



Tema 2. Tipos de datos simples

- En Haskell toda *expresión* debe tener un *tipo* asociado
 - una expresión “mal tipada” no es correcta.
- *Tipado fuerte y estático* (comprobación en compilación):
 - *las definiciones del script*: antes ser cargadas en memoria.
 - *la expresión a evaluar*: antes de ser evaluada.
- El programador puede *declarar* el tipo de una función:

`doble :: Int → Int`

pero el sistema es capaz de *inferirlo* a partir de su definición:

`doble x = x + x`



Tipos básicos predefinidos

- **Char:** 'a', 'B', '+', '3', '\n', '\'', ' ', ...
 - **String ó [Char]:** "hola", "3+4", "Hola\nAdios",...
 - **Bool:** True, False con los operadores lógicos:
 && (conjunción), || (disyunción), not (negación)
 - **Int / Integer:** 3, 0, 5, -4... (rango acotado / arbitrario)
 - **Float / Double:** 4.8, 5.3E2, ... con operadores aritméticos:
 +, -, *, /, `mod`, `div`, ^, abs, ...
- Y todos ellos con los operadores de igualdad y orden:
- ==, /=, >, >=, <, <=, max, min



Ejemplos de funciones

- `isLower, isUpper, isAlpha, isDigit :: Char -> Bool`

`isLower c = c >= 'a' && c <= 'z'`

`isUpper c = c >= 'A' && c <= 'Z'`

`isAlpha c = isLower c || isUpper c`

`isDigit c = c >= '0' && c <= '9'`

- Se definen en base a un orden predeterminado en Char

`Prelude> fromEnum 'a'`

97

`Prelude> fromEnum 'A'`

65

`Prelude> 'A' < 'a'`

True

`Prelude> toEnum 97 :: Char`

'a'



Definición de funciones (1)

- La *definición de una función* consiste en una o más ecuaciones de la forma:

$$\text{f } \underbrace{\text{p1 p2 ... pn}}_{\text{< 0 ó más parámetros >}} = \text{<parte derecha>}$$

<nombre de la función>

- cada **pi** es un *patrón* (en concreto una *variable*)
- la **<parte derecha>** es una expresión que puede usar:
 - condicionales / por casos (guardas)
 - definiciones locales (“where”)
 - puede haber recursión (referencias a **f**)



Definición de funciones (2)

- Definición simple

doble $x = x + x$

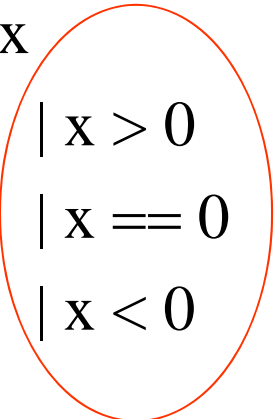
sumdo $x \ y = x + \text{doble } y$

- Definición condicional

absoluto $x = \text{if } x \geq 0 \text{ then } x \text{ else } -x$

- Definición por casos

signo x



$x > 0$	$= 1$
$x == 0$	$= 0$
$x < 0$	$= -1$

guardas



Definición de funciones (3)

- Definiciones recursivas

factorial x

| $x < 0$ = **error** “dato negativo”

| $x == 0$ = 1

| otherwise = $x * \text{factorial } (x-1)$

- guardas evaluadas en orden
- guarda *otherwise* para el último caso
- **error** => función parcial explícita (**factorial** :: $\text{Int} \rightarrow \text{Int}$)
 efecto: causa terminación y muestra mensaje en pantalla



Definición de funciones (4)

- Definiciones locales

$$g \ x \ y = (a+b) * (a-b)$$

where

$$a = x + y$$

$$b = x * y$$

en lugar de:

$$g \ x \ y = ((x + y) + (x * y)) * ((x + y) - (x * y))$$

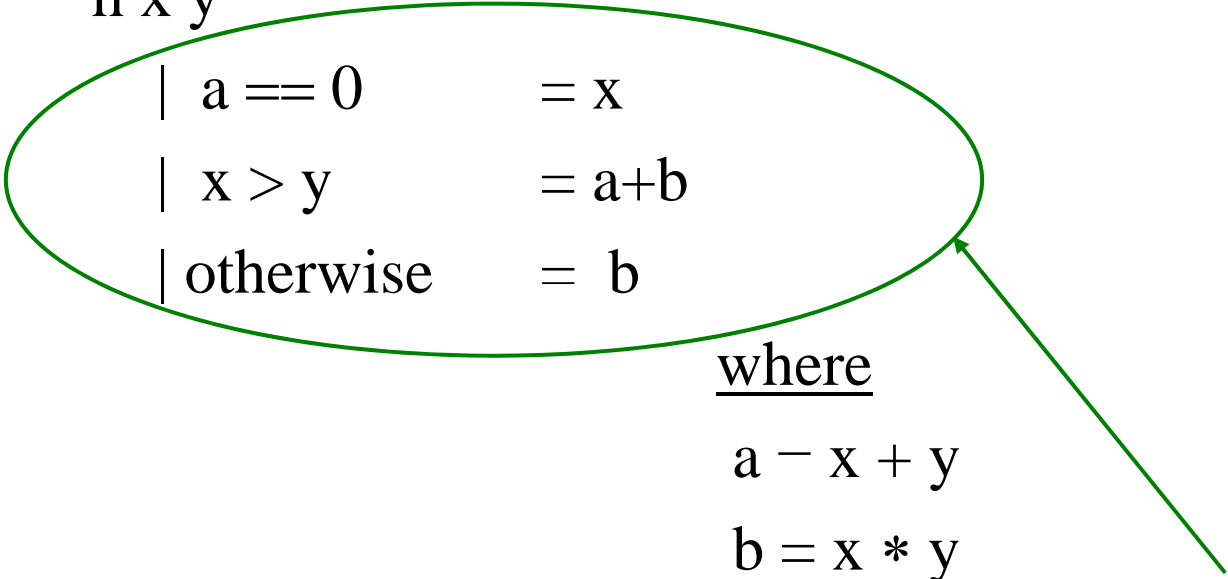
- utilidad: menos repetición y más legibilidad
- eficiencia: evaluación única



Definición de funciones (5)

- Ejemplo de definición con casos + where

h x y



a == 0	= x
x > y	= a+b
otherwise	= b

where

a = x + y

b = x * y

- Definiciones del “where” para toda la parte derecha
- ¡ojo con el layout!



Currificación (1)

- Currificación: El tipo de una función **f** con n argumentos es

$$f :: T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T$$

Ej: sumdo, g :: Int → Int → Int (definidas antes)

- Parámetros sin paréntesis en la definición de **f**

Ej: g x y =
 sumdo x y =

- Argumentos se escriben uno tras otro en la evaluación de **f**

Ej: g 3 4 + sumdo 7 2



Curricación (2)

- Aplicaciones parciales de $f :: T_1 \rightarrow T_2 \dots \rightarrow T_n \rightarrow T$
son a su vez funciones:

$$f \ x_1 \quad \quad \quad :: T_2 \rightarrow T_3 \rightarrow \dots \rightarrow T_n \rightarrow T$$

$$f \ x_1 \ x_2 \quad \quad \quad :: T_3 \rightarrow \dots \rightarrow T_n \rightarrow T$$

.....

$$f \ x_1 \ x_2 \ \dots \ x_{n-1} \quad \quad \quad :: T_n \rightarrow T$$

$$f \ x_1 \ x_2 \ \dots \ x_{n-1} \ x_n \quad \quad \quad :: T$$

(x_i es cualquier elemento del tipo T_i)



Curricación (3)

Ejemplo:

```
sumdo :: Int -> Int -> Int  
sumdo x y = x + doble y
```

genera (automáticamente) funciones como

```
sumdo 3 :: Int -> Int  
      y -> (3 + doble y)
```

que puede ser usada en otras definiciones como

```
func = sumdo 3  
lis = map (sumdo 3) [4,5,2]
```

? func 2
? lis



Operadores. Secciones

- Todas las funciones predefinidas de Haskell (incluidos los operadores) son funciones currificadas

Ej: $(+), \text{mod} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ donde

$(+) \ 3 \ 4$ indica lo mismo que $3 + 4$

$\text{mod} \ 6 \ 4$ indica lo mismo que $6 \ \text{mod} \ 4$

- *Secciones*: nuevas funciones obtenidas añadiendo alguno de sus dos argumentos al operador binario

Ej: $(+3), (/2), (2/), (<0), (0<), \dots$

- Todos los operadores tienen una *prioridad* (de 1 a 9) y una *asociatividad* (a izqda ó dcha).

Ej: Preguntar al sistema mediante `:i (+)`