# Programación Funcional con Haskell

Mauro Jaskelioff

22/03/2016

*Hay dos maneras de construir un diseño de software:*

*Una manera es hacerlo de manera tan simple que sea obvio que no hay deficiencias.*

*La otra es hacerlo de manera tan complicada que no haya deficiencias obvias.*

*Tony Hoare, 1980*

# Desafíos del Software

- Sistemas de software gigantes, cores múltiples, sistemas distribuidos.
- Costo y tiempo de desarrollo.
- Confianza en los programas desarrollados
- ¿Qué tecnología nos permitirá lidiar con los desafíos?

# La arquitectura de Von Neumann

- Una idea en los 1940' que simplificó en forma práctica y elegante varios problemas de ingeniería y programación.
- Las condiciones han cambiado, pero, la máquina de Von Neumann es, esencialmente, lo que ahora conocemos como computadoras.
- En su forma más simple, consiste de un CPU, una memoria y un tubito que los une.
- La tarea de los programas es modificar los contenidos de la memoria en forma significativa.
  - Todas las modificaciones se hacen a través del tubito!

# Lenguajes de Von Neumann

- Los lenguajes imperativos fueron creados para programar esta computadora.
- Son versiones complejas, de alto nivel, de la máquina de Von Neumann
  - **¡Se definen por su traducción a la máquina!**
- La característica de estos lenguajes es:
  - Las variables representan celdas de memoria.
  - Los comandos de control representan saltos y comparaciones.
  - La asignación imita la obtención y almacenamiento de datos.
  - Separación entre datos y programas.
- Dada la popularidad de los lenguajes de Von Neumann, otras arquitecturas no fueron desarrolladas.

# Algunas problemas de los programas imperativos

- Mantienen un estado invisible.
- Uno debe ejecutar los programas mentalmente para entenderlos.
- No son composicionales.
- Es difícil abstraer patrones comunes.
- Fuerzan un orden de ejecución.
- Semántica compleja (o directamente, dada por la implementación).
- Ausencia de propiedades matemáticas útiles.

# Ejemplo

Cálculo del producto escalar de dos vectores:

$$
\begin{array}{l}
c := 0 \\
\textbf{for } i := 1 \textbf{ step } 1 \textbf{ until } n \textbf{ do} \\
\quad c := c + a\,[i] * b\,[i]
\end{array}
$$

Comparemos con:

$$prod = sum \circ zipWith\ (*)$$

# Requerimientos para un Lenguaje de Programación Moderno

- Los programas deben poder ser escritos en forma clara, concisa, y a un alto nivel de abstracción.
- Se debe poder desarrollar componentes de software generales, reusables y modulares.
- La semántica debe ser clara para facilitar la verificación formal.
- Debe permitir el prototipado rápido y proveer herramientas potentes.
- Debe facilitar la programación paralela.
- Debe tener propiedades matemáticas útiles.

# Programación funcional

- La programación funcional cumple en buena medida con estos requerimientos.
- No usa un modelo de computación basado en máquinas sino en un lenguaje simple y elegante (el $\lambda$-cálculo).
- Su simpleza y versatilidad lo hacen ideal para aprender conceptos de
  - programación,
  - lenguajes de programación,
  - computabilidad,
  - semántica,
  - verificación, derivación, testing.

# ¿Qué es la programación funcional?

Hay diferentes opiniones.

- Es un *estilo* de programación en el cual el método básico de computar es aplicar funciones a argumentos
- Un lenguaje de programación funcional es un lenguaje que *permite* y *alienta* el uso de un estilo funcional.

# Haskell

- Haskell posee varias ventajas.
  - Programas Concisos
  - Sistemas de Tipos Poderoso
  - Funciones Recursivas
  - Fácilidad para probar propiedades de programas.
  - Funciones de Alto Orden
  - Evaluación Perezosa
  - Facilidad para definir DSLs
  - Efectos Monádicos
- Haremos un "repaso" por la sintaxis básica de Haskell.
- También introduciremos algunos conceptos nuevos.

# Preludio

- Muchas funciones de uso común están definidas en el Preludio (Prelude.hs)
- Además de las operaciones aritméticas usuales se definen muchas funciones que operan sobre *listas*.
  - *head*, *tail*, (!!), *take*, *drop*, *length*, *sum*, *product*, (++), *reverse*
- Leer el código del preludio puede ser muy instructivo (y sorprendente!).

# GHC

- Usaremos GHC, un compilador (*ghc*) e intérprete (ghci) de Haskell, que también soporta varias extensiones del mismo.
- URL: `http://www.haskell.org/ghc`
- Es conveniente instalar la *Haskell Platform*, que consiste del GHC junto con una colección de bibliotecas frecuentemente usadas.

# La aplicación de funciones

- en Haskell la aplicación se denota con un espacio y asocia a la izquierda.

| Matemáticas | Haskell |
|---|---|
| $f\ (x)$ | $f\ x$ |
| $f\ (x, y)$ | $f\ x\ y$ |
| $f\ (g\ (x))$ | $f\ (g\ x)$ |
| $f\ (x, (g\ (y))$ | $f\ x\ (g\ y)$ |
| $f\ (x)\ g\ (y)$ | $f\ x * g\ y$ |

- La aplicación tiene mayor precedencia que cualquier otro operador.

$$f\ x + y = (f\ x) + y$$

# Nombres y comentarios

- 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.
- 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**
- Los comentarios se escriben

```
-- comentario que finaliza junto con la línea

{- bloque de comentarios, útil para comentarios
que no entran en una sola línea. -}
```

# Offside Rule

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

$$
\begin{aligned}
a &= b + c \\
&\quad \textbf{where} \\
&\qquad b = 1 \\
&\qquad c = 2 \\
d &= a + 2
\end{aligned}
$$

- Gracias a esta regla no hace falta un sintaxis *explícita* para agrupar definiciones.

# Operadores infijos

- Los operadores infijos son funciones como cualquier otra.
- Una función se puede hacer infija con *backquotes*

$$10\ `div`\ 4 = div\ 10\ 4$$

- Se pueden definir operadores infijos usando alguno de los símbolos disponibles

$$a ** b = (a * b) + (a + 1) * (b - 1)$$

- La asociatividad y precedencia se indica usando **infixr** (asociativad der.), **infixl** (asociatividad izq.), o **infix** (si los paréntesis deben ser obligatorios)

**infixr** $6\ (**)$

# Tipos

- Un tipo es un nombre para una colección de valores
  - Ej: *Bool* contiene los valores *True* y *False*.
  - Escribimos *True* :: *Bool* y *False* :: *Bool*.
- En general, si una expresión *e* tiene tipo *t* escribimos

$$e :: t$$

- **En Haskell, toda expresión válida tiene un tipo**
- El tipo de cada expresión es calculado previo a su evaluación mediante la *inferencia de tipos*.
- Si no es posible encontrar un tipo (por ejemplo $(True + 4)$) el compilador/intérprete protestará con un *error de tipo*.

# Tipos básicos

Alguno de los tipos básicos de Haskell son:

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

# Listas

- Una lista es una secuencia de valores del *mismo* tipo
  - $[\mathit{True}, \mathit{True}, \mathit{False}, \mathit{True}] :: [\mathit{Bool}]$
  - $[\texttt{'h'}, \texttt{'o'}, \texttt{'l'}, \texttt{'a'}] :: [\mathit{Char}]$
- En general, $[t]$ es una lista con elementos de tipo $t$.
- $t$, puede ser *cualquier* tipo válido.
  - $[[\texttt{'a'}], [\texttt{'b'}, \texttt{'c'}], []] :: [[\mathit{Char}]]$
- No hay restricción con respecto a la longitud de las listas.

# Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True, \texttt{'a'}, \texttt{'b'}) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una $n$-tupla cuyas componente $i$ tiene tipo $t_i$, para $i$ en $1 \,..\, n$.
- A diferencia de las listas, las tuplas tienen explicitado en su tipo la *cantidad* de elementos que almacenan.
- Los tipos de las tuplas no tiene restricciones
  - $(\texttt{'a'}, (True, \texttt{'c'})) :: (Char, (Bool, Char))$
  - $((['\texttt{a}', '\texttt{b}'], '\texttt{a}'), '\texttt{b}') :: (([Char], Char), Char)$

# Funciones

- Una función mapea valores de un tipo en valores de otro
  - $not :: Bool \to Bool$
  - $isDigit :: Char \to Bool$
- En general, Un tipo $t_1 \to t_2$ mapea valores de $t_1$ en valores de $t_2$.
- Se pueden escribir funciones con múltiples argumentos o resultados usando tuplas y listas.

$$
\begin{aligned}
&add && :: (Int, Int) \to Int \\
&add\ (x, y) && = x + y \\
&deceroa && :: Int \to [Int] \\
&deceroa\ n && = [0\,..\,n]
\end{aligned}
$$

# Currificación y aplicación parcial

- Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$add' :: Int \to (Int \to Int)$$
$$add'\ x\ y = x + y$$

- A diferencia de *add*, *add′* toma los argumentos de a uno por vez. Se dice que *add′* está currificada.

- La ventaja de la versión currificada es que permite la *aplicación parcial*:

$$suma3 :: Int \to Int$$
$$suma3 = add'\ 3$$

# Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: Int \to (Int \to (Int \to Int))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención el constructor de tipos $\to$ asocia a la derecha.
  $mult :: Int \to Int \to Int \to Int$
- Notar que esta convención es consistente con la aplicación asociando a la izquierda.
- A menos que la función lo requiera, en Haskell todas las funciones están currificadas.

# Nombres de los tipos

- A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.
- El espacio de nombres de los tipos está completamente separado del espacio de nombres de las expresiones.

# Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo.
  $length :: [a] \rightarrow Int$
  Para cualquier tipo $a$ la función *length* es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser *instanciadas* a otros tipos

$$\begin{array}{ll} length\ [False, True] & \leftarrow a = Bool \\ length\ [\texttt{'a'}, \texttt{'b'}] & \leftarrow a = Char \end{array}$$

- A veces se llama polimorfismo paramétrico a este tipo de polimorfismo.

# Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas
- Esta sobrecarga se realiza mediante *clases de tipo*.
- El operador $(+)$ tiene tipo:

$$(+) :: \mathit{Num}\ a \Rightarrow a \to a \to a$$

- $(+)$ está definido para cualquier tipo que sea una *instancia* de la clase *Num*.
- A diferencia del polimorfismo paramétrico, hay una definición distinta de $(+)$ para cada instancia.

# Clases de tipo

- Haskell provee una serie de clases básicas:
- *Eq* son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$
$$(/=) :: Eq\ a \Rightarrow a \to a \to Bool$$

¿Qué tipo no puede ser instancia de esta clase?

- *Ord* son los tipos que además de ser instancias de *Eq* poseen un orden total.

$$(<), (\leqslant), (>), (\geqslant) :: Ord\ a \Rightarrow a \to a \to Bool$$
$$min, max :: Ord\ a \Rightarrow a \to a \to a$$

# Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

  $$show :: Show\ a \Rightarrow a \rightarrow String$$

- *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

  $$read :: Read\ a \Rightarrow String \rightarrow a$$

- ¿A qué tipo corresponde la instancia que leerá
  - *not* (*read* `"False"`)?
  - *read* `"3"`?
  - *read* `"3"` :: *Int*?

# Clases de Tipo

- *Num* son los tipos numéricos
- Sus instancias deben implementar

$$(+), (-), (*) \quad :: Num\ a \Rightarrow a \to a \to a$$
$$negate, abs, signum :: Num\ a \Rightarrow a \to a$$

¿Y la división?

- *Integral* son los tipos que son *Num* e implementan

$$div, mod :: Integral\ a \Rightarrow a \to a \to a$$

- *Fractional* son los tipos que son *Num* e implementan

$$(/) \quad :: Fractional\ a \Rightarrow a \to a \to a$$
$$recip :: Fractional\ a \Rightarrow a \to a$$

# Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$
\begin{aligned}
&\mathit{abs} \quad :: \mathit{Int} \to \mathit{Int} \\
&\mathit{abs}\ n = \textbf{if}\ n \geqslant 0\ \textbf{then}\ n\ \textbf{else}\ -n
\end{aligned}
$$

- Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo.
- Las expresiones condicionales siempre deben tener la rama "**else**"
- Por lo tanto no hay ambigüedades en caso de anidamiento:

$$
\begin{aligned}
&\mathit{signum} \quad :: \mathit{Int} \to \mathit{Int} \\
&\mathit{signum}\ n = \textbf{if}\ n < 0\ \textbf{then}\ -1\ \textbf{else} \\
&\qquad\qquad\qquad\qquad \textbf{if}\ n == 0\ \textbf{then}\ 0\ \textbf{else}\ 1
\end{aligned}
$$

# Ecuaciones con Guardas

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

$$\begin{array}{l|ll} \mathit{abs}\ n & n \geqslant 0 & = n \\ & \mathit{otherwise} & = -n \end{array}$$

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

$$\begin{array}{l|ll} \mathit{signum}\ n & n < 0 & = -1 \\ & n == 0 & = 0 \\ & \mathit{otherwise} & = 1 \end{array}$$

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

$$\mathit{otherwise} = \mathit{True}$$

# Pattern Matching

- Muchas funciones se definen más claramente usando *pattern matching*.

$$
\begin{aligned}
&\mathit{not} && :: \mathit{Bool} \to \mathit{Bool} \\
&\mathit{not}\ \mathit{False} && = \mathit{True} \\
&\mathit{not}\ \mathit{True} && = \mathit{False}
\end{aligned}
$$

- Los patrones se componen de constructores de datos y variables (salvo los patrones $0$ y $n+1$) .
- Una variable es un patrón que nunca falla.

$$
\begin{aligned}
&\mathit{succ} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{succ}\ n = n + 1
\end{aligned}
$$

# Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

$$
\begin{aligned}
(\wedge) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} \wedge \mathit{True} &= \mathit{True} \\
\mathit{True} \wedge \mathit{False} &= \mathit{False} \\
\mathit{False} \wedge \mathit{True} &= \mathit{False} \\
\mathit{False} \wedge \mathit{False} &= \mathit{False}
\end{aligned}
$$

  puede ser escrita en forma compacta como

$$
\begin{aligned}
(\wedge) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} \wedge \mathit{True} &= \mathit{True} \\
\_ \quad \wedge \_ \quad &= \mathit{False}
\end{aligned}
$$

- Notar la importancia del orden de las ecuaciones.

# Patrones de tuplas

- Una tupla de patrones es un patrón.

$$
\begin{aligned}
&\mathit{fst} && :: (a, b) \to a \\
&\mathit{fst}\ (x, \_) && = x \\
&\mathit{snd} && :: (a, b) \to b \\
&\mathit{snd}\ (\_, y) && = y
\end{aligned}
$$

- ¿Qué hace la siguiente función?

$$f\ (x, (y, z)) = ((x, y), z)$$

- En general, los patrones pueden anidarse

# Patrones de Listas

- Toda lista (no vacía) se contruye usando el operador (:) (llamado *cons*) que agrega un elemento al principio de la lista

$$[1,2,3,4] \quad \mapsto \quad 1:(2:(3:(4:[])))$$

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

$$\begin{aligned} &\mathit{head} && :: [a] \to a \\ &\mathit{head}\ (x{:}\ \_) && = x \\ &\mathit{tail} && :: [a] \to [a] \\ &\mathit{tail}\ (\_ : xs) && = xs \end{aligned}$$

- $(x : xs)$ sólo matchea el caso de listas no vacías $>\mathit{head}\ []$ *Error*!

# Expresiones Lambda

- Se pueden construir funciones sin darles nombres usando *expresiones lambda*.

$$\lambda x \to x + x$$

- Estas funciones se llaman *funciones anónimas*.
- En Haskell escribimos $\backslash$ en lugar de la letra griega lambda $\lambda$
- La notación proviene del cálculo lambda, en el cual se basa Haskell, y que estudiaremos en detalle más adelante.

# Utilidad de las expresiones lambda

- Usando lambdas puedo explicitar que las funciones son simplemente valores:

$$add\ x\ y = x + y$$
$$add = \lambda x \to (\lambda y \to x + y)$$

- Evito tener que darle un nombre a una función que se usa una sola vez.

$$impares\ n = map\ f\ [0\,..\,n-1]$$
$$\textbf{where}\ f\ x = x * 2 + 1$$

$$odds\ n = map\ (\lambda x \to x * 2 + 1)\ [0\,..\,n-1]$$

# Secciones

- Un operador infijo, puede ser escrito en forma prefija usando paréntesis:

$$
\begin{aligned}
&> (+)\ 1\ 2 \\
&3
\end{aligned}
$$

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

$$
\begin{aligned}
&> (1+)\ 2 \\
&3 \\
&> (+2)\ 1 \\
&3
\end{aligned}
$$

- 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

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

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 $\{1 \ldots 5\}$

# Listas por comprensión

- 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 $[1\,..\,5]$

- 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.

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

- ¿Qué pasa cuando cambiamos el orden de los generadores?

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

- Los generadores posteriores cambian más rápidamente.

# Generadores dependientes

- Un generador puede depender de 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 $y \geqslant x$.

- ¿Qué hace la siguiente función?

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

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

# Guardas

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

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

- ¿Qué hace la siguiente función?

$$\begin{aligned}&\mathit{factors} \quad :: \mathit{Int} \to [\mathit{Int}]\\&\mathit{factors}\ n = [x \mid x \leftarrow [1\,..\,n], n\ `\mathit{mod}`\ x == 0]\end{aligned}$$

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

$$\begin{aligned}&\mathit{prime} :: \mathit{Int} \to \mathit{Bool}\\&\mathit{prime}\ n = \mathit{factors}\ n == [1, n]\\&\mathit{primes} :: \mathit{Int} \to [\mathit{Int}]\\&\mathit{primes}\ n = [x \mid x \leftarrow [2\,..\,n], \mathit{prime}\ x]\end{aligned}$$

# Cadenas

- Una *String* es una lista de caracteres.
- `"Hola"` :: *String*
- `"Hola"` $=$ `['H','o','l','a']`
- Todas las funciones sobre listas son aplicables a *String*, y las listas por comprensión pueden ser aplicadas a *String*s.

$$
\begin{aligned}
&\mathit{cantminusc} \quad :: \mathit{String} \to \mathit{Int} \\
&\mathit{cantminusc}\ \mathit{xs} = \mathit{length}\ [x \mid x \leftarrow \mathit{xs}, \mathit{isLower}\ x]
\end{aligned}
$$

# Zip

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

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

- Ejemplo: Lista de pares de elementos adyacentes:

$$
\begin{aligned}
&pairs \quad :: [a] \to [(a, a)] \\
&pairs\ xs = zip\ xs\ (tail\ xs)
\end{aligned}
$$

- ¿Está una lista ordenada?

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

# Ejemplo *zip*: pares índice/valor

- Podemos usar *zip* para generar índices

$$
\begin{aligned}
&\mathit{rangeof} && :: \mathit{Int} \to \mathit{Int} \to [a] \to [a] \\
&\mathit{rangeof}\ \mathit{low}\ \mathit{hi}\ \mathit{xs} &&= [x \mid (x, i) \leftarrow \mathit{zip}\ \mathit{xs}\ [0\,..], \\
& && \qquad i \geqslant \mathit{low}, \\
& && \qquad i \leqslant \mathit{hi}]
\end{aligned}
$$

$$
\begin{aligned}
&> [x \uparrow 2 \mid x \leftarrow [1\,..\,10]] \\
&[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] \\
&> \mathit{rangeof}\ 3\ 7\ [x \uparrow 2 \mid x \leftarrow [1\,..\,10]] \\
&[16, 25, 36, 49, 64]
\end{aligned}
$$

# Recursión

- En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

$$
\begin{aligned}
&\mathit{factorial} \quad :: \mathit{Int} \to \mathit{Int} \\
&\mathit{factorial}\ 0 = 1 \\
&\mathit{factorial}\ n = n * \mathit{factorial}\ (n-1)
\end{aligned}
$$

- ¿Qué sucede con *factorial n* cuando $n < 0$?
- Recursión sobre listas

$$
\begin{aligned}
&\mathit{length} &&:: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] &&= 0 \\
&\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs
\end{aligned}
$$

# Recursión Mutua

- No hace falta ninguna sintaxis especial para la recursión mutua.

$$
\begin{aligned}
&\mathit{zigzag} :: [(a, a)] \to [a] \\
&\mathit{zigzag} = \mathit{zig} \\
&\mathit{zig}\ [\,] &&= [\,] \\
&\mathit{zig}\ ((x, \_) : \mathit{xs}) &&= x : \mathit{zag}\ \mathit{xs} \\
&\mathit{zag}\ [\,] &&= [\,] \\
&\mathit{zag}\ ((\_, y) : \mathit{xs}) &&= y : \mathit{zig}\ \mathit{xs}
\end{aligned}
$$

- $> \mathit{zigzag}\ [(1,2),(3,4),(5,6),(7,8)]$
  $[1,4,5,8]$

# Quicksort

- El algoritmo de ordenación Quicksort:
  - La lista vacía está ordenada
  - Las listas no vacías pueden ser ordenadas, ordenando los valores de la cola $\leqslant$ 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:

$$
\begin{aligned}
&\mathit{qsort} && :: \mathit{Ord}\ a \Rightarrow [a] \to [a] \\
&\mathit{qsort}\ [\,] && = [\,] \\
&\mathit{qsort}\ (x : xs) && = \mathit{qsort}\ \mathit{chicos} +\!\!+ [x] +\!\!+ \mathit{qsort}\ \mathit{grandes} \\
&\quad \textbf{where}\ \mathit{chicos} && = [a \mid a \leftarrow xs, a \leqslant x] \\
&\qquad\qquad \mathit{grandes} && = [b \mid b \leftarrow xs, b > x]
\end{aligned}
$$

# Referencias


- "Can Programming Be Liberated from the von Neumann Style?". John Backus, Turing Award Lecture. 1977
- *Why Functional Programming Matters*. John Hughes, Computing Journal. Vol 32. 1989.
- *Programming in Haskell*. Graham Hutton, CUP 2007.
- *Introducción a la Programación Funcional con Haskell*. Richard Bird, Prentice Hall 1997.