Introducción a Haskell

El lenguaje Haskell

- Haskell es un lenguaje funcional *puro*, *no estricto* y *fuertemente tipificado*.
  - ✓ Puro = *transparencia referencial*:
  - ✓ No estricto = usa un orden no aplicativo (Evaluación perezosa).
  - ✓ *tipificación fuerte* = los elementos del lenguaje utilizables están clasificados en distintas categorías o tipos.
- Un programa consiste en definiciones de funciones.
  - Declararla: indicar el tipo
  - Definirla: dar el método de computo.
- Los principales tipos de datos básicos predefinidos en Haskell son: $Char$, Int, $Integer$, $Float$, $Double$ y $Bool$.

```
-- Un ejemplo de fichero Haskell

-- Calcula el siguiente entero al argumento

sucesor    :: Integer -> Integer
sucesor x  = x + 1

-- Calcula la suma de los cuadrados de sus dos argumentos

sumaCuadrados      :: Integer -> Integer -> Integer
sumaCuadrados x y  = x * x + y * y
```

Tipos simples predefinidos

El tipo *Bool*

Constructores: *True* y *False*

Funciones y operadores

$(\&\&) :: Bool \to Bool \to Bool.$

$(\|\|) :: Bool \to Bool \to Bool.$

$not :: Bool \to Bool.$

$otherwise :: Bool.$

El tipo *Int*

Números enteros de precisión limitada que cubren al menos el rango $[-2^{29}, 2^{29} - 1]$.

```
Prelude> minBound :: Int
-2147483648 :: Int
Prelude> maxBound :: Int
2147483647 :: Int
```

Funciones y operadores

$(+), (-), (*) :: Int \to Int \to Int.$

$(\uparrow) :: Int \to Int \to Int.$

$div, mod :: Int \to Int \to Int.$

$abs :: Int \to Int.$

$signum :: Int \to Int.$

$negate :: Int \to Int.$

$even, odd :: Int \to Int.$

- Podemos definir una función que a partir de la relación

$$máximo\ (x, y) = \frac{(x + y) + |x - y|}{2}$$

calcule el máximo de sus argumentos

```
máximo     :: Int → Int → Int
máximo x y = ((x + y) + abs(x − y)) `div` 2
```

El tipo *Integer*

- Los valores de este tipo son números enteros de precisión ilimitada.
  - ✓ Mismas operaciones que para *Int*.

```
Prelude> 2 ↑ 100
1267650600228229401496703205376 :: Integer
```

El tipo *Float*

- Representan números reales.
  - En notación habitual: -5.3, 1.0 ó 1.
  - En notación científica: $1.5e7$ ó $1.5e-17$

Funciones y operadores

$(+), (*), (-), (/)$

$:: \ Float \rightarrow Float \rightarrow Float.$

$(\uparrow) :: \ Float \rightarrow Int \rightarrow Float.$

$(**) :: \ Float \rightarrow Float \rightarrow Float.$

$sin, \ asin, \ cos, \ acos, \ tan, \ atan$

$:: \ Float \rightarrow Float.$

$atan2 :: \ Float \rightarrow Float \rightarrow Float.$

$log, \ log10, \ exp :: \ Float \rightarrow Float.$

$sqrt :: \ Float \rightarrow Float.$

...

El tipo *Double*

- Números reales con mayor precisión que *Float* y las mismas operaciones.

El tipo *Char*

- Representan caracteres ($'a'$, $'1'$, $'?'$)
  - ✓ Algunos caracteres especiales se escriben precediéndolos del carácter \:

$'\backslash n'$, $'\backslash t'$, $'\backslash''$, $'\backslash"'$, $'\backslash\backslash'$

Funciones

$ord :: \ Char \rightarrow Int.$

$chr :: \ Int \rightarrow Char.$

$isUpper, \ isLower, \ isDigit, \ isAlpha :: \ Char \rightarrow Bool.$

$toUpper, \ toLower :: \ Char \rightarrow Char.$

Operadores de igualdad y orden

Para todos los tipos básicos comentados están definidos los siguientes *operadores binarios* que devuelven un valor booleano:

$(>)$	mayor que
$(\geq)$	mayor o igual que
$(<)$	menor que
$(\leq)$	menor o igual que
$(==)$	igual a
$(\neq)$	distinto de

El tipo de los dos argumentos para cualquier aplicación de los operadores anteriores debe ser el mismo:

```
Prelude> 10 ≤ 15
True :: Bool
Prelude> 'x' == 'y'
False :: Bool
Prelude> 'x' ≠ 'y'
True :: Bool
Prelude> 'b' > 'a'
True :: Bool
Prelude> False < True
True :: Bool
```

```
Prelude> (1 < 5) && (10 > 9)
True :: Bool
Prelude> True < 'a'

ERROR                  : Type error in application
*** Expression        : True < 'a'
*** Term              : True
*** Type              : Bool
*** Does not match    : Char
```

Constructores de tipo predefinidos

HASKELL define *tipos estructurados* que permiten representar colecciones de objetos.

Tuplas

Una *tupla* es un dato compuesto donde el tipo de cada componente puede ser distinto.

Tuplas

Si $v_1, v_2, \ldots, v_n$ son valores con tipo $t_1, t_2, \ldots, t_n$
entonces $(v_1, v_2, \ldots, v_n)$ es una tupla con tipo $(t_1, t_2, \ldots, t_n)$

```
PRELUDE> ()
() :: ()
PRELUDE> ('a', True)
('a', True) :: (Char, Bool)
PRELUDE> ('a', True, 1.5)
('a', True, 1.5) :: (Char, Bool, Double)
```

- Las tuplas son útiles cuando una función tiene que devolver más de un valor:

```
predSuc   :: Integer → (Integer, Integer)
predSuc x = (x − 1, x + 1)
```

Listas

Una *lista* es una colección de cero o más elementos todos del mismo tipo. Hay dos constructores (operadores que permiten construir valores) para listas:

- $[\,]$ Representa la lista vacía (una lista sin elementos).
- $(:)$ Permite añadir un elemento al principio de una lista.

Si el tipo de todos los elementos de una lista es t, entonces el tipo de la lista se escribe $[t]$:

Listas

Si $v_1, v_2, \ldots, v_n$ son valores con tipo t
entonces $v_1 : (v_2 : (\ldots (v_{n-1} : (v_n : [\,]))))$ es una lista con tipo $[t]$

Asociatividad derecha de $(:)$

$$x_1 : x_2 : \ldots x_{n-1} : x_n : [\,] \rightsquigarrow x_1 : (x_2 : (\ldots (x_{n-1} : (x_n : [\,]))))$$

Sintaxis para listas

$$[x_1, x_2, \ldots x_{n-1}, x_n] \rightsquigarrow x_1 : (x_2 : (\ldots (x_{n-1} : (x_n : [\,]))))$$

```
Prelude> 1 : (2 : (3 : []))
[1,2,3] :: [Integer]
Prelude> 1 : 2 : 3 : []
[1,2,3] :: [Integer]
Prelude> [1,2,3]
[1,2,3] :: [Integer]
```

Cadenas de caracteres

Una *cadena de caracteres* es una secuencia de cero o más caracteres.
$['\mathrm{h}', '\mathrm{o}', '\mathrm{l}', '\mathrm{a}']$ tiene el tipo $[Char]$ (o también $String$).

Cadenas de caracteres

$$"x_1\ x_2\ \ldots\ x_{n-1}\ x_n" \rightsquigarrow ['\mathrm{x}_1', '\mathrm{x}_2', \ldots '\mathrm{x}_{n-1}', '\mathrm{x}_n']$$

```
Prelude> 'U' : 'n' : ' ' : 'C' : 'o' : 'c' : 'h' : 'e' : []
"Un Coche" :: [Char]
Prelude> ['U','n',' ','C','o','c','h','e']
"Un Coche" :: [Char]
Prelude> "Un Coche"
"Un Coche" :: String
```

El constructor de tipo $(\rightarrow)$

Tipos Funcionales

Si $t_1, t_2, \ldots, t_n, t_r$ son tipos válidos
entonces $t_1 \rightarrow t_2 \rightarrow \ldots t_n \rightarrow t_r$ es el tipo
de una función con n argumentos

- El argumento o resultado de una función puede ser otra función, dando lugar a lo que se denomina *funciones de orden superior*

$$componer :: (Integer \to Integer) \to (Integer \to Integer) \to Integer \to Integer$$
$$componer\ g\ f\ x = g\ (f\ x)$$

Los paréntesis no pueden ser eliminados en el tipo de $componer$

$$\text{MAIN}> componer\ inc\ inc\ 10$$
$$12 :: Integer$$

$$componer\ inc\ inc\ 10$$
$$\Longrightarrow \quad \text{! sustituyendo en la definición de } componer$$
$$inc\ (inc\ 10)$$
$$\Longrightarrow \quad \text{! por definición de } inc$$
$$(inc\ 10) + 1$$
$$\Longrightarrow \quad \text{! por definición de } inc$$
$$(10 + 1) + 1$$
$$\Longrightarrow \quad \text{! por definición de } (+)$$
$$11 + 1$$
$$\Longrightarrow \quad \text{! por definición de } (+)$$
$$12$$

Comentarios

Hay dos modos de incluir comentarios en un programa:

- Comentarios de una sola línea: comienzan por dos guiones consecutivos ($--$) y abarcan hasta el final de la línea actual:

$$
\begin{aligned}
&f \;\; :: \; Integer \; \rightarrow \; Integer \\
&f \; x \; = x \; + \; 1 \quad -- \text{Esto es un comentario}
\end{aligned}
$$

- Comentarios que abarcan más de una línea: Comienzan por los caracteres $\{-$ y acaban con $-\}$. Pueden abarcar varias líneas y anidarse:

```
{- Esto es un comentario
   de más de una línea -}
```

$$
\begin{aligned}
&g \;\; :: \; Integer \; \rightarrow \; Integer \\
&g \; x \; = x \; - \; 1
\end{aligned}
$$

Operadores

- Funciones con dos argumentos cuyo nombre es simbólico (o literal)
- Pueden ser invocados de forma *infija* (entre sus dos argumentos).
  - ✓ Ya hemos visto algunos de los operadores predefinidos como &&, ||, +, −, ∗, / y ↑.
- El programador puede definir sus propios operadores. Puede utilizar uno o más de:

  : ! # $ % & ∗ + . / < = > ? @ \ ↑ | −

- También se puede usar el símbolo ∼ (sólo con una aparición y al principio).
  - ✓ Los operadores que comienzan por el carácter dos puntos (:) tienen un significado especial: son *constructores de datos infijos* (usados para construir valores de un tipo).
- Algunos ejemplos de operadores válidos son:

  + ++ && || ≤ == ≠ . // $ predefinidos
  % @@ −+− \/ /\ <+> ?

- Operadores reservados

  :: → ⇒ : .. = @ \ | ← ∼

  A la hora de definir un operador podemos indicar su:
  - *prioridad* (0 a 9; 10 es la prioridad máxima reservada para la aplicación de funciones)
  - su *asociatividad*.

infix

infix | prioridad | | identificador operador | (define un operador no asociativo)

infixl | prioridad | | identificador operador | (define un operador asociativo a la izquierda)

infixr | prioridad | | identificador operador | (define un operador asociativo a la derecha)

✓ Para declarar el tipo de un operador hay que escribir el identificador de éste entre paréntesis.
✓ En la parte izquierda de la definición del cuerpo del operador se puede usar notación infija (el operador aparece entre sus dos argumentos).

```
infix 4 ~=

(~=)    :: Float -> Float -> Bool
x ~= y  = abs(x - y) < 0.0001
```

¿Cuál es el valor de la expresión $8 - 5 - 1$? Puede ser $((8 - 5) - 1)$ ó $(8 - (5 - 1))$. La asociatividad aclara el significado de la expresión en este caso.

Si $\otimes$ es un operador asociativo a la izquierda:

$$x_1 \otimes x_2 \otimes \cdots \otimes x_{n-1} \otimes x_n \rightsquigarrow (((x_1 \otimes x_2) \otimes \ldots\ x_{n-1}) \otimes x_n)$$

Si $\otimes$ es un operador asociativo a la derecha:

$$x_1 \otimes x_2 \otimes \cdots \otimes x_{n-1} \otimes x_n \rightsquigarrow (x_1 \otimes x_2(\otimes \ldots\ (x_{n-1} \otimes x_n)))$$

En HASKELL, todo operador toma por defecto prioridad 9 y asociatividad izquierda.

```
infixr 9  .
infixl 9  !!
infixr 8  ↑, ↑↑, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, −
infixr 5  :
infixr 5  ++
infix  4  ==, ≠, <, ≤
infix  4  ≥, >, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Operadores frente a funciones

Cualquier operador puede usarse tanto de modo prefijo como infijo

```
infix 4 ∼=

(∼=)      :: Float → Float → Bool
(∼=) x y  = abs(x − y) < 0.0001
```

```
PRELUDE> (+) 3 4
7 :: Integer
```

Una función de dos argumentos puede ser convertida en un operador si se escribe entre acentos franceses (el carácter `):

```
suma       :: Integer → Integer → Integer
x `suma` y = x + y
```

```
MAIN> suma 1 2
3 :: Integer
MAIN> 1 `suma` 2
3 :: Integer
```

También se puede dar una prioridad y asociatividad al uso infijo de cualquier función:

```
infixl 6 `suma`
```

Comparación de Patrones

- Un patrón es una expresión como argumento en una ecuación
- Es posible definir una función dando más de una ecuación para ésta.

✓ Al aplicar la función a un parámetro concreto la *comparación de patrones* determina la ecuación a utilizar.

Patrones constantes

Un *patrón constante* puede ser un número, un carácter o un constructor de dato.

$$\begin{aligned} f &:: Integer \rightarrow Bool \\ f\ 1 &= True \\ f\ 2 &= False \end{aligned}$$

- La definición de la conjunción y disyunción de valores lógicos usa patrones constantes ($True$ y $False$ son dos constructores de datos para el tipo $Bool$):

$$\begin{aligned} &\mathbf{infixr}\ 3\ \&\& \\ &(\&\&) :: Bool \rightarrow Bool \rightarrow Bool \\ &False\ \&\&\ x = False \\ &True\ \&\&\ x = x \end{aligned}$$

$$\begin{aligned} &\mathbf{infixr}\ 2\ \| \\ &(\|) :: Bool \rightarrow Bool \rightarrow Bool \\ &False\ \|\ x = x \\ &True\ \|\ x = True \end{aligned}$$

- Regla para la comparación de patrones
  - Se comprueban los patrones correspondientes a las distintas ecuaciones en el orden dado por el programa, hasta que se encuentre una que unifique.
  - Dentro de una misma ecuación se intentan unificar los patrones correspondientes a los argumentos de izquierda a derecha.
  - En cuanto un patrón falla para un argumento, se pasa a la siguiente ecuación.

Patrones para listas

Es posible utilizar patrones al definir funciones que trabajen con listas.

$[\,]$ sólo unifica con un argumento que sea una lista vacía.

$[x]$, $[x, y]$, etc. sólo unifican con listas de uno, dos, etc. argumentos.

$(x : xs)$ unifica con listas con al menos un elemento

$$
\begin{aligned}
&suma && :: [Integer] \to Integer\\
&suma\ [\,] && = 0\\
&suma\ (x : xs) && = x\ +\ suma\ xs
\end{aligned}
$$

$$
\begin{aligned}
&suma\ [1, 2, 3]\\
\rightsquigarrow\ &\text{! sintaxis de listas}\\
&suma\ (1 : (2 : (3 : [\,])))\\
\Longrightarrow\ &\text{! segunda ecuación de } suma\ \{x \leftarrowtail 1,\ xs \leftarrowtail 2 : (3 : [\,])\}\\
&\ldots\\
&6
\end{aligned}
$$

Patrones para tuplas

$$
\begin{aligned}
&primero2 && :: (Integer,\ Integer) \to Integer\\
&primero2\ (x,\ y) && = x
\end{aligned}
$$

$$
\begin{aligned}
&primero3 && :: (Integer,\ Integer,\ Integer) \to Integer\\
&primero3\ (x,\ y,\ z) && = x
\end{aligned}
$$

Los patrones pueden anidarse.

$$
\begin{aligned}
&sumaPares && :: [(Integer,\ Integer)] \to Integer\\
&sumaPares\ [\,] && = 0\\
&sumaPares\ ((x, y) : xs) && = x\ +\ y\ +\ sumaPares\ xs
\end{aligned}
$$

Patrones aritméticos

Es un patrón de la forma $(n + k)$, donde k es un valor constante natural.

$$
\begin{array}{ll}
factorial & :: Integer \rightarrow Integer \\
factorial\ 0 & = 1 \\
factorial\ (n+1) & = (n+1) * factorial\ n
\end{array}
$$

Patrones nombrados o seudónimos

Seudónimo o *patrón alias* para nombrar un patrón, y utilizar el seudónimo en vez del patrón en la parte derecha de la definición.

$$
\begin{array}{ll}
factorial'' & :: Integer \rightarrow Integer \\
factorial''\ 0 & = 1 \\
factorial''\ m@(n+1) & = m * factorial''\ n
\end{array}
$$

El patrón subrayado

Un *patrón subrayado* (_) unifica con cualquier argumento pero no establece ninguna ligadura.

$$
\begin{array}{ll}
longitud & :: [Integer] \rightarrow Integer \\
longitud\ [\,] & = 0 \\
longitud\ (\_ : xs) & = 1 + longitud\ xs
\end{array}
$$

Patrones y evaluación perezosa

La unificación determina qué ecuación es seleccionada para reducir una expresión a partir de la forma de los argumentos.

$$
\begin{array}{ll}
esVacía & :: [a] \rightarrow Bool \\
esVacía\ [\,] & = True \\
esVacía\ (\_ : \_) & = False
\end{array}
$$

Para poder reducir una expresión como $esVacía\ l$ es necesario saber si l es una lista vacía o no.

HASKELL evalúa un argumento hasta obtener un número de constructores suficiente para resolver el patrón, sin obtener necesariamente su forma normal.

$$
\begin{array}{ll}
 & esVacía\ infinita \\
\Longrightarrow & \text{! definición de } infinita \\
 & esVacía\ (1 : infinita) \\
\Longrightarrow & \text{! segunda ecuación de } esVacía \\
 & False
\end{array}
$$

Expresiones *case*

La sintaxis de esta construcción es:

```
case expr of
  patron_1 → resultado_1
  patron_2 → resultado_2
  ...
  patron_n → resultado_n
```

```
long    :: [Integer] → Integer
long ls = case ls of
            []     → 0
            _ : xs → 1 + long xs
```

La función *error*

Si intentamos evaluar una función parcial en un punto en el cual no está definida, se produce un error.

Con la función *error* podemos producir producir nuestro mensaje de error.

```
cabeza'         :: [Integer] → Integer
cabeza' []      = error "lista vacía"
cabeza' (x : _) = x
```

```
Main> cabeza' [1,2,3]
1 :: Integer
Main> cabeza' []
Program error : lista vacía
```

Funciones a trozos

$$absoluto :: \mathbb{Z} \to \mathbb{Z}$$
$$absoluto(x) = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{si } x < 0 \end{cases}$$

En HASKELL la definición anterior puede escribirse del siguiente modo:

```
absoluto   :: Integer → Integer
absoluto x
  | x ≥ 0   = x
  | x < 0   = -x
```

Es posible utilizar como guarda la palabra *otherwise*, que es equivalente al valor *True*.

```
signo          :: Integer → Integer
signo x
  | x > 0      = 1
  | x == 0     = 0
  | otherwise  = -1
```

Expresiones condicionales

Otro modo de escribir expresiones cuyo resultado dependa de cierta condición es utilizando expresiones condicionales.

```
if exprBool then exprSi else exprNo
```

```
maxEnt      :: Integer → Integer → Integer
maxEnt x y  = if x ≥ y then x else y
```

```
PRELUDE> if 5 > 2 then 10.0 else (10.0/0.0)
10.0 :: Double
PRELUDE> 2 * if 'a' < 'z' then 10 else 4
20 :: Integer
```

Definiciones locales

A menudo es conveniente dar un nombre a una subexpresión que se usa varias veces.

```
raíces :: Float → Float → Float → (Float, Float)

raíces a b c
  | b ↑ 2 − 4 ∗ a ∗ c ≥ 0 = ( (−b + sqrt(b ↑ 2 − 4 ∗ a ∗ c))/(2 ∗ a),
                              (−b − sqrt(b ↑ 2 − 4 ∗ a ∗ c))/(2 ∗ a) )
  | otherwise             = error "raíces complejas"
```

Podemos mejorar la legibilidad de la definición anterior.
En HASKELL podemos usar para este propósito la palabra **where**.

```
raíces          :: Float → Float → Float → (Float, Float)
raíces a b c
  | disc ≥ 0    = ((−b + raízDisc)/denom, (−b − raízDisc)/denom)
  | otherwise   = error "raíces complejas"
  where
    disc        =b ↑ 2 − 4 ∗ a ∗ c
    raízDisc    =sqrt disc
    denom       =2 ∗ a
```

Las definiciones locales **where** sólo pueden aparecer al final de una definición de función.
Para introducir definiciones locales en cualquier parte de una expresión se usa **let** e **in**.

```
PRELUDE> let f n =n ↑ 2 + 2 in f 100
10002 :: Integer
```

Expresiones lambda

HASKELL permite definir funciones anónimas mediante las *expresiones lambda* (también denominadas λ-expresiones).

$\lambda\, x \;\rightarrow\; x \;+\; 1$

```
PRELUDE> :t \x -> isUpper x
\x -> isUpper x :: Char -> Bool
PRELUDE> (\x -> isUpper x) 'A'
True :: Bool
```

También podemos definir funciones de más de un argumento con la notación lambda:

```
PRELUDE> :t \x y -> isUpper x && isUpper y
\x y -> isUpper x && isUpper y :: Char -> Char -> Bool
PRELUDE> (\x y -> isUpper x && isUpper y) 'A' 'z'
False :: Bool
```

Ámbitos y módulos

```
f    :: Integer → Integer
f x  = x * h 7
  where
    h z = x + 5 + z ↑ 2
(+++)   :: Integer → Integer → Integer
a +++ b = 2 * a + f b
```

HASKELL proporciona un conjunto de definiciones globales que pueden ser usadas por el programador sin necesidad de definirlas.

Estas definiciones aparecen agrupadas en *módulos de biblioteca*. Una biblioteca es un conjunto de definiciones relacionadas.

• Existe una biblioteca en la que se define el tipo *Rational* que representa números racionales cuyo nombre es *Ratio*.

La importación se consigue escribiendo la palabra **import** al inicio del programa.

```
import Ratio

sumaCubos       :: Rational → Rational → Rational
sumaCubos ra rb = ra ↑ 3 + rb ↑ 3
```

Existe un módulo de biblioteca especial denominado PRELUDE automáticamente importado por cualquier programa.