Introducción a funciones de alto orden

Taller de Álgebra I

Primer cuatrimestre 2019

Mapa del curso

- Introducción a la computación (Problema $\rightarrow$ Algoritmo $\rightarrow$ Programa)
- Introducción a Haskell
- Tipado
- Reducción y órdenes de evaluación
- Recursión sobre números
- Recursión sobre listas
- Pattern matching
- Generalización de la recursión
- Recursión sobre conjuntos
- Combinatoria
- Algoritmo de Euclides y ecuaciones diofánticas
- Polinomos
- Mini introducción a alto orden $\leftarrow$ (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? $(f \circ g)(x) = f(g(x))$

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? $(f \circ g)(x) = f(g(x))$
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? $(f \circ g)(x) = f(g(x))$
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? $(f \circ g)(x) = f(g(x))$
Sí! para eso usamos la función (.)
¿Qué devuelve esto?

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

6

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? $(f \circ g)(x) = f(g(x))$
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 -> CodoiminioF
(.) f g x = f (g x)
```

Funciones como parámetro

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

```
(.) :: <FuncionF> -> <FuncionG> -> DominioG -> CodoiminioF
(.) 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 -> CodoiminioF
(.) 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?
- ¿Qué devuelven 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`?
- ¿Qué devuelven 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 es bien conocida: map.
Muy útil: disponible en tu lenguaje de programación favorito.

Con polinomios...

- ¿Qué hace la siguiente función? `f poli = map neg poli` sabiendo que `neg x = -x`

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** parámetros 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** parámetros 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 `indeciso 5 6`?
- ¿Qué devuelve Haskell si escribimos `indeciso 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** parámetros 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 `indeciso 5 6`?
- ¿Qué devuelve Haskell si escribimos `indeciso 5`? ¡Opa! ¿No hay error?

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 `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 `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 `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)`

Aplicación parcial: El secreto de las flechas

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)`?

Aplicación parcial: El secreto de las flechas

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`?

Aplicación parcial: El secreto de las flechas

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]?

Aplicación parcial: El secreto de las flechas

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)`?

Aplicación parcial: El secreto de las flechas

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> (\rad -> (4*pi*rad**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> (\rad -> (4*pi*rad**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> (\rad -> (4*pi*rad**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> (\rad -> (4*pi*rad**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
```

The End

Es todo lo que tenemos para contarles en este curso. Esperamos que les haya gustado.

- Ahora tienen una herramienta importantísima para el resto de sus vidas: saben programar!
- Para más detalles sobre Haskell, pueden chusmear: `http://learnyouahaskell.com/`
- Van a conocer muchos lenguajes, pregúntense qué propiedades de las que vimos poseen.