# Paradigma Funcional

## Módulo 6:
## Expresiones lambdas

**por Fernando Dodino**
**Franco Bulgarelli**
**Matías Freyre**
**Carlos Lombardi**
**Nicolás Passerini**
**Daniel Solmirano**

**Versión 2.0**
**Marzo 2017**

## Contenido

# 1 Introducción

Haskell está basado en cálculo lambda[1], que es un sistema de reglas de transformación o reductor de expresiones que diseñó Alonzo Church.

λ x : x * x  es una expresión lambda que denota la función cuadrado

En Haskell se codifica así:

```
\x -> x * x
```

Y también podemos definirla como:

```
cuadrado = \x -> x * x
```

La contrabarra (\) es el símbolo que remite a la letra griega lambda λ que como habrán notado es el ícono de la programación funcional. Luego de los parámetros que se separan por espacios, la flecha `->` termina de definir el cuerpo de la función.

Se evalúa de esta manera:

```
λ (\x -> x * x) 2
4
```

Otro ejemplo:

```
λ (\x y -> x + y) 2 3
5
```

Las expresiones lambdas permiten definir funciones *anónimas* que se usan en un contexto limitado (el de la misma función que estoy definiendo). Al no tener nombre es una variante menos expresiva que una función cuadrado o suma, que además se pueden utilizar en diferentes contextos. No obstante son una herramienta muy útil, como veremos más adelante.

# 2 Definición local de una función

Otra forma de escribir lo mismo

```
sumar 2 3 where sumar x y = x + y
```

Si quiero saber el número de raíces de una ecuación cuadrática:

$ax^2 + bx - c = 0$

[1] Para más información ver https://en.wikipedia.org/wiki/Lambda_calculus, http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf

```
numeroDeRaices a b c
      | discriminante > 0  = 2
      | discriminante == 0 = 1
      | discriminante < 0  = 0
  where discriminante = b*b - 4*a*c
```

Defino una expresión en un solo lugar. Tiene un nombre explícito, lo que ayuda a su comprensión y la ventaja de no tener que definirse por "afuera", aunque también sirve solamente en el contexto local de una función.

# 3 Uso efectivo de lambdas

## 3.1 Lambdas con filter y map

Las lambdas son útiles cuando queremos trabajar con funciones de orden superior y no tenemos necesidad de reutilizar una expresión en otro contexto:

```
λ filter (\cliente -> edad cliente > 40) clientes
```

en muchos casos podemos resolver el mismo problema con composición y aplicación parcial:

```
λ filter ((> 40) . edad) clientes
```

Y **ciertamente es la opción que vamos a preferir en la cursada**, ya que no solo es más conciso el código sino que demuestra más entendimiento de los conceptos del paradigma.

Presentamos algunos contraejemplos en donde el uso de funciones anónimas nos facilita resolver ciertos requerimientos, como encontrar los clientes que tengan menos de 20 años ó más de 60:

```
λ filter ((\e -> e < 20 || e > 60) . edad) clientes
```

Y volviendo al ejemplo de flip del capítulo anterior, si tenemos una función que nos dice si la facturación de un cliente supera un determinado monto:

```
facturacionSupera cliente monto =
     ((> monto) . sum . facturas) cliente
```

Y queremos saber quiénes son los clientes cuya facturación supera los 100.000, una opción era utilizar flip para aplicar la función con el orden de los parámetros invertidos:

```
λ filter (flip facturacionSupera 100000) clientes
```

Otra opción es definir una *lambda abstraction*:

```
λ filter (\cliente -> facturacionSupera cliente 100000) clientes
```

Por último, un ejemplo con una función anónima ad-hoc que permite sumar los elementos de una tupla:

```
λ map (\(a, b) -> a + b ) [(1, 2), (3, 5), (6, 3), (2, 6), (2, 5)]
[3, 8, 9, 8, 7]
```

## 3.2 Consecuencias de las lambdas

Algo importante a tener en cuenta es que si no le damos un nombre a nuestras funciones podríamos perder abstracciones útiles que podrían luego ser utilizadas en otros puntos de nuestro programa, por lo tanto es importante ser criteriosos respecto nombrar o no una función.

Por lo general, si tengo una forma sencilla de nombrar una determinada lógica que forma parte de una función más grande, lo más probable es que no quiera definir ese pedacito de lógica usando una lambda, sino con una función que se llame como la idea que tenemos en la cabeza. Si no hay un nombre claro asociado a ese pedacito de lógica, lo más probable es que no sea un concepto del dominio que merezca la pena modelar como algo aparte.

## 3.3 Lambdas con fold

Si un empleado es una estructura definida de la siguiente manera

```
data Empleado = Empleado {
      nombre :: String,
      sueldo :: Float,
      cantidadHijos :: Int,
      sector :: String
} deriving (Show)
```

Y dada una lista de empleados

```
type Nomina = [Empleado]

empleados :: Nomina
empleados = [ Empleado "mara" 17000.0 1 "contaduria",
  Empleado "gerardo" 15000.0 2 "ventas", ...
```

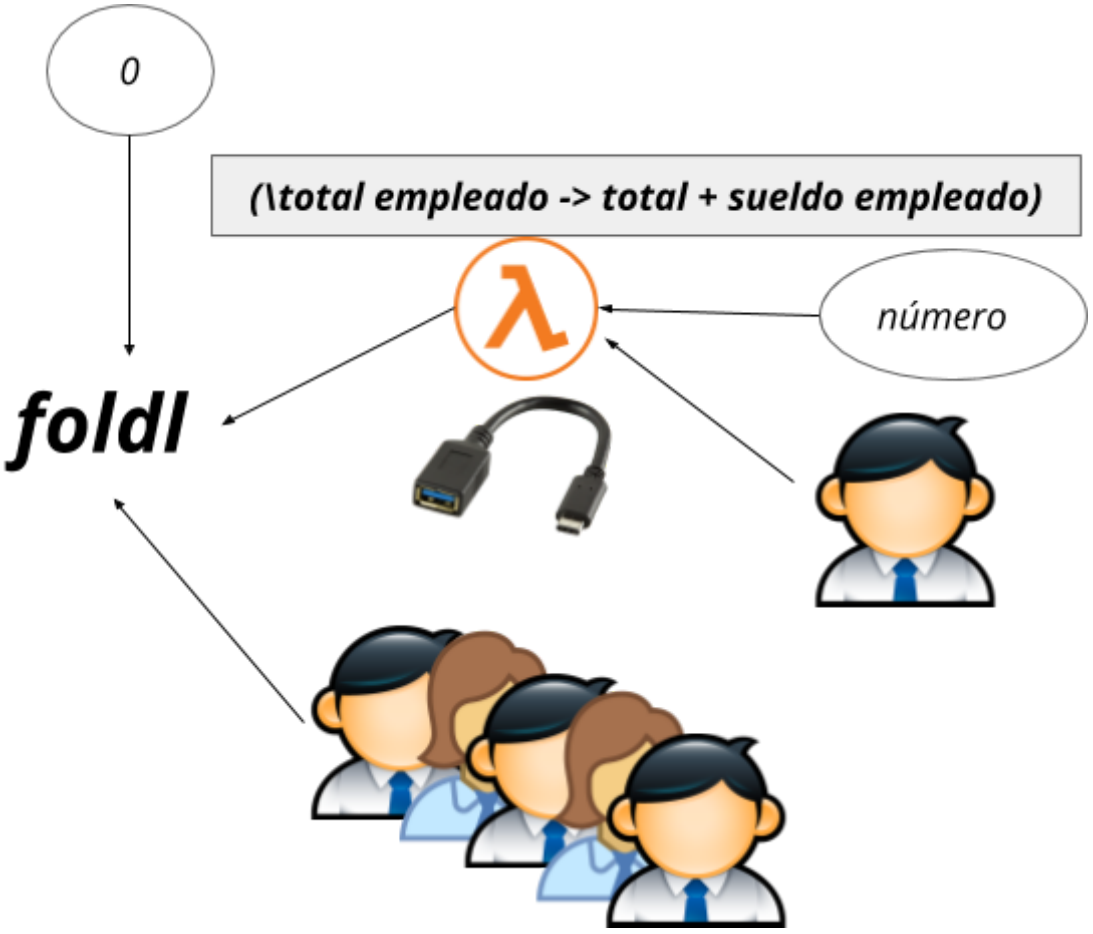
Podemos saber cuál es el monto total que paga la empresa en sueldos, haciendo una consulta por consola:

```
λ foldl (\total empleado -> total + sueldo empleado) 0.0 empleados
```

También podemos saber cuántos hijos tienen los empleados en total:

```
λ foldl (\total empleado -> total + cantidadHijos empleado) 0 empleados
```

La expresión lambda que construimos sirve como *adapter* para sumarizar los valores que nosotros queremos, ya que la suma solo trabaja con enteros, no con un número y un empleado.




Otra opción es utilizar foldr y trabajar con una función adapter *ad-hoc*:

```
λ foldr ((+) . sueldo) 0 empleados
32000.0
```

O utilizar una expresión lambda

```
λ foldr (\empleado total -> total + sueldo empleado) 0.0 empleados
```

### 3.4 Folds especiales

Se agrega el siguiente requerimiento: queremos saber qué empleado es el que más gana de todos.

Utilizaremos *foldl1* ya que no tiene sentido obtener el mayor de una lista de empleados vacía. Pensamos qué *función anónima* sería útil

- recibimos dos empleados
- devolvemos el empleado que gane más

```
(\empleado1 empleado2 -> if (sueldo empleado1 > sueldo empleado2) then empleado1 else empleado2)
```

Sí, suena un poco extraño pero podemos obtener el empleado que gane más con un fold + una expresión lambda que sepa comparar empleados (en este caso por sueldo):

```
λ foldl1 (\empleado1 empleado2 -> if (sueldo empleado1 > sueldo empleado2) then empleado1 else empleado2) empleados
```

## 4 Currificación

### 4.1 Definición

Volviendo al ejemplo

```
λ map (\(a, b) -> a + b) [(1, 2), (3, 5), (6, 3), (2, 6), (2, 5)]
```

Fíjense que la expresión lambda es la suma pero aplicado a una tupla de 2 elementos:

```
\(a, b) -> a + b
```

mientras que el operador (+) trabaja con dos argumentos

```
\ a b -> a + b
```

¿Qué diferencia hay entre ambas expresiones?

1. En la primera solución, no podemos trabajar con aplicación parcial. Decimos que la función está sin *currificar*.

```
λ (\(a, b) -> a + b) (5, 3)
8
λ (\(a, b) -> a + b) (5)
¡Error!
```

2. En la segunda sí podemos aplicar parcialmente la suma, y decimos que la función está *currificada*.

```
λ (\ a b -> a + b) 5
<function>
```

En general, una función
```
f :: a -> b -> c
```

es la forma *currificada* de la función que tiene un solo argumento (una tupla):
```
g :: (a, b) -> c
```
*g* es una función *sin currificar* (como lo conocemos en C o Pascal)

El lector interesado puede revisar dos funciones: *curry* y *uncurry*, que nos recuerdan al matemático y lógico estadounidense Haskell Curry (sí, también en honor a él es el lenguaje que estamos aprendiendo).

Curry transforma una función sin currificar a una que admita aplicación parcial:
```
multiplicaPar :: Num a => (a, a) -> a
multiplicaPar (a, b) = a * b
```

```
λ map ((curry multiplicaPar) 2) [1..5]
[2, 4, 6, 8, 10]
```

## 4.2 Cómo la currificación permite la aplicación parcial

Si recordamos la función (*), y evaluamos en la consola
```
λ 2 * 3
```

Uno podría pensar que el operador (*) recibe dos argumentos y devuelve un número:
```
(*) :: Num a =>  a    ->    a   -> a
                 ↓          ↓
             1# arg     2# arg
```

La realidad es que Haskell considera que (*) es una función que recibe un número y devuelve una función, que termina multiplicando por ese número:
```
(*) :: Num a => a -> (a -> a)
```

Es decir que la reducción se hace:
```
λ (2 *) 3
```

asociando a derecha, así puedo armar una función aplicando parcialmente sus argumentos:
Si el tipo de (*) es

```
Numero -> (Numero -> Numero)
```

El tipo de (2 *) es

```
Numero -> Numero
```

o sea (2 *) me devuelve una función (que dado un número te devuelve el doble de ese número).

## 4.3 Suma de números

¿Cómo describimos la suma de dos números?

```
(\x y -> x + y)
```

Alonzo Church desarrolló el cálculo lambda proponiendo que todas las funciones sean de un solo parámetro, por lo tanto la función suma también puede escribirse así:

```
\x -> (\y -> x + y)
```

Entonces la sumatoria de dos números empieza a verse como una función que recibe un número... y devuelve una función. Esto permite que yo pueda aplicarla parcialmente, si x = 1, entonces de esa manera estoy construyendo una lambda que dado un número me da su valor siguiente:

```
λ (\x -> (\y -> x + y)) 1 =
(\y -> 1 + y)
```

Que no es otra cosa que la función siguiente

```
siguiente = (1 +)
```

Está claro que el tipo de siguiente es:

```
Num a => a -> a
```

Es decir, dado un número te devuelve el siguiente.

## 4.4 Con qué letra empieza una palabra

Queremos saber si una palabra empieza con una letra determinada.

Escribimos una solución

```
type Palabra = String

empiezaCon :: Char -> Palabra -> Bool
empiezaCon letra palabra = ((== letra) . head) palabra
```

Ahora vamos a entender cómo es que se resuelve el mecanismo de aplicación parcial. Traducimos la función empiezaCon a expresiones lambdas, aplicando la restricción de que cada lambda solo tiene un parámetro:

```
(\letra -> (\palabra -> ((== letra) . head) palabra))
```

Si la letra es un valor conocido, por ejemplo la letra 'a':

```
λ (\letra -> (\palabra -> ((== letra) . head) palabra)) 'a'
```

Esto me da... una función que me dice si una palabra empieza con a:

```
(\palabra -> ((== 'a') . head) palabra)
```

(reemplazando el parámetro con 'a')

Por el contrario si hacemos:

```
λ (\letra -> (\palabra -> ((== letra) . head) palabra)) 'r' "rosa"
```

Lo que obtenemos es True... porque se reemplaza letra por r y palabra por rosa.

```
λ ((== 'r') . head) "rosa"
True
```

# 5 Resumen

En el presente capítulo introdujimos las expresiones lambda como una forma de generar funciones anónimas, que son útiles para poder pasarse a funciones de orden superior y también para entender que todas las funciones se definen como una secuencia de expresiones lambda que reciben un parámetro, en ese caso decimos que la función está *currificada* y se puede aplicar parcialmente.

Por ejemplo, la función map parece tener 2 argumentos: una función de conversión y una lista, y devuelve una nueva lista correspondiente al resultado de aplicar la función a cada elemento de la lista.

```
map :: (a -> b) -> [a] -> [b]
```

Pero, en realidad, lo que pasa es que la función map recibe sólo una función (`a -> b`) y devuelve otra nueva función, la cual recibe una lista. O sea, el tipo anterior es equivalente al siguiente:

```
map :: (a -> b) -> ([a] -> [b])
```

Viendo lo mismo en una aplicación concreta, se puede hacer:

```
λ map length ["Hola","mundo"]
```

Y es lo mismo que:

```
λ (map length) ["Hola","mundo"]
```

Ya que `map length` es el resultado de map "con su único parámetro instanciado", y es una función de tipo `[a] -> [b]`.

Entonces, gracias a la currificación podemos hacer cosas como (2 *), siendo que en realidad la función (*) es una función de 2 argumentos. En C y en Pascal yo no tengo esa posibilidad: `mayor(8, 4)` no se puede descomponer. En Haskell podemos recibir una tupla en lugar de dos argumentos:

```
mayor :: (Int, Int) -> Int
mayor (a, b) | a > b     = a
             | otherwise = b
```

Pero no queremos definir la función sin currificar, porque queremos aprovechar la posibilidad de aplicarla parcialmente.

Anotamos en el pizarrón:

- Qué es currificar una función: tener n argumentos en lugar de un argumento solo (una tupla de n elementos)

Función sin currificar `map :: ((a -> b), [a]) -> [b]`

Función currificada `map :: (a -> b) -> [a] -> [b]`

Que se asocia a derecha:

`map :: (a -> b) -> ([a] -> [b])`

- currificar una función permite poder aplicarla parcialmente.