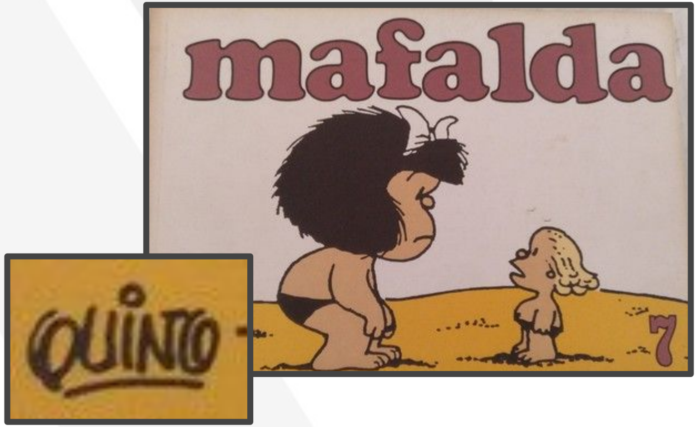
Programación Funcional

Clases teóricas

por Pablo E. “Fidel” Martínez López

3. Currificación


¿VOS SABÍAS QUE MI MAMÁ ES TRADUCTORA DE FRANCÉS, MANOLITO? YO TAMBIÉN SÉ FRANCÉS; SÉ DECIR "PAPÁ" EN FRANCÉS
¿SÍÍ? ¿CÓMO SE DICE, A VER?
PAPÁ
¡AH, ES FÁCIL! ¡SE DICE IGUAL!
¿FÁCIL? ¿IGUAL? ¡NADA DE ESO, EL ASUNTO ES PENSAR EN FRANCÉS! ¡TRATÁ DE DECIR "PAPÁ" PENSÁNDOLO EN FRANCÉS! ¡DALE! ¿A VER? ¡DALE!
¡ES INÚTIL! ¡JAMÁS PODRÉ HABLAR ESE MALDITO IDIOMA!


mafalda
QUINO
7

Descubrir la currificación

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma'  :: ...
suma'  (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

  - ¿Qué tipo tienen las funciones?
  - ¿Qué similitudes hay entre ellas?
  - ¿Y qué diferencias?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (   ,   ) ->
suma' (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (   ,   ) -> Int
suma' (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

Una función que toma un par de enteros y devuelve un entero

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: ...
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma ::       ->
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma ::       -> (     ->     )
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma ::       -> (     -> Int)
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma ::       -> (Int -> Int)
suma x = f where f y = x+y
```

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: Int -> (Int -> Int)
suma x = f where f y = x+y
```

  - ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: Int -> (Int -> Int)
suma x = f where f y = x+y
```

Una función que toma un entero y devuelve *una función de enteros en enteros*

- ¿Qué tipo tienen las funciones?

Aplicación de funciones de orden superior

- Considerar las siguientes definiciones

```
suma' :: (Int,Int) -> Int
suma' (x,y) = x+y

suma :: Int -> (Int -> Int)
suma x = f where f y = x+y
```

Una función que toma un par de enteros y devuelve un entero

Una función que toma un entero y devuelve *una función de enteros en enteros*

  - ¿Qué tipo tienen las funciones?
  - ¿Qué similitudes hay entre ellas?
  - ¿Y qué diferencias?

Aplicación de funciones de orden superior

- Similitudes
  - Ambas expresan la suma de dos enteros
  - ¡Pero no de la misma manera!

Aplicación de funciones de orden superior

- Similitudes
  - Ambas expresan la suma de dos enteros
  - ¡Pero no de la misma manera!
    - para todos **x** e **y**, `suma' (x,y) = (suma x) y`

Aplicación de funciones de orden superior

- Similitudes
  - Ambas expresan la suma de dos enteros
  - ¡Pero no de la misma manera!
    - para todos **x** e **y**, `suma' (x,y) = (suma x) y`
- Diferencias
  - Una toma un par, la otra toma un número
  - Una retorna un número, la otra retorna *una función*

Aplicación de funciones de orden superior

- Similitudes
  - Ambas expresan la suma de dos enteros
  - ¡Pero no de la misma manera!
    - para todos **x** e **y**, `suma' (x,y) = (suma x) y`
- Diferencias
  - Una toma un par, la otra toma un número
  - Una retorna un número, la otra retorna *una función*
  - La segunda forma es más expresiva
    - `succF = suma 1` vs. `succ x = suma' (1,x)`

Aplicación de funciones de orden superior

- Similitudes y diferencias, otra forma de verlo (propuesta por Cristian Sottile)
  - Ambas expresan la suma de dos enteros
  - ¡Pero no de la misma manera!
    - `suma' (1,3) :: Int      -- Ambas expresan sumas`
    - `(suma 1) 3   :: Int      -- Ambas expresan sumas`
  - Pero una es más *expresiva* que la otra
    - `suma 1 :: Int->Int      -- Una puede expresar funciones…`
    - no existe **e**.

```
suma' e :: Int->Int -- …que la otra no
```

Currificación

- Definición de currificación (*currying*)
  - ***Correspondencia*** *uno a uno* entre *cada función* que toma una tupla como argumento *con una función* que retorna una función intermedia que completa el trabajo
  - Pensando en una definición genérica

    cada definición de la forma de **f’** se corresponde con una de la forma de **f**

```
f' :: (A,B) -> C
f' (x,y) = e
```

```
f :: A -> (B -> C)
f x = g where g y = e
```

Currificación

Definición de currificación

Correspondencia *uno a uno* entre *dos conjuntos de funciones*

(Por cada **f'** hay una **f**, y viceversa)

```
f' :: (A,B) -> C          f :: A -> (B -> C)
f' (x,y) = e              f x = g where g y = e
```

- Ejemplos de la *correspondencia* que es la currificación

```
swap :: (a,b) -> (b,a)        riap :: a -> (b -> (b,a))
swap (x,y) = (y,x)            riap x = h

                                  where h y = (y,x)
```

Currificación

Definición de currificación

Correspondencia *uno a uno* entre *dos conjuntos de funciones*

(Por cada **f'** hay una **f**, y viceversa)

```
f' :: (A,B) -> C          f :: A -> (B -> C)
f' (x,y) = e              f x = g where g y = e
```

- Ejemplos de la *correspondencia* que es la currificación

```
swap :: (a,b) -> (b,a)        riap :: a -> (b -> (b,a))
swap (x,y) = (y,x)            riap x = h
                                  where h y = (y,x)

fst :: (a,b) -> a             const :: a -> (b -> a)
fst (x,y) = x                 const x = f
                                  where f y = x
```

Currificación

Definición de currificación

Correspondencia *uno a uno* entre *dos conjuntos de funciones*

(Por cada **f’** hay una **f**, y viceversa)

```
f’ :: (A,B) -> C          f :: A -> (B -> C)
f’ (x,y) = e              f x = g where g y = e
```

- Ejemplos de la *correspondencia* que es la currificación

```
f’              A  B          C
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
f’       x  y      e
```

```
f           A        B        C
riap :: a -> (b -> (b,a))
riap x = h
f    x   g
     where h y = (y,x)
           g y     e
```

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
const :: a -> (b -> a)
const x = f

    where f y = x
```

Currificación

Definición de currificación

Correspondencia *uno a uno* entre *dos conjuntos de funciones*

(Por cada **f'** hay una **f**, y viceversa)

```
f' :: (A,B) -> C          f :: A -> (B -> C)
f' (x,y) = e              f x = g where g y = e
```

- Ejemplos de la *correspondencia* que es la currificación

```
f'          A B        C
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
f'       x y     e
```

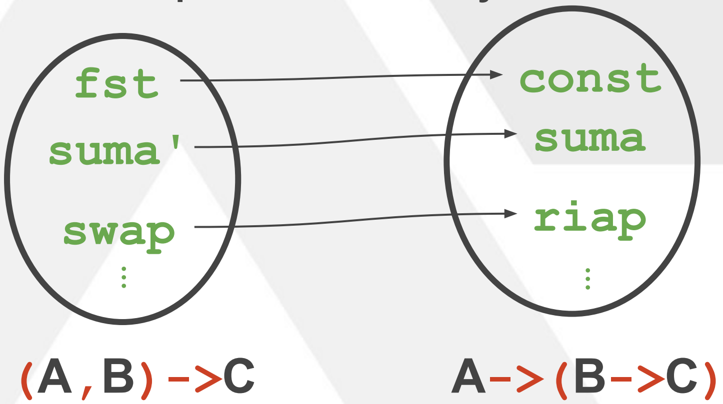
```
f          A        B       C
riap :: a -> (b -> (b,a))
riap x = h
f    x   g
     where h y = (y,x)
           g y    e
```

```
f'        A B      C
fst :: (a,b) -> a
fst (x,y) = x
f'     x y    e
```

```
f           A       B      C
const :: a -> (b -> a)
const x = f
f     x   g
      where f y = x
            g y   e
```

Currificación

- Definición de currificación (*currying*)
  - ***Correspondencia*** *uno a uno* entre *cada función* que toma una tupla como argumento *con una función* que retorna una función intermedia que completa el trabajo
  - Gráficamente



Currificación

- Sobre el nombre “currificación” (*currying*)
  - Es en honor a Haskell B. Curry
  - Aunque la idea la propuso Moses Schönfinkel
    - Pero schönfinkelización suena más complicado…
      - (y ni te cuento *schönfinkeling* para un inglés)
    - ...aunque hay quién lo sigue proponiendo

Haskell B. Curry

Haskell Brooks Curry
(12 de septiembre 1900 – 1 de septiembre 1982) fue un lógico y matemático estadounidense graduado de la Universidad de Harvard que alcanzó la fama por su trabajo en ***lógica combinatoria***. A pesar de que su trabajo se basó principalmente en un único artículo de Moses Schönfinkel, fue Curry el que hizo el desarrollo más importante.
También se lo conoce por la paradoja de Curry y por la *Correspondencia de Curry-Howard* (un vínculo profundo entre la lógica y la computación). Hay 3 lenguajes de programación nombrados en su honor, **Haskell**, **Brook** and **Curry**, así como el concepto de *currificación*, una técnica para aplicar funciones de orden superior.

Currificación

- Existe una correspondencia entre conjuntos
  - ¿Se podrán definir funciones entre ellos?
  - ¡Para eso armamos nuestro lenguaje funcional!

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

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

  - Con los elementos que veremos en esta clase podrán definirlas ustedes mismos

Currificación

- Funciones de currificación

```
curry   :: ((a,b)->c) -> (a->(b->c))
uncurry :: (a->(b->c)) -> ((a,b)->c)
```

  - Se puede demostrar que

    `curry (uncurry f)` = f para todo **f**

    `uncurry (curry f')` = f' para todo **f'**

  - ¡Estamos expresando ideas complejas de programación con las herramientas que estudiamos!

Currificación

- Funciones de currificación

```
curry   :: ((a,b)->c) -> (a->(b->c))
uncurry :: (a->(b->c)) -> ((a,b)->c)
```

- De la función de tipo `a->(b->c)` se dice que está ***currificada***
  - En inglés, *curried*
  - f' se *currifica* mediante `curry`, `(curry f') :: a->(b->c)`
- De la función de tipo `(a,b)->c` se dice que está ***descurrificada***
  - En inglés, *uncurried* (en castellano, también ***no currificada***)
  - f se *descurrrifica* con `uncurry`, `(uncurry f) :: (a,b)->c`

Currificación - Sintaxis

- ¿Cómo podemos escribir una función currificada?
  - Hasta ahora, escribimos

    ```
    twice f = g   where g x = f (f x)
    ```

  - Pero esto es incómodo…
    - …como se puede comprobar con un ejemplo

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: ...

suma5 ...
```

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 ::      ->
suma5 ...
```

Debe tomar un argumento solo y devolver una función…

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(    -> ... )
suma5 x = ...
```

Debe tomar un argumento solo y devolver una función…

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(    -> ... )
suma5 x = ...
```

¿Qué nombre ponerle
a la función?

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(    -> ... )
suma5 x = sum4
  where sum4 ...
```

¿Qué nombre ponerle a la función?

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(    -> ... )
suma5 x = sum4
  where sum4 ...
```

A su vez, esta función debe tomar un argumento solo y devolver una función…

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(Int->(    -> ... ))
suma5 x = sum4
    where sum4 y = ...
```

A su vez, esta función debe tomar un argumento solo y devolver una función…

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(Int-> ... )
suma5 x = sum4
    where sum4 y = sum3
      where sum3 ...
```

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(Int->(Int-> ... ))
suma5 x = sum4
    where sum4 y = sum3
      where sum3 z = sum2
        where sum2 ...
```

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(Int->(Int->(Int-> ... )))
suma5 x = sum4
    where sum4 y = sum3
      where sum3 z = sum2
        where sum2 v = sum1
          where sum1 ...
```

Currificación - Sintaxis

- Considerar la función

```
suma5' :: (Int,Int,Int,Int,Int) -> Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Cómo sería la versión currificada?

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
    where sum4 y = sum3
      where sum3 z = sum2
        where sum2 v = sum1
          where sum1 w = x+y+z+v+w
```

Currificación - Sintaxis

- Considerar la función

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
    where sum4 y = sum3
      where sum3 z = sum2
        where sum2 v = sum1
          where sum1 w = x+y+z+v+w
```

- ¡Es horrible!
- ¿Podemos aprovechar el poder de la denotación?

Curriﬁcación - Sintaxis

- Analicemos las siguientes deﬁniciones

`doble x = x+x` vs. `dobleF = \x -> x+x`

Currificación - Sintaxis

- Analicemos las siguientes definiciones

  `doble x = x+x` vs. `dobleF = \x -> x+x`

- Se vé que `doble = \x->x+x` o sea `doble = dobleF`

Currificación - Sintaxis

- Analicemos las siguientes definiciones

  `doble x = x+x` vs. `dobleF = \x -> x+x`

- Se vé que `doble = \x->x+x` o sea `doble = dobleF`
  - En una definición, *la aplicación a la izquierda equivale a definir una funcion con lambda a la derecha*

Currificación - Sintaxis

- Analicemos las siguientes definiciones

  `doble x = x+x` vs. `dobleF = \x -> x+x`

- Se vé que `doble = \x->x+x` o sea `doble = dobleF`
  - En una definición, *la aplicación a la izquierda equivale a definir una funcion con lambda a la derecha*
  - Abusando del vocabulario:
    un argumento a la izquierda "pasa" como parámetro a la derecha

Currificación - Sintaxis

- Analicemos las siguientes definiciones

  `doble x = x+x` vs. `dobleF = \x -> x+x`

- Se vé que `doble = \x->x+x` o sea `doble = dobleF`
  - En una definición, *la aplicación a la izquierda equivale a definir una funcion con lambda a la derecha*
  - Abusando del vocabulario:
    un argumento a la izquierda "pasa" como parámetro a la derecha

```
doble x = x+x
```

"Pasar" un argumento:

```
f x =        e
f   = \x -> e
```

Currificación - Sintaxis

- Analicemos las siguientes definiciones

  `doble x = x+x` vs. `dobleF = \x -> x+x`

- Se vé que `doble = \x->x+x` o sea `doble = dobleF`
  - En una definición, *la aplicación a la izquierda equivale a definir una funcion con lambda a la derecha*
  - Abusando del vocabulario:
    un argumento a la izquierda “pasa” como parámetro a la derecha

```
doble = \x->x+x
```

“Pasar” un argumento:

```
f x =       e
f   = \x -> e
```

Currificación - Sintaxis

- ¿Cómo podemos escribir una función currificada?
  - Veamos con el ejemplo de twice

```
twice f = g where g x = f (f x)
```

"Pasar" un argumento:

```
f x =        e

f   = \x -> e
```

Currificación - Sintaxis

- ¿Cómo podemos escribir una función currificada?
  - Veamos con el ejemplo de twice

```
twice f = g where g x = f (f x)
```

  - “Pasando” la **x** de **g**, podemos definir

```
twice f = g where g = \x -> f (f x)
```

o sea

```
twice f = \x -> f (f x)
```

“Pasar” un argumento:

```
f x =         e
f   = \x -> e
```

Currificación - Sintaxis

- ¿Cómo podemos escribir una función currificada?
  - Veamos con el ejemplo de twice

```
twice f = g where g x = f (f x)
```

  - “Pasando” la **x** de **g**, podemos definir

```
twice f = g where g = \x -> f (f x)
```

o sea

```
twice f = \x -> f (f x)
```

  - Y “pasando” la **x** para el otro lado

```
(twice f) x = f (f x)
```

“Pasar” un argumento:

```
f x =         e

f   = \x -> e
```

Currificación - Sintaxis

- ¿Cómo podemos escribir una función currificada?
- Formas equivalentes de definir a `twice`
  - `twice f = g where g x = f (f x)`
  - `twice f = \x -> f (f x)`
  - `twice = \f -> (\x -> f (f x))`
  - `(twice f) x = f (f x)`
- ¡¡Pero solo UNA de ellas!! ¿Cuál elegir?
- (En Haskell, una de estas formas puede traer problemas porque el sistema de tipos extendido no puede darle tipo)

Currificación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = sum1
        where sum1 w = x+y+z+v+w
```

Currificación - Sintaxis

- Volvamos a **suma5** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = sum1
        where sum1 w = x+y+z+v+w
```

"Pasar" un argumento:

```
f x =       e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = sum1
        where sum1 = \w->x+y+z+v+w
```

"Pasar" un argumento:

```
f x =        e

f   = \x -> e
```

Curificación - Sintaxis

- Volvamos a **suma5** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = sum1
        where sum1 = \w->x+y+z+v+w
```

Reemplazo de iguales (**sum1**)

Currificación - Sintaxis

- Volvamos a **`suma5`** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = \w->x+y+z+v+w
```

Reemplazo de iguales (`sum1`)

Currificación - Sintaxis

- Volvamos a **`suma5`** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 v = \w->x+y+z+v+w
```

“Pasar” un argumento:

```
f x = e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 = \v->(\w->x+y+z+v+w)
```

"Pasar" un argumento:

```
f x =         e

f   = \x -> e
```

Curríficación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = sum2
      where sum2 = \v->(\w->x+y+z+v+w)
```

Reemplazo de iguales (**sum2**)

Currificación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = \v->(\w->x+y+z+v+w)
```

Reemplazo de iguales
(**sum2**)

Currificación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 z = \v->(\w->x+y+z+v+w)
```

“Pasar” un argumento:

```
f x =       e
f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 = \z->(\v->(\w->x+y+z+v+w))
```

“Pasar” un argumento:

```
f x =       e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = sum3
    where sum3 = \z->(\v->(\w->x+y+z+v+w))
```

Reemplazo de iguales (**sum3**)

Currificación - Sintaxis

- Volvamos a **suma5** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = \z->(\v->(\w->x+y+z+v+w))
```

Reemplazo de iguales (**sum3**)

Currificación - Sintaxis

- Volvamos a **`suma5`** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 y = \z->(\v->(\w->x+y+z+v+w))
```

"Pasar" un argumento:

```
f x =        e
f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

“Pasar” un argumento:

```
f x =          e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = sum4
  where sum4 = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

Reemplazo de iguales (**`sum4`**)

Currificación - Sintaxis

- Volvamos a **`suma5`** y "pasemos" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

Reemplazo de iguales
(**`sum4`**)

Currificación - Sintaxis

- Volvamos a **`suma5`** y “pasemos” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

- ¡La función es la misma!
- Podríamos “pasar” la x para el otro lado…

```
suma5 = \x->(\y->(\z->(\v->(\w->x+y+z+v+w))))
```

  - …¡o podríamos “pasar” los argumentos a la izquierda!

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos “pasando” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 = \x->(\y->(\z->(\v->(\w->x+y+z+v+w))))
```

“Pasar” un argumento:

```
f x =       e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos “pasando” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

“Pasar” un argumento:

```
f x =        e
f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
suma5 x = \y->(\z->(\v->(\w->x+y+z+v+w)))
```

"Pasar" un argumento:

```
f x =        e

f   = \x  -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
(suma5 x) y = \z->(\v->(\w->x+y+z+v+w))
```

"Pasar" un argumento:

```
f x =          e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos “pasando” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
(suma5 x) y = \z->(\v->(\w->x+y+z+v+w))
```

“Pasar” un argumento:

```
f x =         e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((suma5 x) y) z = \v->(\w->x+y+z+v+w)
```

"Pasar" un argumento:

```
f x =        e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((suma5 x) y) z = \v->(\w->x+y+z+v+w)
```

"Pasar" un argumento:

```
f x =       e
f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
(((suma5 x) y) z) v = \w->x+y+z+v+w
```

"Pasar" un argumento:

```
f x =        e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **`suma5`** y sigamos "pasando" los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
(((suma5 x) y) z) v = \w->x+y+z+v+w
```

"Pasar" un argumento:

```
f x =        e

f   = \x -> e
```

Currificación - Sintaxis

- Volvamos a **suma5** y sigamos “pasando” los argumentos

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

“Pasar” un argumento:

```
f x =        e

f   = \x -> e
```

Curriﬁcación - Sintaxis

- Una definición equivalente de **suma5**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` *es una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **suma5**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

suma5 es una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **suma5**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

suma5 es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` *es una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve una función*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función *que toma un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee? Otra vez…
- …señalando el tipo

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve un entero*

Currificación - Sintaxis

- Una definición equivalente de **`suma5`**

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¿Cómo se lee?
- ¿No es molesto?

  ¿Cómo lo mejoramos?
  - ¡Está lleno de paréntesis!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y *devuelve un entero*

Currificación - Sintaxis

- Consideremos esta expresión

  `10 - 3 - 2`

  - ¿Qué número denota? ¿`5` o `9`?
    - ¿Es `7-2`? ¿O es `10-1`?
  - Si la resta es una operación binaria, ¿por qué hay 2 de ellas en la operación?
- ¿Cómo evitamos usar tantos paréntesis?
  - ¡Convenciones de notación!

Currificación - Sintaxis

- Convenciones de notación
  - La resta es *"asociativa a izquierda"*
    - O sea, `10 - 3 - 2 = (10 - 3) - 2`
      - `10 - 3 - 2 ≠ 10 - (3 - 2)`
    - Si hay 2 símbolos seguidos iguales, se resuelve *primero el de la izquierda*
  - ¿Y *"asociativa a derecha"*?
  - ¿Y *"no asociativa"*?

Currificación - Sintaxis

- Convenciones de notación
  - La resta es *“asociativa a izquierda”*
    - O sea, `10 - 3 - 2 = (10 - 3) - 2`
      - `10 - 3 - 2 ≠ 10 - (3 - 2)`
    - Si hay 2 símbolos seguidos iguales,
      se resuelve *primero el de la izquierda*
  - ¿Y *“asociativa a derecha”*?
  - ¿Y *“no asociativa”*?
    - ¡No puede haber dos símbolos seguidos sin paréntesis!

Currificación - Sintaxis

- ¿Qué pasa con la aplicación de funciones?
  - Definimos que la aplicación es *“asociativa a izquierda”*
    - O sea, `(f x) y = f x y`
      - Recordar que el espacio es un símbolo…
    - ¡Los paréntesis a la izquierda no son necesarios!
    - Pero, **f** sigue siendo una función que toma **x** y devuelve una función que toma **y** y devuelve el resultado
    - ¡No es cierto que **f** tenga DOS argumentos!

Currificación - Sintaxis

- ¿Y con el tipo de las funciones?
  - Definimos que la flecha es *"asociativa a derecha"*
    - O sea, **A -> (B -> C) = A -> B -> C**
    - ¡Los paréntesis a la derecha no son necesarios!
    - También sigue siendo una función que toma **A** y devuelve una función de **B** en **C**
  - Esta es una decisión coherente con la anterior

```
f :: A -> B -> C      es      f :: A -> (B -> C)
f x y = e                     (f x) y = e
```

Currificación - Sintaxis

- Mejoremos la definición de **`suma5`** usando asociatividad

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- Se pueden omitir
  - Los paréntesis a izquierda en las aplicaciones
  - Los paréntesis a la derecha de las funciones

Currificación - Sintaxis

- Mejoremos la definición de **`suma5`** usando asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Se pueden omitir
  - Los paréntesis a izquierda en las aplicaciones
  - Los paréntesis a la derecha de las funciones

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Se pueden omitir
  - Los paréntesis a izquierda en las aplicaciones
  - Los paréntesis a la derecha de las funciones
- ¿Y esto cambió algo en la forma de leer?

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Se pueden omitir
  - Los paréntesis a izquierda en las aplicaciones
  - Los paréntesis a la derecha de las funciones
- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` ***es una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve un entero***

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve un entero***

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` ***es una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve una función que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve una función*** que toma un entero y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función ***que toma un entero*** y devuelve un entero

Currificación - Sintaxis

- Mejoremos la definición de usando **`suma5`** asociatividad

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¿Y esto cambió algo en la forma de leer?
  - ¡NO!
  - ¿Se observan las diferentes funciones intermedias?
  - ¡Y lo mismo en el tipo!

`suma5` es una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y devuelve una función que toma un entero y ***devuelve un entero***

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Sigue siendo incómodo
- Comparar con

```
suma5' :: (Int,Int,Int,Int,Int)->Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

`suma5'` ***es una función*** que toma cinco enteros y devuelve un entero

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Sigue siendo incómodo
- Comparar con

```
suma5' :: (Int,Int,Int,Int,Int)->Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

`suma5'` es una función ***que toma cinco enteros*** y devuelve un entero

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Sigue siendo incómodo
- Comparar con

```
suma5' :: (Int,Int,Int,Int,Int)->Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

suma5' es una función que toma cinco enteros y ***devuelve un entero***

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Comparar con

```
suma5' :: (Int,Int,Int,Int,Int)->Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Con cuál nos quedamos?

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- Comparar con la versión no currificada

```
suma5' :: (Int,Int,Int,Int,Int)->Int
suma5' (x,y,z,v,w) = x+y+z+v+w
```

- ¿Con cuál nos quedamos?
  - ¡A mitad de camino!

Currificación - Sintaxis

- Definición de **`suma5`** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

`suma5` ***es una función*** que toma cinco enteros y devuelve un entero

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

suma5 es una función ***que toma cinco enteros*** y devuelve un entero

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

suma5 es una función que toma cinco enteros y ***devuelve un entero***

Currificación - Sintaxis

- Definición de **suma5** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¡Pero no es correcto!

suma5 es una función *que toma cinco enteros* y devuelve un entero

Currificación - Sintaxis

- Definición de **`suma5`** (currificada)

```
suma5 :: Int->(Int->(Int->(Int->(Int->Int))))
((((suma5 x) y) z) v) w = x+y+z+v+w
```

- ¡Pero no es correcto!

`suma5` es una función *que toma cinco enteros* y devuelve un entero

Currificación - Sintaxis

- Definición de **`suma5`** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¡Pero no es correcto!
- Leemos *"en inglés"* a lo Libertad
  - "¡Tratá de decir **`suma5'`** pero pensar en **`suma5`**! ¡Dale, a ver, dale!"

`suma5'` es una función *que toma cinco enteros* y devuelve un entero

Currificación - Sintaxis

- Definición de **`suma5`** (currificada)

```
suma5 :: Int->Int->Int->Int->Int->Int
suma5 x y z v w = x+y+z+v+w
```

- ¡Pero no es correcto!

  > **`suma5`** es una función *que toma cinco enteros* y devuelve un entero

- Leemos *"en inglés"* a lo Libertad
  - "¡Tratá de decir **`suma5'`** pero pensar en **`suma5`**! ¡Dale, a ver, dale!"
- ¿Cuántos argumentos toma la función realmente?

Currificación - Ventajas

- Ventajas de la currificación
  - Mayor expresividad
  - Aplicación parcial
  - Modularidad
    - Para expresar mejor las ideas
    - Para tratamiento de código
      - Al inferir tipos
      - Al transformar programas

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive ...
```

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la "derivada" de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

    - ¿Cuántas funciones se definieron?

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

    - ¿Cuántas funciones se definieron? ¡DOS!
      - Una función que dado **`f`**, devuelve su *función* derivada
      - Una función que dado **`f`** y un punto **`x`**,
        devuelve la derivada de **`f`** en ese punto (un *número*)

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

Estos paréntesis pueden ponerse para destacar la idea de que el resultado es una función…

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)-> Float -> Float
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

…o sacarse, para mostrar que el resultado final es un número

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la “derivada” de Análisis Matemático
    - En realidad, una aproximación algebraica…

```
derive :: (Float->Float)->(Float->Float)
derive f x = (f(x+h)-f x)/h where h = 0.0001
```

¡Estos paréntesis, en cambio, son necesarios!

Currificación - Ventajas

- Ventajas: *Mayor expresividad*
  - Considerar la función “derivada” de Análisis Matemático
    - ¿Cuál sería la versión no currificada?

```
derive' :: (Float->Float,Float) -> Float
derive' ...
```

    - ¿Cuántas funciones se definirían?

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

  - ¿Cómo sería la derivada n-ésima? (Usar **n** veces `derive`)

```
deriveN :: Int->(Float->Float)->(Float->Float)
deriveN ...
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

  - ¿Cómo sería la derivada n-ésima? (Usar **n** veces `derive`)

```
deriveN :: Int->(Float->Float)->(Float->Float)
deriveN 0 f = ...
deriveN n f = ...
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

  - ¿Cómo sería la derivada n-ésima? (Usar **n** veces **`derive`**)

```
deriveN :: Int->(Float->Float)->(Float->Float)
deriveN 0 f = f
deriveN n f = ...
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

  - ¿Cómo sería la derivada n-ésima? (Usar **n** veces `derive`)

```
deriveN :: Int->(Float->Float)->(Float->Float)
deriveN 0 f = f
deriveN n f = deriveN (n-1) (derive f)
```

Aplicación parcial de `derive`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La función derive se puede aplicar a un solo argumento

```
derive f = \x->(f(x+h)-f x)/h where h = 0.0001
```

  - ¿Cómo sería la derivada n-ésima? (Usar **n** veces `derive`)

```
deriveN :: Int->(Float->Float)->Float->Float
deriveN 0 f = f
deriveN n f = deriveN (n-1) (derive f)
```

Aplicación parcial de
`derive`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo sería la derivada n-ésima con **`derive'`**?

```
deriveN' :: (Int,Float->Float,Float) -> Float
deriveN' ...
```

  - ¡Es mucho más fácil con aplicación parcial!
  - Y con aplicación parcial, se puede generalizar…

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - Escribir una función que aplique otra muchas veces

```
many :: Int -> (a->a) -> (a->a)
many ...
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - Escribir una función que aplique otra muchas veces

```
many :: Int -> (a->a) -> (a->a)
many 0 f x = ...
many n f x = ...
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - Escribir una función que aplique otra muchas veces

```
many :: Int -> (a->a) -> (a->a)
many 0 f x = x
many n f x = ...
```

Curriﬁcación - Ventajas

- Ventajas: *Aplicación parcial*
  - Escribir una función que aplique otra muchas veces

```
many :: Int -> (a->a) -> (a->a)
many 0 f x = x
many n f x = f (many (n-1) f x)
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - Escribir una función que aplique otra muchas veces

```
many :: Int -> (a->a) -> (a->a)
many 0 f x = x
many n f x = f (many (n-1) f x)
```

  - Se pueden expresar ideas que ya vimos

`twice = many 2`　　　　`many 1 = apply`

para todo **n**, `deriveN` **n** = `many n derive`

    - También se podrían usar para definir estas ideas…

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

  - ¿Cuántos argumentos lleva **`twice`**?

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

Ninguno

  - ¿Cuántos argumentos lleva **`twice`**?

`twice`

Curríficación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

Uno

  - ¿Cuántos argumentos lleva **`twice`**?

`twice`

`twice doble`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

Dos

  - ¿Cuántos argumentos lleva **`twice`**?

`twice` `twice doble` `twice doble 2`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

Tres

  - ¿Cuántos argumentos lleva **`twice`**?

`twice`

`twice doble`

`twice doble 2`

`twice twice doble 2`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

Cuatro

- ¿Cuántos argumentos lleva **`twice`**?

`twice`

`twice doble`

`twice doble 2`

`twice twice doble 2`

`twice twice twice doble 2`

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - La aplicación parcial permite cosas que parecen raras pero son poderosas

```
twice :: (a->a) -> a -> a
twice f x = f (f x)
```

¿ Ninguno Uno Dos Tres Cuatro ?

  - ¿Cuántos argumentos lleva **`twice`**?

`twice` `twice doble` `twice doble 2`

`twice twice doble 2` `twice twice twice doble 2`

  - ¡Todas estas expresiones tienen sentido!

Curríficación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica `twice twice doble 2`?

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica `(twice twice) doble 2`?

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica `(twice twice) doble 2`?
    - Recordemos el tipado…

```
twice ::      (a'->a') -> a' -> a'
twice :: (a->a) -> (a->a)
```

```
twice twice ::
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica `(twice twice) doble 2`?
    - Recordemos el tipado…

```
twice ::       (a'->a') -> a' -> a'
twice :: (a->a) -> (a->a)
_____________________________________ a' <- a->a
twice twice :: (a->a) -> a -> a
```

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica **`(twice twice) doble 2`**?
    - Recordemos el tipado…

```
twice ::       (a'->a') -> a' -> a'
twice :: (a->a) -> (a->a)
------------------------------------  a' <- a->a
twice twice :: (a->a) -> a -> a
```

  - ¡El **`twice`** de la izquierda es de 3er orden!

$$\texttt{twice :: (}\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{->}\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{) -> }\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{ -> }\underbrace{\texttt{(a->a)}}_{\texttt{a'}}$$

Currificación - Ventajas

- Ventajas: *Aplicación parcial*
  - ¿Cómo se explica **`(twice twice) doble 2`**?
    - Recordemos el tipado…

```
twice ::       (a'->a') -> a' -> a'
twice :: (a->a) -> (a->a)
---------------------------------------  a' <- a->a
twice twice :: (a->a) -> a -> a
```

  - ¡El **`twice`** de la izquierda es de 3er orden!

$$\texttt{twice :: (}\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{->}\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{) -> }\underbrace{\texttt{(a->a)}}_{\texttt{a'}}\texttt{ -> }\underbrace{\texttt{a -> a}}_{\texttt{a'}}$$

Funciones “currificadas”

- ¿Reconocemos una función currificada solo por su tipo?

Funciones “currificadas”

- ¿Reconocemos una función currificada solo por su tipo?
  - NO. Considerar la función “distancia al origen”

```
distancia :: (Float, Float) -> Float
distancia ...
```

Funciones “currificadas”

- ¿Reconocemos una función currificada solo por su tipo?
  - NO. Considerar la función “distancia al origen”

```
distancia :: (Float, Float) -> Float
distancia (x,y) = sqrt (x^2 + y^2)
```

Funciones "currificadas"

- ¿Reconocemos una función currificada solo por su tipo?
  - NO. Considerar

```
distancia :: (Float, Float) -> Float
distancia (x,y) = sqrt (x^2 + y^2)
```

  - ¿Tiene sentido separar el único argumento de **`distancia`**?
  - Pero no devuelve una función… ¿y entonces?

Funciones “currificadas”

- ¿Reconocemos una función currificada solo por su tipo?
  - NO. Considerar

```
distancia :: (Float, Float) -> Float
distancia (x,y) = sqrt (x^2 + y^2)
```

  - ¿Tiene sentido separar el único argumento de **`distancia`**?
  - Pero no devuelve una función… ¿y entonces?
- La currificación puede ser *cuestión de interpretación*

Funciones “currificadas”

- La currificación puede ser *cuestión de interpretación*
- ¿Y tiene sentido saber si algo está currificado?
  - Considerar

```
suma5'' :: (Int,Int)->Int->(Int,Int)->Int
suma5'' (x,y) z (v,w) = x+y+z+v+w
```

  - ¿Podemos decir que está “*un poquito currificada*”?
  - Mhh…

Resumen

Resumen

- Currificación
  - Aplicación de funciones de orden superior
  - Uso de Haskell para expresar ideas complejas
  - Importancia de la notación
  - Ventajas
    - Mucha mayor expresividad
    - Aplicación parcial
    - Modularidad