

## Guardas y Tipos de datos

Taller de Álgebra I

2do Cuatrimestre 2017

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1  
    | otherwise = 0
```

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1  
    | otherwise = 0
```

¿Qué pasa si invertimos las guardas? **¿Por qué?**

## Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1  
    | otherwise = 0
```

¿Qué pasa si invertimos las guardas? **¿Por qué?**

**Presten atención al orden de las guardas.** ¡Cuando las condiciones se solapan, el orden de las guardas cambia el comportamiento de la función!

La función Signo:

$$\textit{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

La función Signo:

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

### Ejercicios

- Implementar la función `signo`.



La función Signo:

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

### Ejercicios

- ▶ Implementar la función `signo`.
- ▶ Implementar la función `absoluto` que calcula el valor absoluto de un número.

La función Signo:

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

### Ejercicios

- ▶ Implementar la función `signo`.
- ▶ Implementar la función `absoluto` que calcula el valor absoluto de un número. ¿Está bueno repetir? ¿Conviene reutilizar?

La función Signo:

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

### Ejercicios

- ▶ Implementar la función `signo`.
- ▶ Implementar la función `absoluto` que calcula el valor absoluto de un número. ¿Está bueno repetir? ¿Conviene reutilizar?
- ▶ Implementar la función `maximo` que devuelve el máximo entre 2 números.
- ▶ Implementar la función `maximo3` que devuelve el máximo entre 3 números.

## Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos

- 1 `Integer` =  $(\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.

## Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos

- 1 Integer =  $(\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
- 2 Float =  $(\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que “representa” a los racionales, con la aritmética de **punto flotante**.

# Tipos de datos

## Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos

- 1 Integer =  $(\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
- 2 Float =  $(\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que “representa” a los racionales, con la aritmética de **punto flotante**.
- 3 Bool =  $(\{\text{True}, \text{False}\}, \{\&\&, ||, \text{not}\})$  representa a los valores lógicos.

# Tipos de datos

## Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos

- 1 Integer =  $(\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
- 2 Float =  $(\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que “representa” a los racionales, con la aritmética de **punto flotante**.
- 3 Bool =  $(\{\text{True}, \text{False}\}, \{\&\&, ||, \text{not}\})$  representa a los valores lógicos.

Dado un valor de un tipo de datos, solamente se pueden aplicar a ese valor las operaciones definidas para ese tipo de datos.



# Tipos de datos

En Haskell los tipos se notan con `::`

Por ejemplo, en GHCi podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

# Tipos de datos

En Haskell los tipos se notan con `::`

Por ejemplo, en GHCi podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

A las expresiones también les corresponde un tipo de dato.

```
Prelude> :t 3 < 1
3 < 1 :: Bool
```

## Tipos de datos

En Haskell los tipos se notan con ::

Por ejemplo, en GHCi podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

A las expresiones también les corresponde un tipo de dato.

```
Prelude> :t 3 < 1
3 < 1 :: Bool
```

# Tipar vs Evaluar

Dada una expresión, se puede determinar su tipo **sin saber su valor**.

```
Prelude> :t 4 ^ 100000000000000000000000000000000 == 8  
4 ^ 100000000000000000000000000000000 == 8 :: Bool
```

¿Qué tipo tiene la expresión?

f True

¿Qué tipo tiene la expresión?

f `True`

Depende de f.

¿Qué tipo tiene la expresión?

f True

Depende de f. Por ejemplo:

```
f :: Bool -> Bool  
f x = not x
```

¿Qué tipo tiene la expresión?

```
f True
```

Depende de f. Por ejemplo:

```
f :: Bool -> Bool  
f x = not x
```

```
f :: Bool -> Float  
f x = pi
```

# Tipos de datos

¿Qué tipo tiene la expresión?

```
f True
```

Depende de f. Por ejemplo:

```
f :: Bool -> Bool  
f x = not x
```

```
f :: Bool -> Float  
f x = pi
```

Aplicación de funciones

```
funcion3 :: Integer -> Integer -> Bool -> Bool  
funcion3 x y b = b || (x > y)
```

```
Prelude>:t funcion3 10 20 True  
funcion3 10 20 True :: Bool
```



### Tipar las siguientes **funciones**

```
doble :: ??  
doble x = x + x  
  
cuadruple :: ??  
cuadruple x = doble (doble x)
```

## Tipos de datos: Ejercicios

### Tipar las siguientes funciones

```
doblo :: ??  
doblo x = x + x  
  
cuadruple :: ??  
cuadruple x = doblo (doblo x)
```

### Tipar las siguientes expresiones

- ▶ doblo 10
- ▶ dist (dist pi 0 pi 1) (doblo 0) (doblo 2) (3/4)  
Sabiendo que dist :: Float -> Float -> Float -> Float -> Float
- ▶ doblo True

# Tipos de datos: Ejercicios

## Tipar las siguientes funciones

```
doble :: ??  
doble x = x + x  
  
cuadruple :: ??  
cuadruple x = doble (doble x)
```

## Tipar las siguientes expresiones

- ▶ `doble 10`
- ▶ `dist (dist pi 0 pi 1) (doble 0) (doble 2) (3/4)`  
Sabido que `dist :: Float -> Float -> Float -> Float -> Float`
- ▶ `doble True`

## Implementar y tipar las siguientes funciones

- ▶ `esPar`: dado un valor determina si es par o no.
- ▶ `esMultiploDe`: dados dos números naturales, determina si el primero es múltiplo del segundo.

- ▶ Es importante observar la **signatura** de las funciones en las definiciones anteriores.
- ▶ Especificamos explícitamente el tipo de datos del dominio y el codominio de las funciones que definimos.

- ▶ Es importante observar la **signatura** de las funciones en las definiciones anteriores.
- ▶ Especificamos explícitamente el tipo de datos del dominio y el codominio de las funciones que definimos.
  - 1 No es estrictamente necesario especificarlo, dado que el mecanismo de **inferencia de tipos** de Haskell puede deducir la signatura más general para cada función.
  - 2 Sin embargo, es buena idea dar explícitamente la signatura de las funciones (¿por qué?).

## Variables de tipo

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Por ejemplo:

```
identidad :: a -> a  
identidad x = x
```

Notar que `a` va en minúscula y denota una **variable de tipo**.

## Variables de tipo

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Por ejemplo:

```
identidad :: a -> a
identidad x = x
```

Notar que `a` va en minúscula y denota una **variable de tipo**.

En Haskell esa función ya existe y se llama `id`:

```
:t id
id :: a -> a
```

Esta función vale para cualquier tipo de datos.

## Variables de tipo

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Por ejemplo:

```
identidad :: a -> a
identidad x = x
```

Notar que `a` va en minúscula y denota una **variable de tipo**.

En Haskell esa función ya existe y se llama `id`:

```
:t id
id :: a -> a
```

Esta función vale para cualquier tipo de datos.

¿Qué pasa con?

- ▶ `id (1 > 3)`
- ▶ `id (sqrt 2)`
- ▶ `:t id (1 > 3)`
- ▶ `:t id (sqrt 2)`



¿La función `triple x = x * 3` admite cualquier tipo de datos?

¿Qué pasa con...? ¿funcionan?

- ▶ `triple 2`
- ▶ `triple 2.5`
- ▶ `triple True`

¿La función `triple x = x * 3` admite cualquier tipo de datos?

¿Qué pasa con...? ¿funcionan?

- ▶ `triple 2`
- ▶ `triple 2.5`
- ▶ `triple True`

```
:t triple  
triple :: Num a => a -> a
```

¿Qué significa `Num a => ...` ?

¿La función `triple x = x * 3` admite cualquier tipo de datos?

¿Qué pasa con...? ¿funcionan?

- ▶ `triple 2`
- ▶ `triple 2.5`
- ▶ `triple True`

```
:t triple  
triple :: Num a => a -> a
```

¿Qué significa `Num a => ...` ?

Lo que aparece antes del símbolo `=>` es la condición que debe cumplir la variable de tipo `a`. La función `triple` solo admite tipos de datos numéricos.

## Clase de tipo

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

## Algunas clases:

- 1 `Num := ({ Int, Integer, Float, Double, ..., }, { (+), (*), abs, ... })`
- 2 `Integral := ({ Int, Integer, ..., }, { mod, div, ... })`
- 3 `Fractional := ({ Float, Double, ..., }, { (/), ... })`
- 4 `Floating := ({ Float, Double, ..., }, { sqrt, sin, cos, tan, ... })`
- 5 `Ord := { Bool, Int, Integer, Float, Double, ... }, { (<=), compare }`
- 6 `Eq := ({ Bool, Int, Integer, Float, Double, ... }, { (==), (/=) }`

## Notación infija

- ▶ Hay muchas funciones predeterminadas por haskell.. entre ellas se encuentra el +.
- ▶ Pero.. si es una función, ¿no debería escribirse + 2 3?

## Notación infija

- ▶ Hay muchas funciones predeterminadas por haskell.. entre ellas se encuentra el +.
- ▶ Pero.. si es una función, ¿no debería escribirse + 2 3?
- ▶ No. El nombre real de la función es (+); prueben (+) 2 3
- ▶ Haskell permite definir funciones con símbolos entre paréntesis, que después pueden ser utilizados de manera **infija** sin los paréntesis
- ▶ Ejemplos: (+), (-), (==), (>), (<), (>=), (<=), (^), (\*\*), (\*), etc.

## Notación infija

- ▶ Hay muchas funciones predeterminadas por haskell.. entre ellas se encuentra el +.
- ▶ Pero.. si es una función, ¿no debería escribirse + 2 3?
- ▶ No. El nombre real de la función es (+); prueben (+) 2 3
- ▶ Haskell permite definir funciones con símbolos entre paréntesis, que después pueden ser utilizados de manera **infija** sin los paréntesis
- ▶ Ejemplos: (+), (-), (==), (>), (<), (>=), (<=), (^), (\*\*), (\*), etc.

## ¿Cuál es la signatura de...?

- ▶ (>=)
- ▶ (==)

## Notación infija

- ▶ Hay muchas funciones predeterminadas por haskell.. entre ellas se encuentra el +.
- ▶ Pero.. si es una función, ¿no debería escribirse + 2 3?
- ▶ No. El nombre real de la función es (+); prueben (+) 2 3
- ▶ Haskell permite definir funciones con símbolos entre paréntesis, que después pueden ser utilizados de manera **infija** sin los paréntesis
- ▶ Ejemplos: (+), (-), (==), (>), (<), (>=), (<=), (^), (\*\*), (\*), etc.

## ¿Cuál es la signatura de...?

- ▶ (>=)
- ▶ (==)

```
:t (>=)
(>=) :: Ord a => a -> a -> Bool
```

```
:t (==)
(==) :: Eq a => a -> a -> Bool
```



## Nueva familia de tipos: Tuplas

- ▶ Dados dos tipos de datos A y B, podemos crear el tipo de datos (A, B) (“tupla de A y B”) que representa **pares ordenados** de elementos, donde el primero es de tipo A y el segundo es de tipo B.

## Nueva familia de tipos: Tuplas

- ▶ Dados dos tipos de datos A y B, podemos crear el tipo de datos (A, B) (“tupla de A y B”) que representa **pares ordenados** de elementos, donde el primero es de tipo A y el segundo es de tipo B.
- ▶ Algunas funciones:
  - ▶ `fst :: (a, b) -> a`
  - ▶ Ejemplo de uso: `fst (1 + 4, 2) ~> 5`
  - ▶ `snd :: (a, b) -> b`
  - ▶ Ejemplo de uso: `snd (1, (2, 3)) ~> (2, 3)`

## Nueva familia de tipos: Tuplas

- ▶ Dados dos tipos de datos A y B, podemos crear el tipo de datos (A, B) (“tupla de A y B”) que representa **pares ordenados** de elementos, donde el primero es de tipo A y el segundo es de tipo B.
- ▶ Algunas funciones:
  - ▶ `fst :: (a, b) -> a`
  - ▶ Ejemplo de uso: `fst (1 + 4, 2) ~> 5`
  - ▶ `snd :: (a, b) -> b`
  - ▶ Ejemplo de uso: `snd (1, (2, 3)) ~> (2, 3)`
- ▶ Ahora podemos definir la norma vectorial un poco más claramente:

```
normaVectorial :: (Float, Float) -> Float
normaVectorial p = sqrt ((fst p) ^ 2 + (snd p) ^ 2)
```

### Nota:

- ▶ Hay tuplas de distintos tamaños: `(True, 1, 4.0), (0, pi, False, pi)`.

## Nueva familia de tipos: Tuplas

- ▶ Dados dos tipos de datos A y B, podemos crear el tipo de datos (A, B) (“tupla de A y B”) que representa **pares ordenados** de elementos, donde el primero es de tipo A y el segundo es de tipo B.
- ▶ Algunas funciones:
  - ▶ `fst :: (a, b) -> a`
  - ▶ Ejemplo de uso: `fst (1 + 4, 2) ~> 5`
  - ▶ `snd :: (a, b) -> b`
  - ▶ Ejemplo de uso: `snd (1, (2, 3)) ~> (2, 3)`
- ▶ Ahora podemos definir la norma vectorial un poco más claramente:

```
normaVectorial :: (Float, Float) -> Float
normaVectorial p = sqrt ((fst p) ^ 2 + (snd p) ^ 2)
```

### Nota:

- ▶ Hay tuplas de distintos tamaños: `(True, 1, 4.0), (0, pi, False, pi)`.

## Ejercicios

- ▶ Implementar las siguientes funciones
  - ▶ `crearPar :: a -> b -> (a, b)` que crea un par a partir de sus dos componentes.
  - ▶ `invertir :: (a, b) -> (b, a)` que invierte el par pasado como parámetro
  - ▶ `distanciaPuntos :: (Float, Float) -> (Float, Float) -> Float`.

## Primero en papel y lápiz

- 1 Implementar las siguientes funciones del Ejercicio 31 Práctica 1 (reemplazamos  $\mathbb{N}$  por  $\mathbb{Z}$ ), usando tipo Integer para los números enteros y tipo Float para los números reales:

- ▶ 31.iv)  $f1 : \mathbb{R} \rightarrow \mathbb{R}^2$ ,  $f1(x) = (e^x, 1 - e^x)$
- ▶ 31.v)  $f2 : \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $f2(n) = \begin{cases} \frac{n}{2} & \text{si } n \text{ es par} \\ n + 1 & \text{si } n \text{ es impar} \end{cases}$

y calcular  $f1(3)$ ,  $f1(\sqrt{2})$ ,  $f2(5)$ ,  $f2(4)$ ,  $f2(-10)$ .

¿Qué sucede si queremos calcular  $f2(\sqrt{2})$ ?

- 2 Implementar las funciones  $f$  y  $g$  del Ejercicio 32.i) Práctica 1:

$$f : \mathbb{Z} \rightarrow \mathbb{Z}, \quad f(n) = \begin{cases} \frac{n^2}{2} & \text{si } n \text{ es divisible por 6} \\ 3n + 1 & \text{en los otros casos} \end{cases}$$

$$g : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}, \quad g(n, m) = n(m + 1)$$

y calcular  $(f \circ g)(3, 4)$ ,  $(f \circ g)(2, 5)$ .

Implementar una función  $h = (f \circ g)$  y calcular  $h(3, 2)$ . ¿Cuál es la signatura de  $h$ ?