

# Programación funcional con Haskell

Juan Manuel Rabasedas



Matemática	Haskell
$f(x)$	$f\ x$
$f(x, x)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x\ \circ\ g\ y$

- En matemática la aplicación de funciones se denota usando paréntesis:  $f\ (a, b) + c\ d$
- En Haskell la aplicación de funciones se denota con un espacio:  $f\ a\ b + c * d$
- La aplicación tiene mayor precedencia que cualquier otro operador:  $f\ x + y = (f\ x) + y$
- La aplicación asocia a las izquierda:  $f\ a\ b + c * d = ((f\ a)\ b) + c * d$

- Las funciones y sus argumentos deben empezar con minúscula, y pueden ser seguidos por cero o más letras (mayúsculas o minúsculas), dígitos, guiones bajos, y apóstrofes.

*myFun fun1 arg\_2 x'*

- Las palabras reservadas son:  
**case class data default deriving do else if import in  
infix infixl infixr instance let module newtype of then type where**
- Por convención los nombres de las listas terminan en *s* para decir que contienen múltiples valores ejemplo *xs*.
- Para comentarios se utiliza `--` este es un comentario

`{-ponga aquí su comentario de más de una línea -}`

# The layout rule

En una serie de definiciones, cada definición debe empezar en la misma columna.

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

También se puede hacer en forma explicita metiendo una secuencia de definiciones entre llaves (`{ }`) y separandolas con (`;`):

```
a = b + c
  where
    { b = 1;
      c = 2 }
d = a * 2
```

Pero no la vamos a usar en este curso.

- Un tipo es una colección de valores relacionados.
  - Bool contiene los valores True y False.
  - Escribimos `True :: Bool` y `False :: Bool`.
  - `Bool → Bool` contiene todas las funciones que van de Bool en Bool
- En general, si la evaluación de una expresión  $e$  tiene tipo  $t$  escribimos

$$e :: t$$

*En Haskell toda expresión tiene que tener un tipo*

- El tipo de una expresión es calculado antes de su evaluación por el inferidor de tipo.
- Si el tipo de una expresión no se puede inferir la expresión no es válida.

# Tipos básicos

Algunos tipos básicos de vamos a considerar son:

- Bool booleanos
- Char caracteres
- Int enteros de precisión fija.
- Integer enteros de precisión arbitraria.
- Float números de punto flotante de precisión simple.

- Una lista es una secuencia de valores del mismo tipo
  - `[False, True, False] :: [Bool]`
  - `['a', 'b', 'c', 'd'] :: [Char]`
- En general, `[t]` es una lista con elementos de tipo `t`
- Donde `t`, puede ser cualquier tipo válido.
- No hay restricción con respecto a la longitud de las listas.
- El tipo de las listas no dice nada sobre su longitud.
- El tipo de elemento de una lista es irrestricto.  
`[['a'], ['b', 'c']] :: [[Char]]`

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - `(True, True) :: (Bool, Bool)`
  - `(True, 'a', 'b') :: (Bool, Char, Char)`
- En general,  $(t_1, t_2, \dots; t_n)$  es el tipo de una  $n$ -tupla cuyas componente  $i$  tiene tipo  $t_i$ , para  $i$  en  $1 \dots n$ .
- A diferencia de las listas, las tuplas tienen explicitado en su tipo la cantidad de elementos que almacenan.
- Los tipos de elementos de las tuplas es irrestricto.  
`('a'; (True; 'c')) :: (Char; (Bool; Char))`



# El Tipo Función

- Una función mapea valores de un tipo en valores de otro:
  - $not :: Bool \rightarrow Bool$
  - $even :: Int \rightarrow Bool$
- En general, Un tipo  $t_1 \rightarrow t_2$  mapea valores de  $t_1$  en valores de  $t_2$ .
- La flecha  $\rightarrow$  se introduce desde el teclado como  $- >$
- Los tipos de los argumentos o de los resultados de una función son irrestrictos.

$add :: (Int, Int) \rightarrow Int$

$add\ (x, y) = x + y$

$zeroto :: Int \rightarrow [Int]$

$zeroto\ n = [0..n]$

# Currificación de Funciones

- Es posible definir funciones con múltiples argumentos retornando una función como resultado.

$$\begin{aligned} add' &:: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\ add' \ x \ y &= x + y \end{aligned}$$

- $add$  y  $add'$  retornan lo mismo pero  $add'$  toma los argumentos de a uno en uno. Está currificada.
- Permite la aplicación parcial:

$$\begin{aligned} add' \ 3 &:: \text{Int} \rightarrow \text{Int} \\ take \ 5 &:: [\text{Int}] \rightarrow [\text{Int}] \\ drop \ 5 &:: [\text{Int}] \rightarrow [\text{Int}] \end{aligned}$$

# Convenciones de la Currificación

- Función que tome más de dos argumentos se pueden definir devolviendo funciones anidadas:

$$mult :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$$
$$mult\ x\ y\ z = x * y * z$$

- Como el constructor de tipos  $\rightarrow$  asocia a la derecha podemos no poner los paréntesis:

$$mult :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

- En consecuencia la aplicación de funciones asociando a la izquierda.

$$((mult\ x)\ y)\ z$$

*Salvo en el caso de las tuplas, todas las funciones en Haskell están currificadas*

- Una función es polimórfica si su tipo contiene una o más variables de tipo.

$length :: [a] \rightarrow Int$

*Para cualquier tipo  $a$  la función  $length$  es la misma.*

- Las variables de tipo pueden ser instanciadas a otros tipos

$length [False, True] \quad a = Bool$

$length ['a', 'b'] \quad a = Bool$

- Las variables de tipo se escriben con minúscula  $a$ ,  $b$ ,  $c$ , etc.

# Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$abs :: \text{Int} \rightarrow \text{Int}$

$abs\ n = \text{if } n > 0 \text{ then } n \text{ else } -n$

*Ambas ramas del **if** deben tener el mismo tipo.*

- Las expresiones condicionales pueden estar anidadas:

$signum :: \text{Int} \rightarrow \text{Int}$

$signum\ n = \text{if } n < 0 \text{ then } -1 \text{ else}$   
 $\quad \text{if } n \equiv 0 \text{ then } 0 \text{ else } 1$

*Las expresiones condicionales siempre deben tener la rama **else**, lo que elimina la posibilidad de ambigüedades cuando se anidan expresiones*

# Ecuaciones con Guardas

- Una alternativa a los condicionales es el uso de ecuaciones con guardas.

$$\begin{aligned} \text{abs } n \mid n > 0 &= n \\ \mid \text{otherwise} &= -n \end{aligned}$$

- Se usan para hacer ciertas definiciones más fáciles de leer.

$$\begin{aligned} \text{signum } n \mid n < 0 &= -1 \\ \mid n \equiv 0 &= 0 \\ \mid \text{otherwise} &= 1 \end{aligned}$$

- La condición *otherwise* se define en el preludio como

$$\text{otherwise} = \text{True}$$

- Muchas funciones se definen más claramente usando pattern matching (Ajuste de Patrones).

*not* :: Bool → Bool

*not* False = True

*not* True = False

- *not* mapea False a True y True a False
- Los patrones se componen de constructores de datos y variables.

# Pattern Matching

- Las funciones pueden ser definidas de distintas formas usando Pattern Matching:

```
(∧) :: Bool → Bool → Bool
True ∧ True = True
True ∧ False = False
False ∧ True = False
False ∧ False = False
```

puede ser escrita en forma compacta como

```
(∧) :: Bool → Bool → Bool
True ∧ True = True
_ ∧ _ = False
```

*El símbolo `_` es un patrón comodín que machea cualquier patrón.*



# Pattern Matching

- Las ecuaciones con patter Matching se evalúan en orden (Top-Down).  
Por ejemplo la siguiente definición siempre retorna False

$$\_ \wedge \_ = \text{False}$$
$$\text{True} \wedge \text{True} = \text{True}$$

- Las ecuaciones no pueden repetir variables  
Por ejemplo la siguiente definición da error:

$$b \wedge b = b$$
$$\_ \wedge \_ = \text{False}$$

# Patrones de Listas

- Toda lista no vacía se construye usando el operador  $(:)$  llamado “*cons*” que agrega un elemento al principio de la lista.

$[1, 2, 3, 4]$  resulta de  $1 : (2 : (3 : (4 : [])))$

- Por lo tanto, puedo definir funciones usando el patrón  $(x : xs)$

$$\text{head} :: [a] \rightarrow a$$
$$\text{head } (x : \_) = x$$
$$\text{tail} :: [a] \rightarrow [a]$$
$$\text{tail } (\_ : xs) = xs$$

$(x : xs)$  sólo machea el caso de listas no vacías.

Si aplicamos:

$\text{head } []$  nos da un error `***Exception: empty list`

- El patrón  $x : xs$  debe ir entre paréntesis ya que la aplicación tiene prioridad sobre el  $(:)$

Las funciones pueden construirse sin nombres usando expresiones lambda:

$$\lambda x \rightarrow x + x$$

Es la función que toma como entrada  $x$  y retorna el resultado  $x + x$ .

- El simbolo  $\lambda$  es la letra griega lambda y en el teclado se ingresa `\`.
- En Haskell el uso de  $\lambda$  para las funciones sin nombres proviene del Calculo Lambda que es la teoria funcional en la que se basa Haskell.

Las funciones lambdas son útiles para evitar darles nombres a funciones que va a ser usadas una sola vez:

```
odds n = map f [0..n-1]
  where
    f x = x * 2 + 1
```

Puede escribirse mas sintéticamente:

```
odds n = map (\x → x * 2 + 1) [0..n-1]
```

# Operadores de Secciones

- Un operador escrito entre sus dos argumentos (infijo), puede ser escrito antes de sus dos argumentos (prefijo) usando paréntesis:

$$\begin{array}{c} > (+) 1 2 \\ 3 \end{array}$$

- También uno de los argumentos puede ser incluido en los paréntesis

$$\begin{array}{c} > (1+) 2 \\ 3 \\ > (+2) 1 \\ 3 \end{array}$$

- En general, dado un operador  $\oplus$ , entonces las funciones de la forma  $(\oplus)$ ,  $(x\oplus)$ ,  $(\oplus y)$  son llamadas secciones.

# Conjuntos por comprensión

En matemáticas, una manera de construir conjuntos a partir de conjuntos existentes es con la notación por comprensión

$$\{x^2 | x \in \{1 \dots 5\}\}$$

*Describe el conjunto  $\{1, 4, 9, 16, 25\}$  o (lo que es lo mismo) el conjunto de todos los números  $x^2$  tal que  $x$  sea un elemento del conjunto  $\{1 \dots 5\}$*

En Haskell, una manera de construir listas a partir de listas existentes es con la notación por comprensión

$$[x \uparrow 2 \mid x \leftarrow [1..5]]$$

*Describe la lista  $[1, 4, 9, 16, 25]$  o (lo que es lo mismo) la lista de todos los números  $x \uparrow 2$  tal que  $x$  sea un elemento de la lista  $[1..5]$*

# Listas por comprensión

- La expresión  $x \leftarrow [1..5]$  es un **generador**, ya que dice como se generan los valores de  $x$ .
- Una lista por comprensión puede tener varios generadores, separados por coma.

$$> [(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$$
$$[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$$

- Cuando cambiamos el orden de los generadores cambiamos el orden de los elementos en la lista final

$$> [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$$
$$[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]$$

- Los múltiples generadores actúan como bucles anidados. Los generadores posteriores cambian más rápidamente.



# Generadores dependientes

- Un generador puede depender de variables introducidas por un generador anterior

$$[(x, y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

Esto es la lista de todos los pares  $(x, y)$  tal que  $x, y$  están en  $[1..3]$  e  $y \geq x$ .

- Usando generadores dependientes podemos definir *concat* definida en el prelude.

$$\begin{aligned} \text{concat} &:: [[a]] \rightarrow [a] \\ \text{concat } xss &= [x \mid xs \leftarrow xss, x \leftarrow xs] \end{aligned}$$

$$\begin{aligned} &> \text{concat } [[1, 2, 3], [4, 5], [6]] \\ &[1, 2, 3, 4, 5, 6] \end{aligned}$$

- Las listas por comprensión pueden usar **guardas** para restringir los valores producidos por generadores anteriores

$$[x \mid x \leftarrow [1..10], \text{even } x]$$

- Usando guardas podemos definir *factors*.

$$\text{factors} \quad :: \text{Int} \rightarrow [\text{Int}]$$
$$\text{factors } n = [x \mid x \leftarrow [1..n], n \text{ 'mod' } x \equiv 0]$$

- Como un número  $n$  es primo si y solo si sus únicos factores son 1 y  $n$ , podemos definir

$$\text{prime} :: \text{Int} \rightarrow \text{Bool}$$
$$\text{prime } n = \text{factors } n \equiv [1, n]$$
$$\text{primes} :: \text{Int} \rightarrow [\text{Int}]$$
$$\text{primes } n = [x \mid x \leftarrow [2..n], \text{prime } x]$$

# La función *zip*

- La función *zip*, mapea dos listas a una lista con los pares de elementos correspondientes

$$\begin{aligned} \text{zip} &:: [a] \rightarrow [b] \rightarrow [(a, b)] \\ &> \text{zip } ['a', 'b', 'c'] [1, 2, 3, 4] \\ &[( 'a', 1), ( 'b', 2), ( 'c', 3)] \end{aligned}$$

- Usando *zip* podemos definir una función que retorna la lista de pares de elementos adyacentes:

$$\begin{aligned} \text{pairs} &:: [a] \rightarrow [(a, a)] \\ \text{pairs } xs &= \text{zip } xs (\text{tail } xs) \end{aligned}$$
$$\text{pairs } [1, 2, 3] = [(1, 2), (2, 3)]$$

- Podemos usar *pairs* para decidir si una lista esta ordenada

$$\begin{aligned} \text{sorted} &:: \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool} \\ \text{sorted } xs &= \text{and } [x \leq y \mid (x, y) \leftarrow \text{pairs } xs] \end{aligned}$$

# La función *zip*

- Podemos usar *zip* para generar una función que retorne la listas de posiciones de un valor determinado

$$\text{positions } x \text{ } xs = [i \mid (x', i) \leftarrow \text{zip } xs \text{ } [0..], x \equiv x']$$

```
> positions 0 [1, 0, 0, 1, 0, 1, 1, 0]
[1, 2, 4, 7]
```

- Una String es una lista de caracteres `[Char]`.
- `"abc" :: String` significa `['a', 'b', 'c'] :: [Char]`

*Todas las funciones sobre listas son aplicables a String*

```
length "abcde"  
5
```

*Las listas por comprensión también pueden usarse para String.*

```
count x xs = length [x' | x' <- xs, x == x']
```

```
> count 's' "Mississippi"  
4
```

- Como hemos visto hasta ahora las funciones pueden definirse en termino de otras funciones.

$$\text{factorial } n = \text{product } [1..n]$$

- Las expresiones son evaluadas por pasos al aplicar las funciones a sus argumentos.

$$\begin{aligned}\text{fac } 4 \\ &= \text{product } [1..4] \\ &= \text{product } [1, 2, 3, 4] \\ &= 1 * 2 * 3 * 4 \\ &= 24\end{aligned}$$

# Funciones Recursivas

- En Haskell las funciones pueden ser definidas en termino de ellas mismas (recursión)

$$\textit{factorial } 0 = 1$$

$$\textit{factorial } n = n * \textit{factorial } (n - 1)$$

- Factorial mapea 0 en 1, y cualquier otro entero al producto de si mismo y el factorial de su predecesor.

$$\textit{factorial } 3$$

$$= 3 * \textit{factorial } 2$$

$$= 3 * (2 * \textit{factorial } 1)$$

$$= 3 * (2 * (1 * \textit{factorial } 0))$$

$$= 3 * (2 * (1 * 1))$$

$$= 3 * (2 * 1)$$

$$= 3 * 2$$

$$= 6$$

- La definición recursiva diverge en el caso  $<0$  por que el caso base nunca se alcanza.

$> factorial (-1)$

*\*\*\* Exception : stack overflow*

- Recursión sobre listas

$product :: Num\ a \Rightarrow [a] \rightarrow a$

$product [] = 1$

$product (n : ns) = n * product\ ns$



# Funciones Recursivas

- Usando la recursión podemos definir el reverso de una lista.

$$\begin{aligned} reverse &:: [a] \rightarrow [a] \\ reverse [] &= [] \\ reverse (x : xs) &= reverse xs \mathbin{++} [x] \end{aligned}$$

- `reverse` mapea la lista vacía `[]` en la lista vacía `[]`, y las no vacías al reverso de su cola *tail* concatenada con su cabeza *head*.

$$\begin{aligned} reverse [1, 2, 3] &= reverse [2, 3] \mathbin{++} [1] \\ &= (reverse [3] \mathbin{++} [2]) \mathbin{++} [1] \\ &= ((reverse [] \mathbin{++} [3]) \mathbin{++} [2]) \mathbin{++} [1] \\ &= ((([] \mathbin{++} [3]) \mathbin{++} [2]) \mathbin{++} [1]) \\ &= [3, 2, 1] \end{aligned}$$

- Funciones que toman más de un argumento también pueden ser definidas en forma recursiva.

$$\text{zip} :: [a] \rightarrow [b] \rightarrow [(a, b)]$$
$$\text{zip} [] \_ = []$$
$$\text{zip} \_ [] = []$$
$$\text{zip} (x : xs) (y : ys) = (x, y) : \text{zip} xs ys$$

```
> zip [1,2,3,4] ['a', 'b', 'c']  
[(1, 'a'), (2, 'b'), (3, 'c')]
```

- El algoritmo de ordenación Quicksort:
  - La lista vacía está ordenada
  - Las listas no vacías pueden ser ordenadas, ordenando los valores de la cola  $\leq$  que la cabeza, ordenando los valores  $>$  que la cabeza y rearmando el resultado con las listas resultantes a ambos lados de la cabeza.
- Su implementación:

$$\begin{aligned} qsort & \quad :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\ qsort [] & \quad = [] \\ qsort (x : xs) & = qsort \text{ chicos } ++ [x] ++ qsort \text{ grandes} \\ & \quad \textbf{where} \text{ chicos } = [a \mid a \leftarrow xs, a \leq x] \\ & \quad \text{grandes} = [b \mid b \leftarrow xs, b > x] \end{aligned}$$