

# SESIÓN 7

## INTRODUCCIÓN WINHUGS



**Universidad de Huelva**

# 7.1. REPASO

Entrada y Salida

Main

Bloque Do

Acciones

Ejercicios

```
1 juego :: IO ()
2 juego =
3     do putStrLn "Piensa un numero entre el 1 y el 100."
4         adivina 1 100
5         putStrLn "Fin del juego"
6
7 adivina :: Int -> Int -> IO ()
8 adivina a b =
9     do putStr ("Es " ++ show conjetura ++ "? [mayor/menor/exacto]
10    ")
11        s <- getLine
12        case s of
13            "mayor"  -> adivina (conjetura+1) b
14            "menor"  -> adivina a (conjetura-1)
15            "exacto" -> return ()
16            _        -> adivina a b
17    where
18        conjetura = (a+b) `div` 2
```

## 7.2. DEFINICION DE TIPOS

### Tipos vistos hasta ahora

**Int:** representa enteros. Se utiliza para representar número enteros. Int está acotado, tiene un valor máximo y un valor mínimo.

**Integer:** representa enteros. La diferencia es que no están acotados.

**Float:** representa un número real en coma flotante de precisión simple.

**Double:** representa un número real en coma flotante de doble precisión.

**Bool:** tipo booleano con solo dos valores posibles. *¿Cuál falta?*

**Char:** representa un caracter. Como veremos, ***String*** es un tipo “definido” a partir del tipo básico Char.

## Declaración *type*

La instrucción *type* permite asignar un alias a una declaración de tipo

La sintaxis es la siguiente:

```
typeDecl ::= type simpleType = expresion_de_tipo
```

El símbolo ***simpleType*** define el identificador del tipo y puede incluir variables de tipo.

```
simpleType ::= Identificador tvar1, tvar2 ... tvarN
```

Las variables incluidas en *simpleType* deben corresponder a las variables de tipo incluidas en la expresión.

### Declaraciones de tipos como sinónimos

Se puede definir un nuevo nombre para un tipo existente mediante una declaración de tipo.

Ejemplo: Las cadenas son listas de caracteres.

El nombre del tipo tiene que empezar por mayúscula

```
type String = [Char]
```

```
words :: String -> [String]
```

## Declaraciones de tipos como sinónimos

```
1 libretaDirecciones :: [(String,String)]
2 libretaDirecciones = [("Antonio","657123456"), ("Jesus","657123457"), ("Albaro","657123458")]
3
4 --Sinonimos
5 type Registro    = [(Nombre,Telefono)]
6 type Telefono    = String
7 type Nombre      = String
8
9 existeRegistro :: Nombre -> Telefono -> Registro -> Bool
10 existeRegistro nombre telefono libreta = (nombre,telefono) `elem` libreta
```

```
Main> existeRegistro "Antonio" "657123456" libretaDirecciones
True :: Bool
```



### Declaraciones de tipos nuevos

Las declaraciones de tipos pueden usarse para facilitar la lectura de tipos.

Las posiciones son pares de enteros:

origen es la posición (0,0)

(izquierda p) es la posición a la izquierda de la posición p.

```
type Pos = (Int,Int)
```

```
origen :: Pos  
origen = (0,0)
```

```
izquierda :: Pos -> Pos  
izquierda (x,y) = (x-1,y)
```

## Declaraciones de data o datos algebraicos

La forma convencional de definir un nuevo tipo de dato (lo que se conoce como datos algebraicos) es por medio de la instrucción data.

```
dataDecl ::= data [context =>] simpleType [= constrs] [deriving]  
simpleType ::= Identificador tvar1 tvar2 ... tvarn
```

El contexto es opcional y permite indicar las clases a las que pertenecen los variables de tipo.

## Declaraciones de data

La declaración de tipos de datos algebraicos se basa en definir constructores del tipo.

Por ejemplo, el tipo de dato *Bool* está basado en dos constructores: *True* y *False*. Los constructores de tipo utilizan identificadores que comienzan en mayúscula.

```
data Bool = False  
          | True
```

Los constructores sin argumentos permiten definir fácilmente enumeraciones:

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

## Declaraciones de data

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

```
Main> Miercoles
```

```
ERROR - Cannot find "show" function for:
```

```
*** Expression : Miercoles
```

```
*** Of type    : Dias
```

¿Cómo lo arreglamos?

### Declaraciones de data

```
data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo  
deriving (Show)
```

```
Main> :reload
```

```
Main> Miercoles
```

```
Miercoles :: Dias
```

## Declaraciones de data

```
data Dias = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
deriving (Eq,Show)
```

```
Main> :reload
Main> Miercoles == Miercoles
ERROR - Cannot infer instance
*** Instance      : Eq Dias
*** Expression   : Miercoles == Miercoles

Main> :reload
Main> Miercoles == Miercoles
True :: Bool
```

## Declaraciones de data

**Los constructores pueden tener argumentos.**

No es necesario que todos los constructores de un tipo tengan los mismos argumentos.

Por ejemplo: (Shape) que pueda tomar dos formas: un círculo o un rectángulo.

El círculo necesita la posición del centro y el radio

El rectángulo necesita la posición de una esquina y su contraria

```
data Shape = Circle Float Float Float  
           | Rectangle Float Float Float Float
```

Los constructores de tipo son funciones que toman los parámetros como entrada y generan un valor del tipo descrito

## Declaraciones de data: uso

```
1 data Shape = Circle Point Float | Rectangle Point Point
2
3 surface2 :: Shape -> Float
4 surface2 (Circle _ r) = pi * r ^ 2
5 surface2 (Rectangle (Point x1 y1) (Point x2 y2)) = (abs (x2 - x1)) * (abs (y2 - y1))
6
```

```
Main> surface (Circle 5 3 2)
```

```
12.56637 :: Float
```

```
Main> surface (Rectangle 5 6 4 5)
```

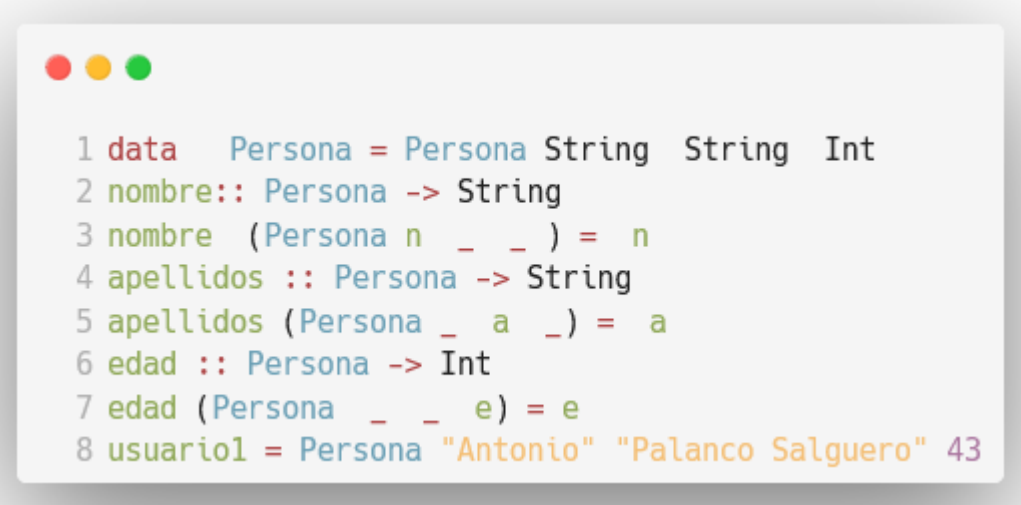
```
1.0 :: Float
```



## Declaraciones de data

Los tipos de datos algebraicos permiten definir estructuras de tipo registro. Por ejemplo, podríamos definir el tipo `Persona` que almacenara el nombre, apellidos y edad. Para acceder a estos campos podríamos definir funciones basadas en **patrones**.

```
Main> nombre (usuario1)
"Antonio" :: [Char]
Main> apellidos (usuario1)
"Palanco Salguero" :: [Char]
Main> edad (usuario1)
43 :: Int
```



```
1 data Persona = Persona String String Int
2 nombre :: Persona -> String
3 nombre (Persona n _ _) = n
4 apellidos :: Persona -> String
5 apellidos (Persona _ a _) = a
6 edad :: Persona -> Int
7 edad (Persona _ _ e) = e
8 usuario1 = Persona "Antonio" "Palanco Salguero" 43
```

## Declaraciones de data

Existe una sintaxis especial que permite definir las estructuras de tipo registro de una forma mucho más compacta. Este formato define directamente las funciones asociadas a los campos y permite indicar el valor de cada campo al construir el dato. El orden de los campos puede incluso ser diferente al de la definición del tipo.

```
Main> nombre(usuario2)  
"Juan" :: [Char]
```



```
1 data Persona = Persona { nombre :: String, apellidos :: String, edad :: Int }  
2 usuario2 = Persona { nombre = "Juan" , apellidos = "Nadie" , edad = 50 }
```

## Declaraciones de data

Las definiciones de tipos pueden ser recursivas: las expresiones de tipos de los constructores pueden utilizar el mismo tipo de dato que estamos definiendo. Por ejemplo, podemos definir un tipo de dato árbol con dos tipos de nodos: un nodo hoja que almacene un valor entero y un nodo interno que almacene dos ramas.

¿Y si queremos un valor en el elemento raíz?

```
1 data Arbol = Hoja Int | Nodo Arbol Arbol deriving (Eq, Show)
```

```
Main> arbol
```


```
Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3) :: Arbol
```

## Ejemplos

Crear el tipo RGB que define un color:  $([0..255], [0..255], [0..255])$

Definir colores: red (255,0,0), green(0,255,0) y blue(0,0,255)

Crear colores mezclados (la suma de cada una de las coordenadas):



```
1 data RGBColor = RGBColor Int Int Int deriving (Show)
2
3 red    = RGBColor 255 0 0
4 green  = RGBColor 0 255 0
5 blue   = RGBColor 0 0 255
6
7 mixColor (RGBColor r1 g1 b1) (RGBColor r2 g2 b2) = RGBColor (r1+r2) (g1+g2) (b1+b2)
```

## 7.3. PRACTICAR

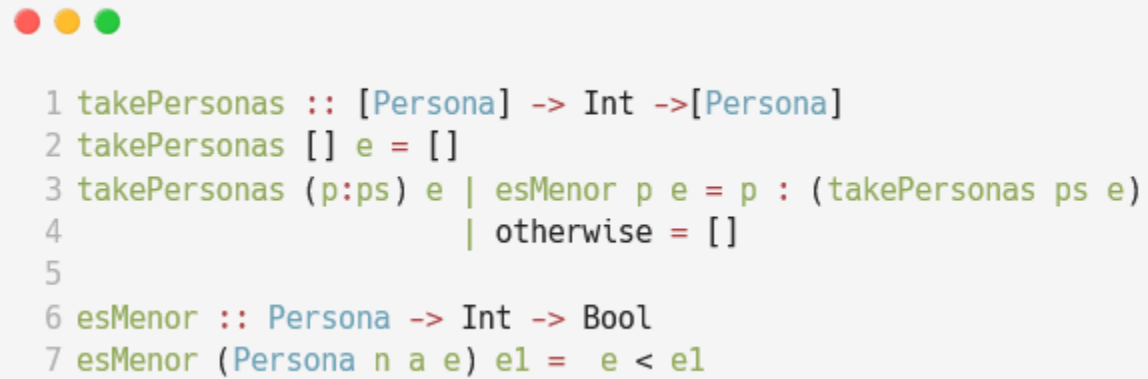
### Ejercicios

- Realizar un ejemplo de definición de tipos (como por ejemplo los datos de un préstamo de materiales: id, nombre, fk\_usuario, fecha\_salida, fecha\_entrada, observaciones). Crear las instancias de al menos 3 prestamos y mostrar por pantalla uno de ellos.
- Definir una lista de personas (al menos 5) con los siguientes campos: (Nombre,Apellidos,Edad). La lista debe definirse de forma ordenada por edad, de menor a mayor. Una vez hecho, realizar las siguientes funciones:
  - Obtener el segmento más largo de la lista con las personas de edad inferior a la dada
  - Buscar una persona por nombre.

## Ejercicios

```
1 data Persona = Persona { nombre :: String, apellidos :: String, edad :: Int } deriving (Show,Eq)
2 usuario1 = Persona { nombre = "n1" , apellidos = "a1" , edad = 39 }
3 usuario2 = Persona { nombre = "n2" , apellidos = "a2" , edad = 40 }
4 usuario3 = Persona { nombre = "n3" , apellidos = "a3" , edad = 55 }
5 usuario4 = Persona { nombre = "n4" , apellidos = "a4" , edad = 60 }
6 usuario5 = Persona { nombre = "n5" , apellidos = "a5" , edad = 65 }
7
8 personas = [usuario1, usuario2,usuario3,usuario4,usuario5]
```

## Ejercicios



```
1 takePersonas :: [Persona] -> Int -> [Persona]
2 takePersonas [] e = []
3 takePersonas (p:ps) e | esMenor p e = p : (takePersonas ps e)
4                           | otherwise = []
5
6 esMenor :: Persona -> Int -> Bool
7 esMenor (Persona n a e) e1 = e < e1
```



## Ejercicios

```
1 existe :: [Persona] -> String -> [Persona]
2 existe [] p1 = []
3 existe (p:ps) p1 = do
4     {
5         if esta p p1 then [p]
6         else existe ps p1
7     }
8
9 esta :: Persona -> String -> Bool
10 esta (Persona n a e) e1 = n == e1
```

### Ejercicios

Definir un árbol binario (una estructura de datos jerárquica donde cada nodo tiene como máximo dos hijos uno izquierdo y uno derecho) que almacena un valor entero en cada uno de sus nodos y la raíz.

Definir alguna de estas funciones.

- insertarNodo
- buscarNodo
- numeroNodos
- inOrden

## Ejemplos: árbol binario

```
1 -- Estructura del árbol
2 data Abb a = Vacio | Nodo a (Abb a) (Abb a) deriving (Show)
3
4 -- Insertar un nuevo nodo a un árbol definido
5 -- insertarNodo (Valor a insertar) (Árbol)
6 insertarNodo :: (Ord a) => a -> Abb a -> Abb a
7 insertarNodo nuevo Vacio = Nodo nuevo Vacio Vacio
8 insertarNodo nuevo (Nodo a izq der)
9   | nuevo <= a = Nodo a (insertarNodo nuevo izq) der
10  | nuevo > a  = Nodo a izq (insertarNodo nuevo der)
```

## Ejemplos: árbol binario

```
1 -- Devuelve True o False si el nodo se halla en el árbol
2 -- BuscarNodo (Valor a buscar) (Árbol)
3 buscarNodo :: (Ord a) => a -> Abb a -> Bool

5 buscarNodo buscado (Nodo a izq der)
6 | buscado == a = True
7 | buscado < a = buscarNodo buscado izq
8 | otherwise = buscarNodo buscado der
```

## Ejemplos: árbol binario



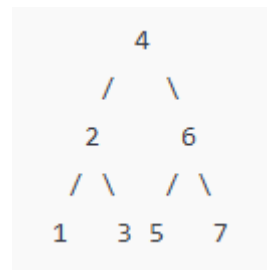
```
1 -- Devuelve el número de elementos en el árbol
2 -- numeroNodos (Árbol)
3 numeroNodos :: (Num a) => Abb t -> a
4 numeroNodos Vacio = 0
5 numeroNodos (Nodo _ izq der) = 1 + numeroNodos izq + numeroNodos der
```

## Ejemplos: árbol binario

El recorrido "en orden" (in-order) de un árbol binario visita los nodos en el orden: izquierda, raíz, derecha



```
1 -- Devuelve una lista de los nodos en recorrido inorden (izquierda, raíz, derecha)
2 -- inorden (Árbol)
3 inorden :: (Ord a) => Abb a -> [a]
4 inorden Vacio = []
5 inorden (Nodo actual izq der) = inorden izq ++ [actual] ++ inorden der
```



1, 2, 3, 4, 5, 6, 7