: Integer
```

```
Hugs> (succ . succ) 'b'
```

```
'd' :: Char
```

```
Hugs> :info .
```

```
infixr 9 .
```

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

¿Cuál se evalúa primero? Y ¿por qué?



Notación	Prioridad	Operador
infixr	9	.
infixl	9	!!
infixr	8	$\wedge$ , $\wedge\wedge$ , **
infixl	7	*, /, `quot`, `rem`, `div`, `mod`
infixl	6	+, -
infixr	5	:
infixr	5	++
infix	4	==, /=, <, <=, >=, >, `elem`, `notElem`
infixr	3	&&
infixr	2	
infixl	1	>>, >>=
infixr	1	=<<
infixr	0	\$, \$!, `seq`

## Evaluación perezosa

Los lenguajes que utilizan esta técnica sólo evalúan una expresión cuando se necesita

```
Main> soloPrimero 4 (7/0)
4 :: Integer
```

```
1 | soloPrimero :: a -> b -> a
2 | soloPrimero x _ = x
```

El subrayado “\_” denota que se espera un parámetro pero que no se necesita nombrarlo ya que no se va a utilizar en el cuerpo de la función.

La expresión 7/0 en Haskell tiene el valor Infinito. La evaluación de la expresión anterior provocaría un error en la mayoría de los lenguajes imperativos, sin embargo en Haskell no se produce un error debido a que el segundo argumento no llega a evaluarse por no ser necesario.

**NOTA: Notación (x:xs) -> cabecera:resto**

```
2  
3 suma :: [Int] -> Int  
4 suma [] = 0  
5 suma (x:xs) = x + sum xs  
6
```

```
Hugs> :reload
```

```
Main> suma [1,2,3,4]
```

```
10 :: Int
```

```
Main> suma [1..20]
```

## 4.4 SINTAXIS DE FUNCIONES

### Formas de definir una función

Tenemos diferentes posibilidades a la hora de definir funciones:

- Utilizando varias ecuaciones (escribiendo cada ecuación en una línea)
- Guardas (en inglés “guards”, barreras, defensas, “|”)
- If then else
- Case
- En definiciones locales

## Formas de definir una función: varias ecuaciones

Factorial de un número entero:

```
1
2 {- FACTORIAL VARIAS ECUACIONES -}
3
4 factorial_ecuaciones :: Int -> Int
5 factorial_ecuaciones 0 = 1
6 factorial_ecuaciones n = n * factorial_ecuaciones (n-1)
7
```

¿Existen diferencias? ¿Cuál?

factorial\_ecacuaciones2 (-10)

```
{- FACTORIAL VARIAS ECUACIONES -}

factorial_ecuaciones2 :: Integral a => a -> a
factorial_ecuaciones2 0 = 1
factorial_ecuaciones2 n = n * factorial_ecuaciones2 (n-1)
```

## Formas de definir una función: utilizando guardas

Factorial de un número entero:

```
{- FACTORIAL GUARDAS -}  
  
--factorial_guarda ::Integer->Integer  
--factorial_guarda :: (Num a, Ord a) => a -> a  
factorial_guarda :: (Num a, Ord a) => a -> a  
factorial_guarda n  
  | n ==0    = 1  
  | n > 0    = n * factorial_guarda(n-1)  
  | otherwise = error "valor negativo"
```

## Formas de definir una función: utilizando guardas

Comprobar signo de un número:

```
24
25  --positivo :: (Num a, Ord a) => a -> Bool
26  positivo x
27      | x > 0 = True
28      | x < 0 = False
29      | otherwise = True
```



## Formas de definir una función: if then else

Factorial de un número entero / posicionar en intervalo:

```
30
31 {- FACTORIAL IF THEN ELSE -}
32
33 --factorialIF :: Num a => a -> a
34 factorialIF n = if (n==0) then 1 else n*factorialIF (n-1)
35
```

## Formas de definir una función: if then else

Factorial de un número entero / posicionar en intervalo:

```
{- INTERVALO IF THEN ELSE -}  
  
--intervalo :: (Ord a, Num a) => a -> [Char]  
intervalo x =  
    if x > 75 then "Intervalo 4"  
    else if 50 < x && x <= 75 then "Intervalo 3"  
    else if 25 < x && x <= 50 then "Intervalo 2"  
    else "Primer intervalo"
```

## Formas de definir una función: CASE

case **expresion** of patron -> resultado

    patron -> resultado

    patron -> resultado

...

La expresión se compara con los patrones. Se utiliza el primer patrón que coincide con la expresión. Si no cubre toda la expresión del caso y no se encuentra un patrón adecuado, se produce un error de tiempo de ejecución.

## Formas de definir una función: CASE

Comprobar si un número es par

```
58
59  -- comprobar si numero par
60  case_par :: Integral a => a -> Bool
61  case_par n = case (mod n 2 == 0) of
62    |      |      |      |      True->True
63    |      |      |      |      False->False
```

## Formas de definir una función: CASE

Mini calculadora: sumar o restar en función de un parametro

```
65  -- operar segun parametro
66  sumar :: Num a => a -> a -> a
67  sumar x y = x + y
68  restar :: Num a => a -> a -> a
69  restar x y = x - y
70  operar :: (Num a, Num b) => b -> a -> a -> a
71  operar x = case x of
72      1 -> sumar
73      2 -> restar
```

## Formas de definir una función: CASE

Mostrar por pantalla si una lista es vacía, de 1 elemento, 2 elementos o mas

```
75  -- comprobar tipo de lista
76  tipolista :: [a] -> String
77  ▼ tipolista xs = "Resultado: " ++ case xs of []      -> "Lista vacia."
78                                     (x:[])    -> "Lista con un elemento."
79                                     (x:y:[])  -> "Lista de dos elementos."
80                                     (x:y:_)   -> "Lista con mas de dos elementos"
```

## Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
85  --divisible a y b
86  divisible :: Int -> Int -> Bool
87  divisible x y = resto == 0
88      where resto = mod x y
```

## Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
90  -- comprobar si cadena vacia
91  cadenaNoVacia :: [Char] -> Bool
92  cadenaNoVacia x = numwords > 0
93      where numwords = length (words x)
```



## Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
95  -- comprobar si lista vacia
96  listaNoVacía :: [a] -> [a] -> Bool
97  listaNoVacía x y = numitems > 0
98      where numitems = length (x ++ y)
```

**Realizar ejemplos en WINHUGS**

**RESTO DE LA CLASE**

## Realizar ejemplos en WINHUGS

- Dada una lista de enteros, devuelve la lista de los sucesores de cada entero.

```
sucesoresDeLista :: [Integer] -> [Integer]
sucesoresDeLista [] = []
sucesoresDeLista (x:xs) = x+1 : sucesoresDeLista xs
```

## Realizar ejemplos en WINHUGS

- Dada una lista de enteros, devuelve una lista con los elementos que son positivos

```
obtenerPositivos :: [Integer] -> [Integer]
obtenerPositivos [] = []
obtenerPositivos (x:xs)
  | x >= 0 = x : obtenerPositivos xs
  | otherwise = obtenerPositivos xs
```

## Realizar ejemplos en WINHUGS

- Redefinir la función until tal que until p f x aplica la f a x el menor número posible de veces, hasta alcanzar un valor que satisface el predicado p. `until (>1000) (2*) 1 => 1024`

```
hasta :: (a -> Bool) -> (a -> a) -> a -> a
hasta p f x =
    if p x then x
    else hasta p f (f x)
```

# PRÁCTICA 2

**Ejercicio 1:**

Definir la función `comb` tal que `comb n k` es el número de combinaciones de  $n$  elementos tomados de  $k$  en  $k$ ; es decir:

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

```
Main> comb 6 2  
15 :: Integer
```

**Ejercicio 2:**

Definir la función raíces tal que raíces a b c es la lista de las raices de la ecuación  $ax^2 + bx + c = 0$ .

```
Main> raices_1 1 (-2) 1  
[1.0,1.0] :: [Double]  
Main> raices_1 1 3 2  
[-1.0,-2.0] :: [Double]
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



### Ejercicio 3:

Los intervalos cerrados se pueden representar mediante -- una lista de dos números (el primero es el extremo inferior del intervalo y el segundo el superior).

Definir la función **interseccion** :: **Ord a => [a] -> [a] -> [a]**, tal que (interseccion i1 i2) es la intersección de los intervalos i1 e-- i2. Por ejemplo,

interseccion [] [3,5] == []

interseccion [3,5] [] == []

interseccion [2,4] [6,9] == []

interseccion [2,6] [6,9] == [6,6]

interseccion [2,6] [0,9] == [2,6]

interseccion [2,6] [0,4] == [2,4]

interseccion [4,6] [0,4] == [4,4]

interseccion [5,6] [0,4] == []

**Ejercicio 4:**

Comprobar la pertenencia a una lista usando una función recursiva.

```
Main> pertenece 5 [5,4]
```

```
True :: Bool
```

Hacer lo mismo para una lista de tuplas

#### **Ejercicio 5:**

Realizar 2 retos de Euler utilizando las mismas condiciones que en los ejercicios anteriores. Solo se puede usar lo visto hasta ahora e implementar al menos 3 sintaxis diferentes