

Introducción a funciones de alto orden

Taller de Álgebra I

Segundo cuatrimestre 2018

Mapa del curso (Herramientas computacionales)

- ▶ Introducción a la computación
- ▶ Problema \rightarrow Algoritmo \rightarrow Programa
- ▶ Tipado
- ▶ Reducción y órdenes de evaluación
- ▶ Recursión sobre números
- ▶ Recursión sobre listas
- ▶ Pattern matching
- ▶ Mini introducción a alto orden (usted está aquí)

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función adicional?

$$(f \circ g)(x) = f(g(x))$$

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función adicional?

$$(f \circ g)(x) = f(g(x))$$

¡Sí! para eso usamos la función (.)

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función adicional?

$(f \circ g)(x) = f(g(x))$

¡Sí! para eso usamos la función `(.)`

¿Qué devuelve esto?

```
> (f . g) 2
> (.) f g 2
```

Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función adicional?

$(f \circ g)(x) = f(g(x))$

¡Sí! para eso usamos la función `(.)`

¿Qué devuelve esto?

```
> (f . g) 2
> (.) f g 2
```


Componer funciones

Vimos

- ▶ Cómo definir funciones simples.
- ▶ Cómo definir funciones partidas.
- ▶ Cómo definir funciones recursivas.
- ▶ Cómo aplicar funciones.

Veamos lo siguiente

```
f :: Integer -> Integer
f x = 2 * x

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función adicional?

$(f \circ g)(x) = f(g(x))$

¡Sí! para eso usamos la función `(.)`

¿Qué devuelve esto?

```
> (f . g) 2
> (.) f g 2
```

6

¿Qué tipo tendrá `(.)`?

Funciones como parámetro

Las funciones pueden **tomar** parámetros de cualquier tipo, ¿no?

```
(.) :: <FuncionF> -> <FuncionG> -> DominioG -> CodominioF
```

```
(.) f g x = f (g x)
```

Funciones como parámetro

Las funciones pueden **tomar** parámetros de cualquier tipo, ¿no?

```
(.) :: <FuncionF> -> <FuncionG> -> DominioG -> CodominioF  
(.) f g x = f (g x)
```

Entonces las funciones tienen que tener tipo...

Valores

- ▶ Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- ▶ Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`,

Funciones como parámetro

Las funciones pueden **tomar** parámetros de cualquier tipo, ¿no?

```
(.) :: <FuncionF> -> <FuncionG> -> DominioG -> CodominioF  
(.) f g x = f (g x)
```

Entonces las funciones tienen que tener tipo...

Valores

- ▶ Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- ▶ Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

Funciones como parámetro

Las funciones pueden **tomar** parámetros de cualquier tipo, ¿no?

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(.) f g x = f (g x)
```

Entonces las funciones tienen que tener tipo...

Valores

- ▶ Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- ▶ Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

¿Qué devuelve como valor de salida `(.)`?

Una función muy útil

¿Qué hace la siguiente función?

```
magia _ [] = []  
magia f (x:xs) = (f x) : magia f xs
```

Una función muy útil

¿Qué hace la siguiente función?

```
magia _ [] = []  
magia f (x:xs) = (f x) : magia f xs
```

Magia

- ▶ ¿Cuál es el tipo de magia?

Una función muy útil

¿Qué hace la siguiente función?

```
magia _ []      = []  
magia f (x:xs) = (f x) : magia f xs
```

Magia

- ▶ ¿Cuál es el tipo de magia?
- ▶ ¿A qué reducen las siguientes expresiones?
 - ▶ `magia not [True, True, False, True]`
 - ▶ `magia reverse [[1,2,3], [5,6,7]]`
 - ▶ `magia id [1,2,3,4,5,6]`

Una función muy útil

¿Qué hace la siguiente función?

```
magia _ [] = []  
magia f (x:xs) = (f x) : magia f xs
```

Magia

- ▶ ¿Cuál es el tipo de magia?
- ▶ ¿A qué reducen las siguientes expresiones?
 - ▶ `magia not [True, True, False, True]`
 - ▶ `magia reverse [[1,2,3], [5,6,7]]`
 - ▶ `magia id [1,2,3,4,5,6]`

Esta función `magia` viene programada, se llama `map`.

Dada su utilidad, suele estar disponible en varios lenguajes de programación.

Otro problema frecuente

Prototipo de función muy común

```
func []      = []  
func (x:xs) | (alguna condición) = x : func xs  
           | otherwise          = func xs
```

Otro problema frecuente

Prototipo de función muy común

```
func []      = []  
func (x:xs) | (alguna condición) = x : func xs  
           | otherwise          = func xs
```

Podemos pensar la condición como una función auxiliar `cond :: a -> Bool`.

Otro problema frecuente

Prototipo de función muy común

```
func []      = []  
func (x:xs) | (alguna condición) = x : func xs  
           | otherwise          = func xs
```

Podemos pensar la condición como una función auxiliar `cond :: a -> Bool`.

Por lo tanto, redefiniendo `func :: (a -> Bool) -> [a] -> [a]`, podemos cubrir este esquema!

Otro problema frecuente

Prototipo de función muy común

```
func []      = []  
func (x:xs) | (alguna condición) = x : func xs  
           | otherwise          = func xs
```

Podemos pensar la condición como una función auxiliar `cond :: a -> Bool`.

Por lo tanto, redefiniendo `func :: (a -> Bool) -> [a] -> [a]`, podemos cubrir este esquema!

Esto también ya viene dado y es la función `filter`

Ejemplos

► ¿A qué reducen las siguientes expresiones?

- `filter esPar [2, 3, 5, 10]`
- `filter esVacia [[1,2,3], [], [2], [7,4], []]`
- `filter id [False,True,True,False,True]`

Consideremos las siguientes funciones

```
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs

todosMayoresQue3 [] = True
todosMayoresQue3 (x:xs) = (x > 3) && todosMayoresQue3 xs
```

Consideremos las siguientes funciones

```
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs

todosMayoresQue3 [] = True
todosMayoresQue3 (x:xs) = (x > 3) && todosMayoresQue3 xs
```

Aquí también existe una estructura regular que podemos abstraer de la siguiente manera:

Consideremos las siguientes funciones

```
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs

todosMayoresQue3 [] = True
todosMayoresQue3 (x:xs) = (x > 3) && todosMayoresQue3 xs
```

Aquí también existe una estructura regular que podemos abstraer de la siguiente manera:

```
plegar f cb [] = cb
plegar f cb (x:xs) = f x (plegar f cb xs)
```


Consideremos las siguientes funciones

```
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs

todosMayoresQue3 [] = True
todosMayoresQue3 (x:xs) = (x > 3) && todosMayoresQue3 xs
```

Aquí también existe una estructura regular que podemos abstraer de la siguiente manera:

```
plegar f cb [] = cb
plegar f cb (x:xs) = f x (plegar f cb xs)
```

Una vez más esta función ya existe, y se llama `foldr`

Ejemplos

- ▶ ¿A qué reducen las siguientes expresiones?
 - ▶ `plegar (+) 0 [1,2,3,4,5]`
 - ▶ `plegar (*) 1 [1,2,3,4,5]`
 - ▶ `plegar ((&&).(>3)) True [1,2,3,4,5]`

Funciones como salida de otras funciones

Valores

- ▶ Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- ▶ Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

Las funciones también pueden **devolver** valores de cualquier tipo, ¿no?

```
indeciso :: Integer -> Integer -> ([Integer] -> Integer)
indeciso x y | x > y      = product
              | otherwise = sum
```

¿Qué hace lo siguiente?

```
Prelude> (indeciso 5 10) [3,3,3]
Prelude> (indeciso 15 5) [3,3,3]
```

Funciones como salida de otras funciones

Valores

- ▶ Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- ▶ Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

Las funciones también pueden **devolver** valores de cualquier tipo, ¿no?

```
indeciso :: Integer -> Integer -> ([Integer] -> Integer)
indeciso x y | x > y      = product
              | otherwise = sum
```

¿Qué hace lo siguiente?

```
Prelude> (indeciso 5 10) [3,3,3]
Prelude> (indeciso 15 5) [3,3,3]
```

Para hacer entre todos

- ▶ ¿Qué devuelve Haskell si escribimos `:t (indeciso 5 6)`?
- ▶ ¿Qué devuelve Haskell si escribimos `:t (indeciso 5)`?

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

¿Por qué pasa lo siguiente?:

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

¿Por qué pasa lo siguiente?:

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

¿Qué devuelve Haskell si escribimos `:t (max 4)`?

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

¿Por qué pasa lo siguiente?:

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

¿Qué devuelve Haskell si escribimos `:t (max 4)`?

La magia está en los paréntesis

- ▶ En Haskell, `t1 -> t2 -> t3 -> t4`
es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- ▶ De la misma manera `f a b c`
es equivalente a `((f a) b) c`

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

¿Por qué pasa lo siguiente?:

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

¿Qué devuelve Haskell si escribimos `:t (max 4)`?

La magia está en los paréntesis

- ▶ En Haskell, `t1 -> t2 -> t3 -> t4` es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- ▶ De la misma manera `f a b c` es equivalente a `((f a) b) c`
- ▶ `max :: Integer -> Integer -> Integer` es equivalente a `max :: Integer -> (Integer -> Integer)`

La magia está en los paréntesis

- ▶ En Haskell $t1 \rightarrow t2 \rightarrow t3 \rightarrow t4$
es equivalente a $t1 \rightarrow (t2 \rightarrow (t3 \rightarrow t4))$
- ▶ De la misma manera $f\ a\ b\ c$
es equivalente a $((f\ a)\ b)\ c$

Para pensar

- ▶ ¿Si $(+) :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$, que devuelve $((+) 10)$?

La magia está en los paréntesis

- ▶ En Haskell $t1 \rightarrow t2 \rightarrow t3 \rightarrow t4$
es equivalente a $t1 \rightarrow (t2 \rightarrow (t3 \rightarrow t4))$
- ▶ De la misma manera $f\ a\ b\ c$
es equivalente a $((f\ a)\ b)\ c$

Para pensar

- ▶ ¿Si $(+) :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$, que devuelve $((+) 10)$?
- ▶ ¿Qué tipo tiene la función $\text{doble} = (*) 2$?

La magia está en los paréntesis

- ▶ En Haskell `t1 -> t2 -> t3 -> t4`
es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- ▶ De la misma manera `f a b c`
es equivalente a `((f a) b) c`

Para pensar

- ▶ ¿Si `(+) :: Integer -> Integer -> Integer`, que devuelve `((+) 10)`?
- ▶ ¿Qué tipo tiene la función `doble = (*) 2`?
- ▶ ¿Qué devuelve `map ((+) 10) [1,2,3]`?

La magia está en los paréntesis

- ▶ En Haskell `t1 -> t2 -> t3 -> t4`
es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- ▶ De la misma manera `f a b c`
es equivalente a `((f a) b) c`

Para pensar

- ▶ ¿Si `(+) :: Integer -> Integer -> Integer`, que devuelve `((+) 10)`?
- ▶ ¿Qué tipo tiene la función `doble = (*) 2`?
- ▶ ¿Qué devuelve `map ((+) 10) [1,2,3]`?
- ▶ ¿Qué devuelve `map ((+) 10)`?

La magia está en los paréntesis

- ▶ En Haskell `t1 -> t2 -> t3 -> t4`
es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- ▶ De la misma manera `f a b c`
es equivalente a `((f a) b) c`

Para pensar

- ▶ ¿Si `(+) :: Integer -> Integer -> Integer`, que devuelve `((+) 10)`?
- ▶ ¿Qué tipo tiene la función `doble = (*) 2`?
- ▶ ¿Qué devuelve `map ((+) 10) [1,2,3]`?
- ▶ ¿Qué devuelve `map ((+) 10)`?
- ▶ ¿Qué devuelve `map`?

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

▶ `Prelude> (\x -> x + 10) 20`

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

- ▶ `Prelude> (\x -> x + 10) 20`
- ▶ `Prelude> (\r -> (4*pi*r**3)/3) 10`

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

- ▶ `Prelude> (\x -> x + 10) 20`
- ▶ `Prelude> (\r -> (4*pi*r**3)/3) 10`
- ▶ `Prelude> map (\x -> x + 10) [1,2,3,4]`

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

- ▶ `Prelude> (\x -> x + 10) 20`
- ▶ `Prelude> (\r -> (4*pi*r**3)/3) 10`
- ▶ `Prelude> map (\x -> x + 10) [1,2,3,4]`
- ▶ `Prelude> (\x y -> x + y) 20 30`



Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ▶ ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ▶ ¿No sería cómodo, a veces, usar funciones “anónimas”?
- ▶ La **notación lambda** nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

- ▶ `Prelude> (\x -> x + 10) 20`
- ▶ `Prelude> (\r -> (4*pi*r**3)/3) 10`
- ▶ `Prelude> map (\x -> x + 10) [1,2,3,4]`
- ▶ `Prelude> (\x y -> x + y) 20 30`



¿Qué diferencia hay entre las siguientes definiciones? ¿Qué tipo tienen?

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

Veamos dos ejercicios del parcial resueltos con Alto Orden:

Veamos dos ejercicios del parcial resueltos con Alto Orden:

Ejercicio 4

Definimos:

```
sacarUno :: Integer -> [Integer] -> [Integer]
sacarUno n xs = filter (/= n) xs
```

```
sacarTodos :: [Integer] -> [Integer] -> [Integer]
sacarTodos xs ys = foldr sacarUno xs ys
```

Ejercicio 5

Definimos:

```
listaDeNotas :: [(Integer, Float)] -> Integer -> [Float]
listaDeNotas xs n = map snd (filter (\p->fst p == n) xs)

promedioDe :: [(Integer, Float)]->Integer->Float
promedioDe xs n = foldr (+) 0 notas / fromIntegral (length notas)
                  where notas = listaDeNotas xs n
```



- ▶ Para más detalles sobre Haskell, pueden chusmear: <http://learnyouahaskell.com/>
- ▶ Ante cada nuevo lenguaje, pregúntense qué propiedades de las vistas en el curso posee.