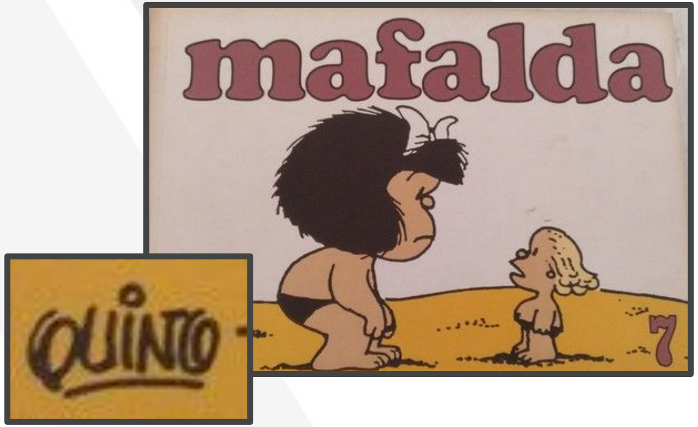
Programación Funcional

Clases teóricas

por Pablo E. “Fidel” Martínez López

4. Orden superior


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


mafalda
QUINO
7

Repaso

Expresiones y valores

- En Programación Funcional solo hay **expresiones**
  - Describen **valores**
    - Datos básicos (números, booleanos, strings, estructuras)
    - Funciones (transforman información)
  - El **tipo** de una expresión establece su naturaleza
    - A qué conjunto pertenece
    - Qué operaciones se pueden realizar
    - Se usa **polimorfismo** para expresiones que pueden ser tipadas de infinitas formas diferentes

Definiciones

- Las **definiciones** vinculan nombres con expresiones
  - Se dan en forma de **ecuaciones**
    - Del lado izquierdo el nombre a definir
      del derecho una expresión cuyo valor se conoce
    - Si son funciones, del lado izquierdo pueden tener parámetros a los que se aplica ese nombre
  - El **tipo** de una definición se puede inferir
    - A través de combinar los tipos de las partes de la expresión
    - Instanciando variables de tipo si hace falta
    - Si la inferencia **falla**, no se puede asignar un tipo, y da error

Tipos algebraicos

- Los tipos algebraicos son tipos nuevos
  - Se definen a través de ***constructores***
    - Los constructores pueden llevar argumentos
    - Cada constructor expresa a un grupo de elementos
  - Se acceden mediante ***pattern matching***
    - Los constructores se usan para preguntar

Tipos algebraicos

- Los tipos algebraicos se clasifican en
  - enumerativos
    - varios constructores sin argumentos (e.g. `Direccion`)
  - registros o productos
    - un único constructor con varios argumentos (e.g. `Persona`)
  - sumas o variantes
    - varios constructores con argumentos (e.g. `Helado`)
  - recursivos
    - suma que usa el mismo tipo como argumento (e.g. `Listas`)

Recursión estructural

- Tipos algebraicos recursivos
  - Sumas con algún constructor con el mismo tipo como argumento
  - Se utilizan *las mismas técnicas* que con otros tipos algebraicos
  - Recursión estructural
    - Una ecuación por cada constructor
    - La misma función en las partes recursivas
    - Solo hace falta pensar cómo agregar lo que falta

Recursión estructural

- Construcción de definiciones recursivas estructurales
  1. Se plantea la recursión
  2. Se prueba con un ejemplo
  3. Luego se resuelve el caso recursivo
     a. Se pregunta qué son los llamados recursivos
     b. Se decide qué falta agregarles a ellos para dar el resultado (quizás con funciones auxiliares)
  4. Se completa con el caso base

Funciones anónimas

Funciones anónimas: expresiones lambda

- Las funciones anónimas son funciones sin nombre
  - Expresiones que denotan directamente a una función
  - E.g. `\x -> x+x`
  - Se lee *"soy la función que al dato de entrada* `x` *lo transforma en el dato denotado por la expresión* `x+x`*"*
  - El símbolo `\` se llama "lambda" (por la letra griega **λ**)
    - De ahí que se llama "expresiones lambda" a esta forma de escribir funciones anónimas
- ¡Las funciones son valores!

Funciones anónimas: expresiones lambda

- Las expresiones lambda son expresiones especiales

“que al dato de entrada **`x`***”*

“el valor denotado por **`x+x`***”*

“soy la función”

“lo transforma en”

```
\x -> x+x
```

Parámetro de la función

Cuerpo de la función

Funciones anónimas: expresiones lambda

- Funciones en visión denotacional: **`\x->x+x`**
  - **`\`** **`x`** **`->`** **`x+x`**

    soy la función que a cada dato **`x`** *lo relaciona con el dato* **`x+x`**
- Funciones en visión operacional: **`\x->x+x`**
  - **`\`** **`x`** **`->`** **`x+x`**

    soy la función que recibe un dato **`x`** *y lo transforma en* **`x+x`**

    soy la función que *toma* **`x`** *y* *devuelve* **`x+x`**

Funciones anónimas: expresiones lambda

- Usando expresiones lambda se puede definir

  ```
  dobleF = \x -> x+x
  ```

  en lugar de usar

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

- Observar que el parámetro está en un lugar diferente
  - En una definición,

    la aplicación del lado izquierdo equivale a usar una función anónima (con un parámetro) del lado derecho

Funciones anónimas: “pasar” parámetros

- 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 del lado izquierdo equivale a usar una función anónima (con un parámetro) del lado derecho*
  - 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
```

Funciones anónimas: “pasar” parámetros

- 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 del lado izquierdo equivale a usar una función anónima (con un parámetro) del lado derecho*
  - 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
```

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

Funciones anónimas: aplicación

- Las funciones anónimas SON funciones
  - Y por lo tanto se pueden aplicar como otras
  - Comparar
    - `doble 2`
    - `dobleF 2`
    - `(\x -> x+x) 2`
  - Todas son válidas
  - ¿Cómo reduce cada una?

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

Funciones anónimas: aplicación

- Las funciones anónimas SON funciones
  - ¿Cómo reduce cada una? Veamos **doble 2**

```
              x
    doble    [2]
->           x   x                  (doble)
            [2]+[2]
->                                  (+)
             4
```

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

Funciones anónimas: aplicación

- Las funciones anónimas SON funciones
  - ¿Cómo reduce cada una? Veamos `(\x -> x+x) 2`

```
                         x
   (\x -> x+x) [2]
   ---------------
->  x   x               (regla Beta)
   [2]+[2]

->                      (+)
    4
```

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

Funciones anónimas: aplicación

- Las funciones anónimas SON funciones
  - ¿Cómo reduce cada una? Veamos **dobleF 2**

```
     dobleF 2
->                            (dobleF)
     (\x -> x+x) 2
->                            (regla Beta)
     2+2
->                            (+)
     4
```

Funciones anónimas: reducción

- La regla Beta
  - Para ***reducir*** aplicaciones de funciones anónimas
  - La aplicación se cambia por
    el cuerpo de la función modificado, donde
    el parámetro que sigue al lambda (**`\x ->...`**)
    se cambia por
    el argumento en cada uso en el cuerpo (**`...-> x+x`**)

Funciones anónimas en definiciones

```
doble x = x+x
cuadruple x = 4*x
succ x = x+1
fst (x,y) = x
swap (x,y) = (y,x)
```

- Otras funciones definidas con funciones anónimas

```
dobleF = \x -> x+x
cuadrupleF = \x -> 4*x
succF = \x -> x+1
fstF = \(x,y) -> x
swapF = \(x,y) -> (y,x)
```

  - Pueden tener un par como argumento
  - Pero con otros patterns NO ANDA BIEN

Funciones anónimas como resultados

- ¿Y qué pasa con funciones de más de un parámetro?

```
suma x y = x+y
sumaF x = \y -> x+y
```

  - ¡Las funciones pueden devolver funciones!
  - Las funciones son valores y pueden ser resultados
  - ¿Puedo usar **`sumaF x`** como expresión?

```
suma  x  y  =  x+y
sumaF  x  =  \y  ->  x+y
```

Funciones anónimas: aplicaciones

- ¿Puedo usar **`sumaF x`** como expresión? ¡Sí!
  - Considerar

    ```
    succF2 = sumaF 1
    ```

  - ¿Cómo se usaría?
  - ¿Cómo reduciría?

Funciones anónimas: aplicaciones

```
suma x y = x+y
sumaF x = \y -> x+y
succF2 = sumaF 1
```

- ¿Cómo reduciría una función que usa **`sumaF x`**?

```
     succF 16
->                  (succF)
     (sumaF 1) 16
->                  (sumaF)
     (\y -> 1+y) 16
->                  (regla Beta)
     1+16
->                  (+)
     17
```

Funciones anónimas: aplicaciones

```
suma x y = x+y
sumaF x = \y -> x+y
succF2 = sumaF 1
```

- Comparar estas dos reducciones

```
    (sumaF 1) 16
->                      (sumaF)
    (\y -> 1+y) 16
->                      (regla Beta)
    1+16
->                      (+)
    17
```

```
    suma 1 16
->                      (suma)
    1+16
->                      (+)
    17
```

Funciones anónimas: notación

```
suma x y = x+y
sumaF x = \y -> x+y
succF2 = sumaF 1
```

- ¡Ya estábamos usando funciones anónimas!
  - `suma 1 16 = (sumaF 1) 16`
  - Son dos formas equivalentes de definir lo mismo
  - `suma 1` ES una función de 1 parámetro
  - ¿Y los paréntesis?

Funciones anónimas: notación

- ¡Hay paréntesis invisibles!
  - 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 se pueden hacer invisibles!
    - ¡No es cierto que **f** tenga DOS argumentos!
    - **f** es una función que toma **x** y devuelve una función (llamada **f x**), que toma **y** y devuelve el resultado

Funciones anónimas: notación

```
suma x y = x+y
sumaF x = \y -> x+y
succF2 = sumaF 1
```

- Con paréntesis invisibles

  `(suma 1) 16 = suma 1 16`

  - ¡Los paréntesis a izquierda se pueden hacer invisibles!
  - ¡No es cierto que **`suma`** tenga DOS argumentos!
  - **`suma`** es una función que toma **`x`** y devuelve una función (llamada **`suma x`**), que toma **`y`** y devuelve el resultado
  - Por eso las definiciones de **`suma`** y **`sumaF`** son equivalentes (**`sumaF x`** es **`\y -> x+y`**, o sea, una función que toma **`y`** y devuelve el resultado)

Funciones anónimas como resultado

- La regla de “pasar” un argumento como parámetro, ¿se puede usar más de una vez?

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

  y entonces

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

- Observar que **`suma`** es una función que devuelve una función
- Convención de notación:
  varios lambda se pueden poner juntos…

Funciones anónimas: notación

- Varios lambdas se pueden poner juntos
  - Es lo mismo

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

    que

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

- Esto refuerza la *ilusión* de que `suma` tiene dos parámetros
- **RECORDAR:** `suma` tiene *solamente UN parámetro*,
  y devuelve una función

```
suma x y = x+y
sumaF x = \y -> x+y
succF2 = sumaF 1
```

Funciones anónimas: notación

- ¿Y qué pasa en los tipos con paréntesis invisibles?

  ```
  suma :: Int -> Int -> Int
  ```

  - Como **`suma`** es una función que toma un número y devuelve una función, debería ser

  ```
  suma :: Int -> (Int -> Int)
  ```

  - ¡Los paréntesis a derecha se pueden hacer invisibles!
    - La flecha asocia a derecha
  - Nuevamente, la función NO tiene DOS parámetros, sino solamente uno

Funciones anónimas: notación

- ¿Y qué pasa en los tipos con paréntesis invisibles?
  - Definimos que la flecha es *"asociativa a derecha"*
    - O sea, **A -> (B -> C) = A -> B -> C**
    - ¡Los paréntesis a la derecha se pueden hacer invisibles!
    - 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

Currificación

- Hay una relación estrecha entre
  - Cada función de “varios parámetros” que en realidad toma uno y devuelve una función, y
  - Una que toma un tupla con todos los parámetros juntos
  - Comparar

```
suma :: Int -> Int -> Int
suma x y = x+y
-- suma x = \y -> x+y
```

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

Currificación

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

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

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

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

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

Currificación

- 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`

Currificación

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

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

- 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*

Currificación

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

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

- 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. `succF' x = suma' (1,x)`

Currificación

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

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

- 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

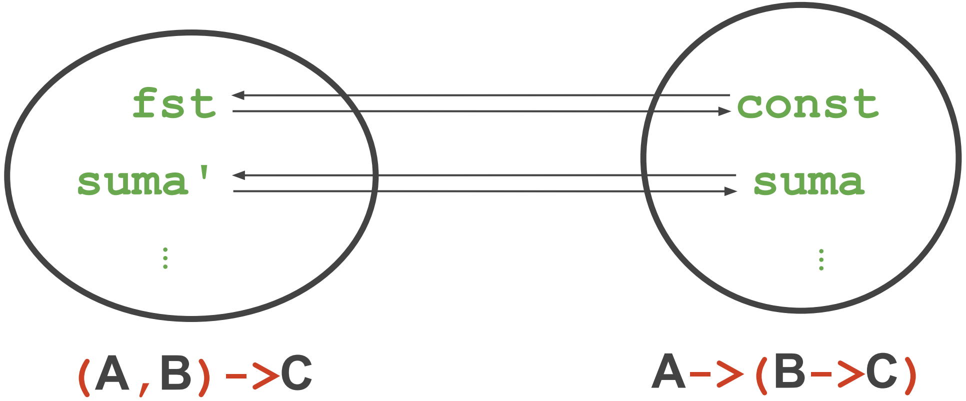
- Definición de ***currificación*** (*currying*)
  - ***Correspondencia*** *uno a uno* entre *cada función* que toma una tupla como argumento *con una función* que retorna una función intermedia que completa el trabajo
  - Pensando en una definición genérica

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

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

PERSONALIDADES: 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 ...
```

  - Las van a definir en la práctica

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

    `uncurry (curry f’) = f’` para todo **f’** `:: (a,b)->c`

  - ¡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`

Funciones de orden superior

Funciones como argumento

- ¿Si las funciones son valores,
  se pueden usar como argumento? ¡Sí!
- Ejemplo

```
pendiente f x = f (x+1) - f x
```

  - ¿Cómo saber que **`f`** es una función?

Funciones como argumento

- ¿Si las funciones son valores,
  se pueden usar como argumento? ¡Sí!
- Ejemplo

```
pendiente f x = f (x+1) - f x
```

  - ¿Cómo saber que **`f`** es una función?
  - Porque se aplica,
    se usa para modificar a **`x`** y a **`x+1`**.
  - ¿Y qué tipo tiene **`pendiente`**?

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente :: ??
pendiente f x = f (x+1) - f x
```

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente ::      ??      -> ??   -> ??
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros (“en francés”)

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente ::      ??      -> Int -> ??
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros ("en francés")
  - El 2do, **`x`**, es un número (porque está sumado)

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente ::      ??      -> Int -> ??
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros (“en francés”)
  - El 2do, **`x`**, es un número (porque está sumado)

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente ::      ??      -> Int -> Int
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros ("en francés")
  - El 2do, **`x`**, es un número (porque está sumado)
  - El resultado es un número (la resta de otros 2)

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente :: (?? -> ??) -> Int -> Int
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros ("en francés")
  - El 2do, **`x`**, es un número (porque está sumado)
  - El resultado es un número (la resta de otros 2)
  - ¡El primero, **`f`**, es una función (de 1 parámetro)!

Funciones como argumento

- ¿Qué tipo tiene tiene **`pendiente`**?

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

  - Es una función de 2 parámetros ("en francés")
  - El 2do, **`x`**, es un número (porque está sumado)
  - El resultado es un número (la resta de otros 2)
  - ¡El primero, **`f`**, es una función (de 1 parámetro)!
    - Toma un número y devuelve un número
      ¿Por qué?

Funciones de orden superior

- Hay una escalera de órdenes de funciones
  - **Orden 0:** `0`,`1`,`2`,`3`, … Datos básicos, valores que NO son funciones
  - **Orden 1:** `doble`, `succ`, … Funciones que transforman valores de orden 0
  - **Orden 2:** `pendiente`, … Funciones que transforman funciones de orden 1
  - **Orden n:** Funciones que transforman funciones de orden (n-1)
- Funciones de **orden superior**
  - Si toman cualquier función como argumento
    - Funciones de orden 2 o más
  - Si devuelven funciones de orden superior

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- ¿Cómo se usan las funciones de orden superior?
  - Igual que las demás, pero aplicándolas a funciones
  - Ejemplos:

```
pendiente doble 10 = 2
pendiente (\x->x^2) 4 = 9
pendiente (suma 2) 5 = 1
```

    - Comprobar los resultados usando reducción (así se vé cómo funcionan)

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Reducción de

```
    pendiente doble 10
->  … ->
    doble 11 - doble 10
->  … ->
    22 - 20
->
    2
```

  - Observar que `doble` es una función, y al pasar como argumento, se aplica a distintos valores

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Reducción de

```
    pendiente (\x->x^2) 4
-> … ->
    (\x->x^2) 5 - (\x->x^2) 4
-> … ->
    25 - 16
->
    9
```

  - Observar que `(\x->x^2)` es una función, y al pasar como argumento, se aplica a distintos valores

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Reducción de

```
    pendiente (suma 2) 4
->  … ->
    suma 2 5 - suma 2 4
->  … ->
    7 - 6
->
    1
```

¡ATENCIÓN a los paréntesis invisibles!

- Observar que `(suma 2)` es una función, y al pasar como argumento, se aplica a distintos valores

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Observar que el tipo de `pendiente doble` es

  ```
  pendiente doble :: Int -> Int
  ```

- ¿Y qué función es, entonces `pendiente doble`?

  ```
  pendiente doble = \n -> 2
  ```

  - ¿Cómo se comprueba?
    - Pedir un parámetro `n`, aplicarlo y usar equivalencias

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Comprobar que `pendiente doble = \n -> 2`

```
\n -> pendiente doble n
```

¡ATENCIÓN a los paréntesis invisibles!

Se pide un parámetro **`n`** y se lo aplica

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Comprobar que `pendiente doble = \n -> 2`

```
    \n -> pendiente doble n
=                                        (pendiente)
    \n -> doble (n+1) - doble n
```

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Comprobar que `pendiente doble = \n -> 2`

```
      \n -> pendiente doble n
   =                                      (pendiente)
      \n -> doble (n+1) - doble n
   =                                      (doble, 2 veces)
      \n -> (n+1)+(n+1) - (n+n)
```

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Comprobar que `pendiente doble = \n -> 2`

```
    \n -> pendiente doble n
=                                   (pendiente)
    \n -> doble (n+1) - doble n
=                                   (doble, 2 veces)
    \n -> (n+1)+(n+1) - (n+n)
=                                   (aritm.)
    \n -> 2*n + 2 - 2*n
```

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Comprobar que `pendiente doble = \n -> 2`

```
     \n -> pendiente doble n
  =                                   (pendiente)
     \n -> doble (n+1) - doble n
  =                                   (doble, 2 veces)
     \n -> (n+1)+(n+1) - (n+n)
  =                                   (aritm.)
     \n -> 2*n + 2 - 2*n
  =                                   (aritm.)
     \n -> 2
```

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior

- Sabemos, entonces que

```
pendiente doble = \n -> 2
```

  - ¡Las computadoras NO pueden calcular esto en general!
  - La relación de equivalencia = NO es computable
  - Para trabajar con este tipo de ecuaciones se precisan más herramientas que no veremos en profundidad
    - Propiedades y demostraciones

```
pendiente :: (Int->Int) -> Int -> Int
pendiente f x = f (x+1) - f x
```

Funciones de orden superior: aplicación parcial

- Si `pendiente` tiene 2 parámetros, ¿se la puede usar con 1 argumento, como en `pendiente doble`?
  - Sí. En realidad NO tiene 2 parámetros, sino solamente 1
  - Al tomar 1 argumento devuelve una función
  - Por eso `pendiente doble :: Int -> Int`
  - Se conoce con el nombre de ***aplicación parcial*** cuando a una función de "varios parámetros" se le dan menos
    - Es útil para pasar funciones como parámetro

Funciones de orden superior: aplicación parcial

- Consideremos un ejemplo

```
suma3 :: ??
suma3 x y z = x+y+z
```

  - ¿Qué tipo tiene **`suma3`**?

Funciones de orden superior: aplicación parcial

- Consideremos un ejemplo

```
suma3 :: ??  -> ??  -> ??  -> ??
suma3 x y z = x+y+z
```

  - ¿Qué tipo tiene **`suma3`**?
  - Hablando “en francés”, ¿cuántos parámetros tiene?

Funciones de orden superior: aplicación parcial

- Consideremos un ejemplo

```
suma3 :: ??  -> ??  -> ??  -> ??
suma3 x y z = x+y+z
```

  - ¿Qué tipo tiene **`suma3`**?
  - Hablando "en francés", ¿cuántos parámetros tiene?
  - ¿De qué tipo es cada parámetro? ¿Y el resultado?

Funciones de orden superior: aplicación parcial

- Consideremos un ejemplo

```
suma3 :: Int -> Int -> Int -> Int
suma3 x y z = x+y+z
```

  - ¿Qué tipo tiene **`suma3`**?
  - Hablando “en francés”, ¿cuántos parámetros tiene?
  - ¿De qué tipo es cada parámetro? ¿Y el resultado?

Funciones de orden superior: aplicación parcial

- Consideremos un ejemplo

```
suma3 :: Int -> Int -> Int -> Int
suma3 x y z = x+y+z
```

  - ¿Qué tipo tiene **`suma3`**?
  - Hablando “en francés”, ¿cuántos parámetros tiene?
  - ¿De qué tipo es cada parámetro? ¿Y el resultado?
  - ¿Es *realmente* cierto que toma 3 enteros?
    - ¡No! Toma 1 entero y devuelve una función

```
suma3 :: Int -> Int -> Int -> Int
suma3 x y z = x+y+z
```

Funciones de orden superior: aplicación parcial

- Usando **suma3** con aplicación parcial

```
suma3 2 3 4 = 9
suma3 2 3 = \z -> 5+z
suma3 2 = \y z -> 2+y+z
suma3 = \x y z -> x+y+z
```

- En cada caso toma diferente cantidad de parámetros
- ¡Pero los demás quedan en el resultado!
- ¿Qué función es `suma3 0`?

Funciones de orden superior: a

```
suma3 :: Int -> Int -> Int -> Int
suma3 x y z = x+y+z

suma :: Int -> Int -> Int
suma x y = x+y
```

- Usando **`suma3`** con aplicación parcial

```
suma3 2 3 4 = 9
suma3 2 3 = \z -> 5+z
suma3 2 = \y z -> 2+y+z
suma3 = \x y z -> x+y+z
```

- En cada caso toma diferente cantidad de parámetros
- ¡Pero los demás quedan en el resultado!
- ¿Qué función es **`suma3 0`**? ¡Es **`suma`**!

Funciones de orden superior: usos

- Aumentan la expresividad
  - Considerar la siguiente definición

```
losBajos :: [Persona] -> [Persona]
losBajos []     = []
losBajos (p:ps) = consSiCumple esBajo p (losBajos ps)

consSiCumple :: (a->Bool) -> a -> ([a] -> [a])
consSiCumple pred x = if pred x then …
                                 else …
```

  - ¿Con qué expresiones debe completarse?

Funciones de orden superior: usos

- Aumentan la expresividad
  - Considerar la siguiente definición

```
losBajos :: [Persona] -> [Persona]
losBajos []     = []
losBajos (p:ps) = consSiCumple esBajo p (losBajos ps)

                   pred         x          …
consSiCumple :: (a->Bool) -> a -> ([a] -> [a])
consSiCumple pred x = if pred x then …
                                else …
```

  - ¿Con qué expresiones debe completarse?

Funciones de orden superior

```
losBajos :: [Persona] -> [Persona]
losBajos []      = []
losBajos (p:ps) =
   consSiCumple esBajo p (losBajos ps)
```

- Aumentan la expresividad
  - Considerar la siguiente definición

```
consSiCumple :: (a->Bool) -> a -> ([a] -> [a])
consSiCumple pred x = if pred x then …
                                 else …
```

  - ¿Con qué expresiones debe completarse?
    - ¡Con funciones!

Funciones de orden superior

```
losBajos :: [Persona] -> [Persona]
losBajos []       = []
losBajos (p:ps) =
   consSiCumple esBajo p (losBajos ps)
```

- Aumentan la expresividad
  - Considerar la siguiente definición

```
consSiCumple :: (a->Bool) -> a -> ([a] -> [a])
consSiCumple pred x = if pred x then …
                                 else …
```

  - ¿Con qué expresiones debe completarse?
    - ¡Con funciones!
    - En el `then` agrega `x` usando `(:)`, en el `else` no hace nada…
    - ¿Qué función "no hace nada"?

```haskell
losBajos :: [Persona] -> [Persona]
losBajos []     = []
losBajos (p:ps) =
   consSiCumple esBajo p (losBajos ps)
```

Funciones de orden superior

- Aumentan la expresividad
  - Considerar la siguiente definición

```haskell
consSiCumple :: (a->Bool) -> a -> ([a] -> [a])
consSiCumple pred x = if pred x then \xs -> x:xs
                                else \xs -> xs
```

  - ¿Con qué expresiones debe completarse?
    - ¡Con funciones!
    - En el **`then`** agrega **`x`** usando **`(:)`**, en el **`else`** no hace nada…
    - ¿Qué función “no hace nada”?

Constructores como funciones de orden superior

Funciones de orden superior: usos

```
data Helado =
      Vasito Gusto
    | Cucurucho Gusto Gusto
    | Pote Gusto Gusto Gusto
```

- Los constructores con argumentos son funciones
  - Sabemos que la expresión

    ```
    Vasito DDL :: Helado
    ```

  - También sabemos que

    ```
    DDL :: Gusto
    ```

  - ¿Qué tipo tiene entonces **`Vasito`**?

    ```
    Vasito :: ??
    ```

```
data Helado =
    Vasito Gusto
  | Cucurucho Gusto Gusto
  | Pote Gusto Gusto Gusto
```

Funciones de orden superior: usos

- Los constructores con argumentos son funciones
  - Sabemos que la expresión

    ```
    Vasito DDL :: Helado
    ```

  - También sabemos que

    ```
    DDL :: Gusto
    ```

  - ¿Qué tipo tiene entonces **`Vasito`**?

    ```
    Vasito :: ?? -> ??
    ```

Funciones de orden superior: usos

```
data Helado =
    Vasito Gusto
  | Cucurucho Gusto Gusto
  | Pote Gusto Gusto Gusto
```

- Los constructores con argumentos son funciones
  - Sabemos que las expresiones

    ```
    Vasito DDL :: Helado
    ```

    ```
    DDL :: Gusto
    ```

  - ¿Qué tipo tiene entonces `Vasito`?
    - `DDL` es argumento de `Vasito`
    - `Vasito` ES una función

      ```
      Vasito :: Gusto -> Helado
      ```

Funciones de orden superior: usos

```
data Helado =
     Vasito Gusto
   | Cucurucho Gusto Gusto
   | Pote Gusto Gusto Gusto
```

- Los constructores con argumentos son funciones

```
Vasito :: Gusto -> Helado

Cucurucho :: ??

Pote :: ??

Cucurucho Chocolate :: ??

Pote Chocolate :: ??

Pote Chocolate DDL :: ??
```

Funciones de orden superior: usos

```
data Helado =
    Vasito Gusto
  | Cucurucho Gusto Gusto
  | Pote Gusto Gusto Gusto
```

- Los constructores con argumentos son funciones

```
Vasito :: Gusto -> Helado

Cucurucho :: ?? -> ?? -> Helado

Pote :: ?? -> ?? -> ?? -> Helado

Cucurucho Chocolate :: ?? -> Helado

Pote Chocolate :: ?? -> ?? -> Helado

Pote Chocolate DDL :: ?? -> Helado
```

Funciones de orden superior: usos

```
data Helado =
     Vasito Gusto
   | Cucurucho Gusto Gusto
   | Pote Gusto Gusto Gusto
```

- Los constructores con argumentos son funciones

```
Vasito :: Gusto -> Helado

Cucurucho :: Gusto -> Gusto -> Helado

Pote :: Gusto -> Gusto -> Gusto -> Helado

Cucurucho Chocolate :: Gusto -> Helado

Pote Chocolate :: Gusto -> Gusto -> Helado

Pote Chocolate DDL :: Gusto -> Helado
```

Funciones de orden superior: usos

```
data Pizza =
    Prepizza
  | Capa Ingrediente Pizza
```

- Los constructores con argumentos son funciones

```
Prepizza :: ??

Capa :: ??

Capa Queso :: ??
```

```
data Pizza =
    Prepizza
  | Capa Ingrediente Pizza
```

Funciones de orden superior: usos

- Los constructores con argumentos son funciones

```
Prepizza :: Pizza

Capa :: ?? -> ?? -> Pizza

Capa Queso :: ?? -> Pizza
```

```
data Pizza =
    Prepizza
  | Capa Ingrediente Pizza
```

Funciones de orden superior: usos

- Los constructores con argumentos son funciones

```
Prepizza :: Pizza

Capa :: Ingrediente -> Pizza -> Pizza

Capa Queso :: Pizza -> Pizza
```

Funciones como datos

Funciones como datos

- Queremos definir un tipo para representar conjuntos

```
data Set a = ...
```

- ¿Qué operaciones podemos hacer sobre un `Set a`?

```
emptySet :: Set a
belongs :: a -> Set a -> Bool
singleton :: Eq a => a -> Set a
union :: Set a -> Set a -> Set a
```

- ¿Cómo se puede implementar este tipo?

Funciones como datos

- Considerar la siguiente definición

```
data Set a = S (a -> Bool)
```

- ¿Qué valores hay un `Set Int`?
  - Todos los elementos se construyen con el constructor `S`
  - ¿De qué tipo es el argumento que debe llevar `S`?
    - ¡Una función!
  - ¿Y esto es una representación válida de conjuntos?
    - Sí, porque las funciones pueden ser datos

```haskell
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
emptySet :: Set a
emptySet = …
```

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = …
```

  - Si es un `Set`, debe usar el constructor `S`

```haskell
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
emptySet :: Set a
emptySet = S …
```

  - Si es un `Set`, debe usar el constructor `S`

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = S …
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = S …
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = S (\x -> …)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `x`, ¿está o no en el conjunto vacío? ¿Qué devolver entonces?

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = S (\x -> …)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `x`, ¿está o no en el conjunto vacío? ¿Qué devolver entonces?
    - El valor de verdad falso: ningún elemento está

```
data Set a = S (a -> Bool)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
emptySet :: Set a
emptySet = S (\x -> False)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `x`, ¿está o no en el conjunto vacío? ¿Qué devolver entonces?
    - El valor de verdad falso: ningún elemento está

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs … …       = …
```

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x …       = …
```

  - Comenzamos por hacer un PM sobre el `Set`

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
belongs :: a -> Set a -> Bool
belongs x (S f) = …
```

  - Comenzamos por hacer un PM sobre el `Set`

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x (S f) = …
```

  - Comenzamos por hacer un PM sobre el `Set`
  - ¿Qué tipo tiene `f`?

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x (S f) = …
```

  - Comenzamos por hacer un PM sobre el **`Set`**
  - ¿Qué tipo tiene **`f`**?
    - `f :: a -> Bool`

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x (S f) = …
```

  - Comenzamos por hacer un PM sobre el `Set`
  - ¿Qué tipo tiene `f`?
    - `f :: a -> Bool`
  - ¿Y cómo sabemos si `x` pertenece al conjunto?

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x (S f) = …
```

- Comenzamos por hacer un PM sobre el `Set`
- ¿Qué tipo tiene `f`?
  - `f :: a -> Bool`
- ¿Y cómo sabemos si `x` pertenece al conjunto?
  - ¡Es justo la información que `f` expresa!

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
belongs :: a -> Set a -> Bool
belongs x (S f) = f x
```

- Comenzamos por hacer un PM sobre el `Set`
- ¿Qué tipo tiene `f`?
  - `f :: a -> Bool`
- ¿Y cómo sabemos si `x` pertenece al conjunto?
  - ¡Es justo la información que `f` expresa!

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
singleton :: Eq a => a -> Set a
singleton x = …
```

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = …
```

  - Si es un `Set`, debe usar el constructor `S`

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = S …
```

  - Si es un `Set`, debe usar el constructor `S`

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = S …
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = S …
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = S (\y -> …)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
singleton :: Eq a => a -> Set a
singleton x = S (\y -> …)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `y`, ¿está o no en el singleton {`x`}? ¿Qué devolver entonces?

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
singleton :: Eq a => a -> Set a
singleton x = S (\y -> …)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `y`, ¿está o no en el singleton {`x`}? ¿Qué devolver entonces?
    - Tenemos que comprar a `x` con `y`

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
```

- ¿Y cómo implementamos las operaciones?

```
singleton :: Eq a => a -> Set a
singleton x = S (\y -> x==y)
```

  - Si es un `Set`, debe usar el constructor `S`
  - ¿Cómo escribimos la función que es argumento de `S`?
    - Por ejemplo, con una expresión lambda
  - Y para un elemento `y`, ¿está o no en el singleton {`x`}? ¿Qué devolver entonces?
    - Tenemos que comprar a `x` con `y`

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union    …         …      = …
```

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union    …      …     = …
```

  - Comenzamos por hacer PM sobre ambos `Set`s

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = …
```

  - Comenzamos por hacer PM sobre ambos `Set`s

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = …
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = …
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = S …
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = S …
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor
  - El argumento es una expresión lambda

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

Funciones como datos

- ¿Y cómo implementamos las operaciones?

```haskell
union :: Set a -> Set a -> Set a
union (S f) (S g) = S (\x -> …)
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor
  - El argumento es una expresión lambda

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = S (\x -> …)
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor
  - El argumento es una expresión lambda
  - El elemento `x` pertenece si pertenece a alguno de ellos

Funciones como datos

```
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
```

- ¿Y cómo implementamos las operaciones?

```
union :: Set a -> Set a -> Set a
union (S f) (S g) = S (\x -> f x || g x)
```

  - Comenzamos por hacer PM sobre ambos `Set`s
  - Recordar que `f` y `g` son funciones de tipo `a -> Bool`
  - El resultado es un `Set`, y así, debe usar el constructor
  - El argumento es una expresión lambda
  - El elemento `x` pertenece si pertenece a alguno de ellos

```haskell
data Set a = S (a -> Bool)
emptySet = S (\x -> False)
belongs x (S f) = f x
singleton x = S (\y -> x==y)
union (S f) (S g) =
  S (\x -> f x && g x)
```

Funciones como datos

- Conclusiones
  - Las funciones pueden ser datos
  - Un argumento de función solamente se puede aplicar
  - Pero se pueden armar muchas operaciones así
  - ¿Qué funciones no se pueden expresar con esta representación?
    - Enumeración de elementos (salvo que **`a`** sea finito)
    - **`set2list :: Set a -> [a]`**
    - El problema es que hay que mirar infinitos valores

Esquemas de funciones

Esquemas de funciones

- Considerar las siguientes funciones

```
succs :: [Int] -> [Int]          -- Suma uno a cada elemento
succs  ...

uppers :: [Char] -> [Char]       -- Pasa cada carácter a mayúsculas
uppers ...

tests :: [Int] -> [Bool]         -- Cambia cada número a un bool
tests  ...                       --  que dice si es 0 o no
```

- ¿Cómo definirlas?

Esquemas de funciones

- Considerar las siguientes funciones

```
succs :: [Int] -> [Int]          -- Suma uno a cada elemento
succs  []     = ...
succs  (n:ns) = ... n ... succs ns ...

uppers :: [Char] -> [Char]       -- Pasa cada carácter a mayúsculas
uppers []     = ...
uppers (c:cs) = ... c ... uppers cs ...

tests :: [Int] -> [Bool]         -- Cambia cada número a un bool
tests  []     = ...              --  que dice si es 0 o no
tests  (x:xs) = ... x ... tests xs ...
```

- ¡Por recursión en la estructura de listas!

Esquemas de funciones

- Considerar las siguientes funciones

```
succs :: [Int] -> [Int]            -- Suma uno a cada elemento
succs  []      = ...
succs  (n:ns)  = n+1 : succs ns

uppers :: [Char] -> [Char]         -- Pasa cada carácter a mayúsculas
uppers []      = ...
uppers (c:cs)  = upper c : uppers cs

tests :: [Int] -> [Bool]           -- Cambia cada número a un bool
tests  []      = ...               --  que dice si es 0 o no
tests  (x:xs)  = x ==0 : tests xs
```

- Primero los casos recursivos…

Esquemas de funciones

- Considerar las siguientes funciones

```
succs :: [Int] -> [Int]              -- Suma uno a cada elemento
succs  []      = []
succs  (n:ns)  = n+1 : succs ns

uppers :: [Char] -> [Char]           -- Pasa cada carácter a mayúsculas
uppers []      = []
uppers (c:cs)  = upper c : uppers cs

tests :: [Int] -> [Bool]             -- Cambia cada número a un bool
tests  []      = []                  --  que dice si es 0 o no
tests  (x:xs)  = x ==0 : tests xs
```

  - …finalmente los casos base

Esquemas de funciones

- Considerar las siguientes funciones

```
succs :: [Int] -> [Int]           -- Suma uno a cada elemento
succs  []     = []
succs  (n:ns) = n+1 : succs ns
uppers :: [Char] -> [Char]        -- Pasa cada carácter a mayúsculas
uppers []     = []
uppers (c:cs) = upper c: uppers cs
tests :: [Int] -> [Bool]          -- Cambia cada número a un bool
tests  []     = []                --  que dice si es 0 o no
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común? ¡Se marcan las diferencias!

Esquemas de funciones

- ¿Qué tienen en común?

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs   []     = []
succs   (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests   []     = []
tests   (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común?
  - Todas procesan cada elemento de alguna manera

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs   []     = []
succs   (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests   []     = []
tests   (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común?
  - Todas procesan cada elemento de alguna manera
  - ¿Se puede pedir ese procesamiento como parámetro?

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común?
  - Todas procesan cada elemento de alguna manera
  - ¿Se puede pedir ese procesamiento como parámetro?
    - ¡Sí! Con funciones de orden superior

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común?
  - Todas procesan cada elemento de alguna manera
  - ¿Se puede pedir ese procesamiento como parámetro?
    - ¡Sí! Con funciones de orden superior

```
map :: ??
map f []     = []
map f (x:xs) = f x : map f xs
```

Esquemas de funciones

```haskell
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tienen en común?
  - Todas procesan cada elemento de alguna manera
  - ¿Se puede pedir ese procesamiento como parámetro?
    - ¡Sí! Con funciones de orden superior

```haskell
map :: ??
map f []     = []
map f (x:xs) = f x : map f xs
```

  - ¿Qué tipo tiene **map**?

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tipo tiene **map**?

```
map :: ??
map f []     = []
map f (x:xs) = f x : map f xs
```

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs   []     = []
succs   (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests   []     = []
tests   (x:xs) = x ==0 : tests xs
```

- ¿Qué tipo tiene **map**?

```
map ::    ??   ->  ??   -> ??
map f []       = []
map f (x:xs)   = f x : map f xs
```

  - Tiene dos parámetros

```
succs :: [Int] -> [Int]
succs   []     = []
succs   (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests   []     = []
tests   (x:xs) = x ==0 : tests xs
```

Esquemas de funciones

- ¿Qué tipo tiene **`map`**?

```
map ::    ??    -> [?] -> [?]
map f []        = []
map f (x:xs)    = f x : map f xs
```

  - Tiene dos parámetros
  - El segundo parámetro y el resultado son listas

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tipo tiene **map**?

```
map :: (?->?) -> [?] -> [?]
map f []       = []
map f (x:xs)   = f x : map f xs
```

  - Tiene dos parámetros
  - El segundo parámetro y el resultado son listas
  - El primer parámetro es una función

```haskell
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

Esquemas de funciones

- ¿Qué tipo tiene **map**?

```haskell
map :: (a->?) -> [a] -> [?]
map f []      = []
map f (x:xs)  = f x : map f xs
```

  - Tiene dos parámetros
  - El segundo parámetro y el resultado son listas
  - El primer parámetro es una función
  - La función transforma los elementos de la lista de entrada…

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- ¿Qué tipo tiene **map**?

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

  - Tiene dos parámetros
  - El segundo parámetro y el resultado son listas
  - El primer parámetro es una función
  - La función transforma los elementos de la lista de entrada en los elementos de la lista de salida
    - Por eso se usan 2 variables de tipo distintas

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- Esquema de **map**

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- **“En francés”:** dada una función y una lista, transforma la lista dada elemento a elemento, según la función dada

Esquemas de funciones

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

- Esquema de **`map`**

```
map :: (a->b) -> ([a] -> [b])
map f []     = []
map f (x:xs) = f x : tests xs
```

- **Currificada:** es una *función* que transforma funciones de elementos en funciones de listas

```haskell
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

Esquemas de funciones

- Esquema de **map**

```haskell
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- **"En francés":** dada una función y una lista, transforma la lista dada elemento a elemento, según la función dada
- **Currificada:** es una *función* que transforma funciones de elementos en funciones de listas
- ¡Ambas lecturas son posibles!

Esquemas de funciones

- Esquema de **map**
  - Se puede aplicar parcialmente…
  - …y así expresar otras funciones

```
succs'  = map (\x -> x+1)
uppers' = map (\x -> upper x)
tests'  = map (\x -> x==0)
```

- Las definiciones son equivalentes
  - succs' = succs
  - uppers' = uppers
  - tests' = tests

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : tests xs

succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n+1 : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x ==0 : tests xs
```

Esquemas de funciones

- Esquema de **`map`**
  - ¿Por qué se llama “map”?
  - Según el diccionario Merriam-Webster, “to map” significa *“to be assigned in a relation or connection”*
  - En castellano podría traducirse como “asociar”, “relacionar”, “conectar”
  - Así, **`map`** “conecta” cada elemento de la lista de entrada con un elemento de la lista de salida

Esquemas de funciones

- Esquema de **map**
  - "Asignar", "conectar", "relacionar"

```
map :: (a->b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : tests xs

succs' = map (\n -> n+1)
uppers' = map (\x -> upper x)
tests' = map (\x -> x==0)
```

  - Otra forma de ver su definición

Resumen

Resumen

- Funciones como valores
  - Funciones anónimas y regla Beta
  - Funciones como resultado de otras funciones
  - Currificación y aplicación parcial
  - Funciones como parámetro de otras funciones
    - Funciones de orden superior
  - Funciones como datos
  - Esquemas de funciones sobre listas: map

Apéndice I
Sobre la asociatividad

Sobre la asociatividad

- Consideremos esta expresión

```
10 - 3 - 2
```

  - ¿Qué número denota? ¿`5` o `9`?

Sobre la asociatividad

- 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?

Sobre la asociatividad

- 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!

Sobre la asociatividad

- 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"*?

Volver

Apéndice II
Deducción del esquema de map

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []      = []
succs  (n:ns)  = n + 1    : succs ns
uppers :: [Char] -> [Char]
uppers []      = []
uppers (c:cs)  = upper c  : uppers cs
tests :: [Int] -> [Bool]
tests  []      = []
tests  (x:xs)  = x == 0   : tests xs
```

- ¿Cómo se arregla? **Hay que extender la técnica**

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n + 1     : succs ns
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c   : uppers cs
tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x == 0    : tests xs
```

  - **Técnica extendida:** identificar los nombres problemáticos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n + 1       : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper c     : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x == 0      : tests xs
```

  - **Técnica extendida:** separarlos con la misma técnica

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs   []     = []
                    n
succs   (n:ns) = ( )+ 1       : succs ns
uppers :: [Char] -> [Char]
uppers  []     = []
                            c
uppers  (c:cs) = upper ( )    : uppers cs
tests :: [Int] -> [Bool]
tests   []     = []
                    x
tests   (x:xs) = ( )== 0      : tests xs
```

- **Técnica extendida:** separarlos con la misma técnica

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = ( )+ 1     n    : succs ns
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper ( )    c  : uppers cs
tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = ( )== 0    x    : tests xs
```

- **Técnica extendida:** separarlos con la misma técnica

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = ( )+ 1      n    : succs ns
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper ( )   c    : uppers cs
tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = ( )== 0     x    : tests xs
```

- **Técnica extendida:** separarlos con la misma técnica

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = ( )+ 1    n : succs ns
                 x
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = upper ( )  c : uppers cs
                       x
tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = ( )== 0   x : tests xs
                 n
```

- **Técnica extendida:** nombrar los agujeros resultantes…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = (      ( ) + 1)  n : succs ns
                        x
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = (        upper ( )) c : uppers cs
                                x
tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = (      ( ) == 0)  x : tests xs
                        n
```

- **Técnica extendida:** nombrar los agujeros resultantes…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []      = []
succs  (n:ns)  = (\x-> (x) + 1) n : succs ns
uppers :: [Char] -> [Char]
uppers []      = []
uppers (c:cs)  = (\x-> upper (x)) c : uppers cs
tests :: [Int] -> [Bool]
tests  []      = []
tests  (x:xs)  = (\n-> (n) == 0) x : tests xs
```

- **Técnica extendida:** …y volverlos parámetros

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs   []     = []
succs   (n:ns) = (\x-> x + 1)   n : succs ns
uppers :: [Char] -> [Char]
uppers []      = []
uppers (c:cs)  = (\x-> upper x) c : uppers cs
tests :: [Int] -> [Bool]
tests   []     = []
tests   (x:xs) = (\n-> n == 0)  x : tests xs
```

- **Técnica extendida:** Se puede seguir con los demás pasos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = (\x-> x + 1) n : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = (\x-> upper x) c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = (\n-> n == 0) x : tests xs
```

- **Técnica extendida:** Recortar los recuadros (ahora independientes)...

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []             (\x-> x + 1)
succs  (n:ns) =                    n : succs ns
uppers :: [Char] -> [Char]
uppers []     = []             (\x-> upper x)
uppers (c:cs) =                    c : uppers cs
tests :: [Int] -> [Bool]
tests  []     = []             (\n-> n == 0)
tests  (x:xs) =                    x : tests xs
```

- **Técnica extendida:** …y separarlos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs   []     = []                        (\x-> x + 1)
succs   (n:ns) =              n  :  succs ns

uppers :: [Char] -> [Char]
uppers []      = []                        (\x-> upper x)
uppers (c:cs)  =              c  :  uppers cs

tests :: [Int] -> [Bool]
tests   []     = []                        (\n-> n == 0)
tests   (x:xs) =              x  :  tests xs
```

- **Técnica extendida:** …y separarlos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs   []      = []
succs   (n:ns)  =            n  :  succs ns                (\x-> x + 1)

uppers :: [Char] -> [Char]
uppers  []      = []
uppers  (c:cs)  =            c  :  uppers cs               (\x-> upper x)

tests :: [Int] -> [Bool]
tests   []      = []
tests   (x:xs)  =            x  :  tests xs                (\n-> n == 0)
```

- **Técnica extendida:** …y separarlos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []      = []
succs  (n:ns)  =       n : succs ns                  (\x-> x + 1)
uppers :: [Char] -> [Char]
uppers []      = []
uppers (c:cs)  =       c : uppers cs                 (\x-> upper x)
tests :: [Int] -> [Bool]
tests  []      = []
tests  (x:xs)  =       x : tests xs                  (\n-> n == 0)
```

- **Técnica extendida:** …y separarlos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs   []      = []
succs   (n:ns) =     n : succs ns          succs'   = ......... (\x-> x + 1)
uppers :: [Char] -> [Char]
uppers []      = []
uppers (c:cs) =     c : uppers cs          uppers' = ......... (\x-> upper x)
tests :: [Int] -> [Bool]
tests   []      = []
tests   (x:xs) =     x : tests xs          tests'   = ......... (\n-> n == 0)
```

- **Técnica extendida:** …y separarlos

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) =    n : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) =    c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) =    x : tests xs
```

```
succs'  = ........ (\x-> x + 1)

uppers' = ........ (\x-> upper x)

tests'  = ........ (\n-> n == 0)
```

- **Técnica extendida:** Identificar las partes comunes…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
succs  []     = []
succs  (n:ns) = n : succs ns

uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = c : uppers cs

tests :: [Int] -> [Bool]
tests  []     = []
tests  (x:xs) = x : tests xs
```

```
succs'  = (\x-> x + 1)

uppers' = (\x-> upper x)

tests'  = (\n-> n == 0)
```

- **Técnica extendida:** Identificar las partes comunes…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
tests :: [Int] -> [Bool]
tests []     = []
tests (x:xs) = [ ] x : tests xs
```

```
succs'  = ........ (\x-> x + 1)

uppers' = ........ (\x-> upper x)

tests'  = ........ (\n-> n == 0)
```

- **Técnica extendida:** Identificar las partes comunes…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map ::              [a] -> [b]
map     []        = []
map     (x:xs)    =     x : map    xs
```

```
succs'   = ......... (\x-> x + 1)

uppers'  = ......... (\x-> upper x)

tests'   = ......... (\n-> n == 0)
```

- **Técnica extendida:** Identificar las partes comunes y nombrarlas

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map ::                [a] -> [b]
map     []        = []
map     (x:xs)    = [    ] x : map    xs
                     f
```

```
succs'   = ........ (\x-> x + 1)

uppers'  = ........ (\x-> upper x)

tests'   = ........ (\n-> n == 0)
```

- **Técnica extendida:** Nombrar el recuadro…

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: ?? -> [a] -> [b]
map f []     = []
map f (x:xs) = [   ]x : map f xs
                 f
```

```
succs'  = ......... (\x-> x + 1)
uppers' = ......... (\x-> upper x)
tests'  = ......... (\n-> n == 0)
```

- **Técnica extendida:** ...y ponerlo de parámetro

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: ?? -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
succs'  = ........ (\x-> x + 1)
uppers' = ........ (\x-> upper x)
tests'  = ........ (\n-> n == 0)
```

- **Técnica extendida:** ...y ponerlo de parámetro

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: ??  -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
succs'  = map (\x-> x + 1)
uppers' = map (\x-> upper x)
tests'  = map (\n-> n == 0)
```

- **Técnica extendida:** Recuperar los datos con la función hecha

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map ::    ??    -> [a] -> [b]
map f  []       = []
map f  (x:xs)   = f  x : map f xs
```

```
succs'  = map (\x-> x + 1)
uppers' = map (\x-> upper x)
tests'  = map (\n-> n == 0)
```

- **Técnica extendida:** Recuperar los datos con la función hecha

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: ?? -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
succs'  = map (\x-> x + 1)
uppers' = map (\x-> upper x)
tests'  = map (\n-> n == 0)
```

- ¿Qué tipo tiene el parámetro?

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: ( -> ) -> [a] -> [b]
map f  []     = []
map f  (x:xs) = f x : map f xs

        succs'  = map (\x-> x + 1)
        uppers' = map (\x-> upper x)
        tests'  = map (\n-> n == 0)
```

- ¿Qué tipo tiene el parámetro?
  - ¡Es una función!

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: (a->b) -> [a] -> [b]
map f   []     = []
map f   (x:xs) = f x : map f xs

succs'  = map (\x-> x + 1)
uppers' = map (\x-> upper x)
tests'  = map (\n-> n == 0)
```

  - ¿Qué tipo tiene el parámetro?
    - ¡Es una función!

Deducción del esquema de map

- Parámetros y funciones recursivas sobre listas

```
map :: (a->b) -> [a] -> [b]
map f   []     = []
map f   (x:xs) = f x : map f xs

succs'  = map (\x-> x + 1)
uppers' = map (\x-> upper x)
tests'  = map (\n-> n == 0)
```

  - El esquema de map (asociación), transforma listas elemento a elemento, según el parámetro dado

Volver