

# Definición de funciones

Dpto. Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla

# Definición de funciones

- El sangrado es importante (ver [este enlace](#))
- Composición (expresiones con operadores y funciones)
- Equiparación de patrones
- Condicionales
  - Instrucciones `if-then-else`
  - Ecuaciones con guardas
  - Instrucciones `case-of`

## Where

Definición de variables auxiliares dentro de una función.

- Ejemplo de la función *media*.

```
media :: Fractional a => [a] -> a
media xs = sum xs / (fromIntegral (length xs))
```

- Cuidado, *length* devuelve un *Int*. Se puede convertir a *Num* con la función:

```
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

- Se puede redefinir con *where*:

```
media :: Fractional a => [a] -> a
media xs = sum xs / n
  where n = (fromIntegral (length xs))
```

# if-then-else

- Instrucciones if-then-else

```
valorAbsolutoIf :: Int -> Int
valorAbsolutoIf n = if (n >= 0) then n else (-n)
```

- if-then-else anidados:

```
signo :: Int -> Int
signo n = if (n < 0)
           then (-1)
           else (if (n==0)
                   then 0
                   else 1)
```

# Guardas

- Ecuaciones con guardas: Haskell utiliza la primera que se verifica

```
valorAbsolutoG :: Int -> Int
valorAbsolutoG n
  | n >= 0      = n
  | otherwise   = (-n)

signo :: Int -> Int
signo n
  | n < 0      = (-1)
  | n == 0     = 0
  | otherwise  = 1
```

- otherwise** es una constante con valor True.

## Equiparación de patrones

- Se pueden utilizar varias ecuaciones.
- Haskell utiliza la primera que “encaja” con el argumento.
- Para valores específicos de los parámetros (no condiciones).
- Deben abarcar todo el dominio (si no, error `non-exhaustive patterns`)

```
and' :: Bool -> Bool -> Bool
and' True True = True
and' False True = False
and' True False = False
and' False False = False
```

Otra forma con menos patrones:

```
and'' :: Bool -> Bool -> Bool
and'' True y = y
and'' False y = False
```

## Variables anónimas

Otra forma:

```
and''' :: Bool -> Bool -> Bool
and''' True True = True
and''' x y       = False
```

Si  $x$  e  $y$  no se utilizan, no merece la pena darles nombre:

```
and'''' :: Bool -> Bool -> Bool
and'''' True True = True
and'''' _ _       = False
```

¿Y si cambiamos el orden de los patrones de este último ejemplo?

## Variables anónimas

¿Y si cambiamos el orden de los patrones de este último ejemplo?

```
and'''' :: Bool -> Bool -> Bool
and'''' _ _      = False
and'''' True True = True
```

warning: [-Woverlapping-patterns]

Pattern match is redundant

In an equation for 'and''''': and'''' True True = ...

```
*Main> and'''' True True
False
```



## Patrones para listas

```
suma :: Num a => [a] -> a
suma []      = 0      -- patrón lista vacía
suma [x]     = x      -- patrón lista con solo un elemento
suma [x,y]   = x + y  -- patrón lista con solo dos elementos
suma (x:xs)  = x + suma xs -- patrón lista con al menos un elemento
suma (x:y:xs) = x + y + suma xs -- patrón con al menos dos elementos
suma (x:y:z:xs) = x+y+z+suma xs -- patrón con al menos tres elementos
```

Pregunta: ¿Qué patrones son necesarios para la suma?

Equivale a:

```
suma :: Num a => [a] -> a
suma xs
  | null xs = 0
  | otherwise = (head xs) + suma (tail xs)
```

## Patrones para tuplas

Redefinición de la función `fst` (solo para pares):

```
fst' :: (a, b) -> a
fst' (x, _) = x
```

Redefinición de la función `snd` (solo para pares):

```
snd' :: (a, b) -> b
snd' (_, y) = y
```

Ejemplo de función para tuplas de 4 elementos:

```
tercero :: (a, b, c, d) -> c
tercero (_, _, z, _) = z
```

Otro ejemplo de función para tuplas de 4 elementos:

```
actualiza :: Num t => (t, t, [Char], [Char]) -> (t, t, [Char], [Char])
actualiza (x, y, xs, ys) = (x+y, x-y, xs++ys, ys++xs)
```

# Patrones para listas y tuplas

```
segundos :: [(a, b)] -> [b]
segundos []           = []
segundos ((_, x):ys) = x:(segundos ys)
```

```
*Main> segundos [(1, True), (2, False), (3, True)]
[True,False,True]
*Main> segundos [("avión", 'a'), ("barco", 'b')]
"ab"
```

Más información sobre patrones en [esta presentación](#), en las dispositivas 6 a 14.

## Lanzamiento de errores

Para lanzar un error que sea más útil que `non-exhaustive patterns` con la función `error`. Por ejemplo:

```
head' :: [a] -> a
head' [] = error "head: lista no puede estar vacía"
head' (x:_) = x
```

```
*Main> head' [24,25,26]
24
*Main> head' []
*** Exception: head: lista no puede estar vacía
```

## Instrucciones case-of

Se basa en la coincidencia de patrones en la parte derecha de la ecuación, posiblemente haciendo uso de guardas:

```
siguiente' n =  
  case n of  
    1 -> -1  
    x  
      | even x -> div x 2  
      | otherwise -> x*3+1
```

Más información en [este enlace](#).

# Funciones binarias y operadores

- Funciones **binarias** (con solo dos argumentos) son las únicas infijas, el resto son prefijas
  - **Funciones:** nombre alfanumérico

```
mas :: Int -> Int -> Int
mas x y = x + y
```

- **Operadores:** nombre simbólico

```
(*&)amp; :: Bool -> Bool -> Bool
(*&)amp; True True = True
(*&)amp; _ _ = False
infixr 1 *&
```

## Notación prefija e infija

Uso de funciones binarias y operadores con notación prefija:

```
*Main> (*&) False True  
False  
*Main> mas 6 19  
25
```

Uso de funciones binarias y operadores con notación infija:

```
*Main> True *& True  
True  
*Main> 4 `mas` 5  
25
```

# Precedencia

- Precedencia de algunos operadores
  - 9: .
  - 8: \*\*
  - 7: \*, /, ``div``, ``mod``
  - 6: +, -
  - 5: ++, :
  - 4: ==, /=, <, <=, >, >=
  - 3: &&
  - 2: ||
  - 1: >>, >>=
- Observación: ``div`` y ``mod`` tienen precedencia 7, mientras que `div` y `mod` tienen precedencia 9 (como todas las funciones).



# Precedencia

- Los operadores tienen asignado un orden de precedencia (entre 1 y 9) y de agrupación (l o r). Por defecto `infixl 9`.

`1 `más` 4 * 4  $\equiv$  (1 `más` 4) * 4`

`1 / 3 / 4  $\equiv$  (1 / 3) / 4`

`False *& True || True  $\equiv$  False *& (True || True)`

```
Prelude> :i ||  
(||) :: Bool -> Bool -> Bool          -- Defined in 'GHC.Classes'  
infixr 2 ||
```

- Las funciones tienen siempre mayor precedencia que los operadores

`div 8 2 * 2  $\equiv$  (div 8 2) * 2`

## Agrupación (asociatividad)

- La agrupación o asociatividad de los operadores se definen con izquierda (infixl) o derecha (infixr).

```
(~+) :: Int -> Int -> Int  
a ~+ b = (a + b) 'div' 2
```

Algunas combinaciones:

```
infixr 8 ~+
```

$4 \sim+ 2 \sim+ 14 * 10 \equiv (4 \sim+ (2 \sim+ 14)) * 10 = 60$

```
infixl 8 ~+
```

$4 \sim+ 2 \sim+ 14 * 10 \equiv ((4 \sim+ 2) \sim+ 14) * 10 = 80$

```
infixl 6 ~+
```

$4 \sim+ 2 \sim+ 14 * 10 \equiv ((4 \sim+ 2) \sim+ (14 * 10)) = 71$

# Comprobación de propiedades

- Propiedad: El doble de  $x$  más  $y$  es el doble de  $x$  más el doble de  $y$
- Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble  
+++ OK, passed 100 tests.
```

- Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero:

```
import Test.QuickCheck
```

# Refutación de propiedades

- Propiedad: El producto de dos números cualesquiera es distinto de su suma.
- Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```

- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma  
*** Failed! Falsifiable (after 1 test):  
0  
0
```

# Refutación de propiedades

- Refinamiento: El producto de dos números no nulos cualesquiera es distinto de su suma.
- Expresión de la propiedad:

```
prop_prod_suma' x y = x /= 0 && y /= 0 ==> x*y /= x+y
```

- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'  
*** Failed! Falsifiable (after 5 tests):  
2  
2
```

# Bibliografía I



R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.

Capítulo 1: Conceptos fundamentales



G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.

Chapter 4: Defining functions



B. O'Sullivan, D. Stewart y J. Goerzen. *Real World Haskell*. O'Reilly, 2008.

Chapter 2: Types and Functions



B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004..

Capítulo 2: Introducción a Haskell



S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.

Chapter 3: Basic types and definitions