



Programación Funcional

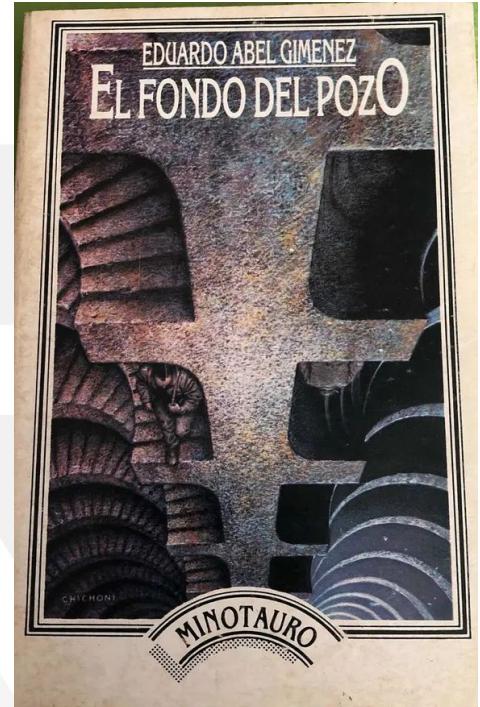
Clases teóricas
por Pablo E. “Fidel” Martínez López

11. Esquemas de funciones I

“Detrás de cada acontecimiento hay un truco de espejos. Nada es, todo parece. Escóndase, si quiere. Espíe por las ranuras. Alguien estará preparando otra ilusión. Las diferencias entre las personas son las diferencias entre las ilusiones que perciben.”

(Consejero, 121:6:33)”

El Fondo del Pozo
Eduardo Abel Giménez



Parámetros y parametrización

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ Considerar las siguientes expresiones que denotan números

$$1 + 1$$

$$1 + 16$$

$$1 + 41$$

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ Considerar las siguientes expresiones que denotan números

$$a = 1 + 1$$

$$b = 1 + 16$$

$$c = 1 + 41$$

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 0:** le damos nombre para distinguirlas

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$a = 1 + \boxed{1}$$

$$b = 1 + \boxed{16}$$

$$c = 1 + \boxed{41}$$

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 1:** recuadrar las diferencias

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{1} \\ b &= 1 + \boxed{16} \\ c &= 1 + \boxed{41} \end{aligned}$$

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros...

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{1} \\ b &= 1 + \boxed{16} \\ c &= 1 + \boxed{41} \end{aligned}$$

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros y separarlos

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$a = 1 + \boxed{}$$

1

$$b = 1 + \boxed{}$$

16

$$c = 1 + \boxed{}$$

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros y separarlos

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$a = 1 + \boxed{}$$

1

$$b = 1 + \boxed{}$$

16

$$c = 1 + \boxed{}$$

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros y separarlos

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$a = 1 + \boxed{}$$

1

$$b = 1 + \boxed{}$$

16

$$c = 1 + \boxed{}$$

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros y separarlos

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$a = 1 + \boxed{}$$

1

$$b = 1 + \boxed{}$$

16

$$c = 1 + \boxed{}$$

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 2:** recortar los recuadros y separarlos

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{} \\ b &= 1 + \boxed{} \\ c &= 1 + \boxed{} \end{aligned}$$

$$1 + \boxed{}$$

1

16

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 3:** identificar las partes comunes que quedaron

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{b} \\ b &= 1 + \boxed{c} \\ c &= 1 + \boxed{1} \end{aligned} \quad \begin{aligned} s &= 1 + \boxed{1} \end{aligned}$$

1

16

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 3:** identificar las partes comunes y darles nombre

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{} \\ b &= 1 + \boxed{} \\ c &= 1 + \boxed{} \end{aligned} \quad \begin{aligned} s &= 1 + \boxed{} \\ &\quad \text{y} \end{aligned}$$

1

16

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 4:** nombrar el recuadro...

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{} \\ b &= 1 + \boxed{} \\ c &= 1 + \boxed{} \end{aligned}$$

$\xrightarrow{\quad}$

$$s = \text{\textbackslash}y\text{\textgreater} 1 + \boxed{}_y$$

1

16

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 4:** nombrar el recuadro y ponerlo de parámetro

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar

$$\begin{aligned} a &= 1 + \boxed{y} \\ b &= 1 + \boxed{y} \\ c &= 1 + \boxed{y} \end{aligned}$$

$\xrightarrow{\quad}$

$$s = \lambda y \rightarrow 1 + \boxed{y}$$

1

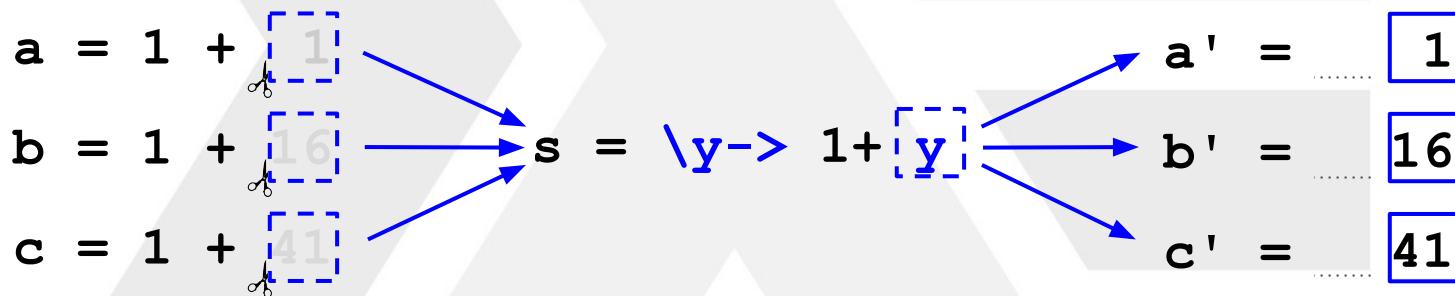
16

41

- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 4:** nombrar el recuadro y ponerlo de parámetro

Parámetros y la “técnica de los recuadros”

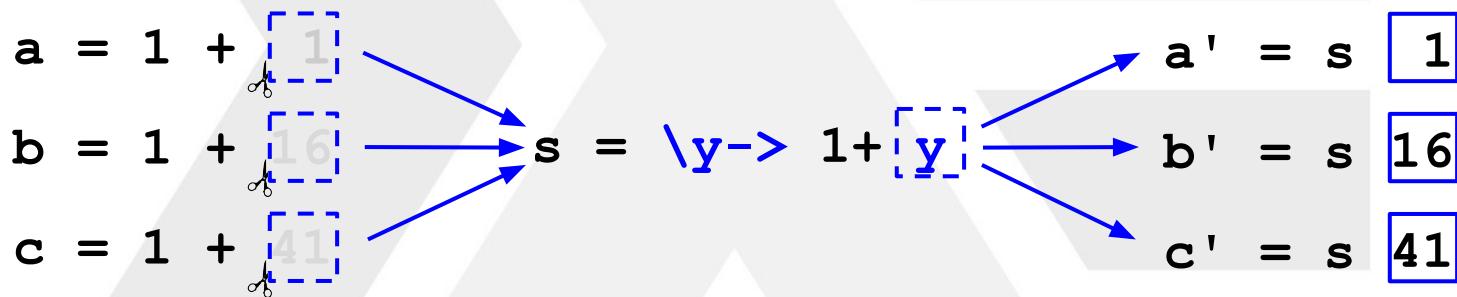
- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar



- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 5:** recuperar los datos...

Parámetros y la “técnica de los recuadros”

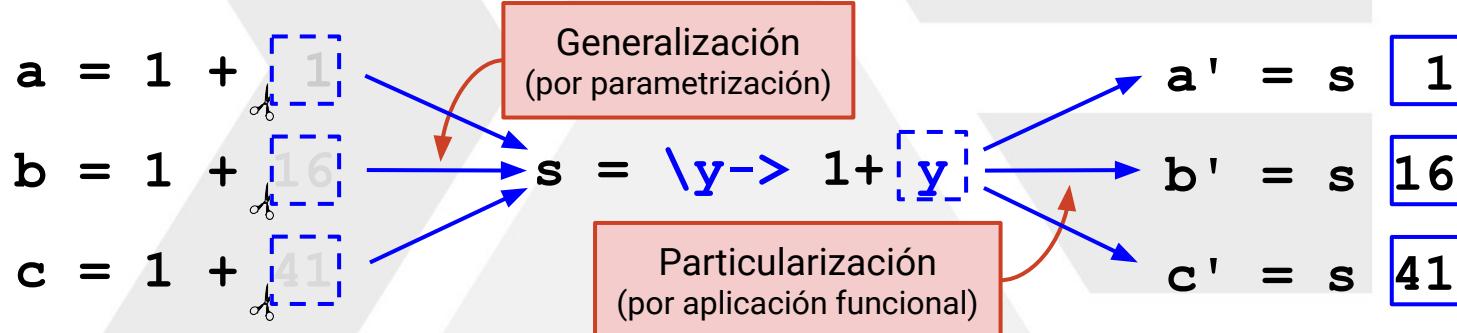
- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar



- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ **Paso 5:** recuperar los datos usando la función obtenida

Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar



- ❑ ¿Qué tienen en común? ¿Cómo se puede aprovechar?
 - ❑ Generalización (**abstracción**) mediante **parametrización**

Parámetros y la “técnica

- ❑ ¿Qué es un parámetro? ¿Cómo se usa?
- ❑ *Técnica de los recuadros para*

$$a = 1 + 1$$

$$b = 1 + 16$$

$$c = 1 + 41$$

$$s = \text{y} \rightarrow 1 + \text{y}$$

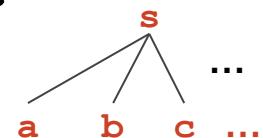
$$a' = s \ 1$$

$$b' = s \ 16$$

$$c' = s \ 41$$



- ❑ Se puede demostrar que $a = a'$, $b = b'$ y $c = c'$
 - ❑ Se definió una *función s*



Parámetros y la “técnica de los recuadros”

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
- ❑ **Técnica de los recuadros** para parametrizar
 - ❑ ¿Cómo realizarla?
 - ❑ **Paso 1:** recuadrar las diferencias
 - ❑ **Paso 2:** recortar los recuadros y separarlos
 - ❑ **Paso 3:** identificar las partes comunes que quedaron
 - ❑ **Paso 4:** nombrar el recuadro y ponerlo de parámetro
 - ❑ **Paso 5:** recuperar los datos usando la función obtenida
- ❑ Generalización (**abstracción**) mediante **parametrización**

Parámetros y parametrización

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
 - ❑ Desde el punto de vista *semántico* (denotacional)
 - ❑ Es un dato que falta y debe suministrarse
 - ❑ Se expresa mediante un nombre que lo representa
 - ❑ Desde el punto de vista *sintáctico* (operacional)
 - ❑ Es un *mecanismo simbólico y operacional*
 - ❑ Permite definir funciones por *parametrización* y usarlas por *aplicación*

Parámetros y parametrización

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
 - ❑ Desde el punto de vista *semántico* (denotacional)
 - ❑ Es un dato que falta y debe suministrarse
 - ❑ Se expresa mediante un nombre que lo representa
 - ❑ Desde el punto de vista *sintáctico* (operacional)
 - ❑ Es un *mecanismo simbólico y operacional*
 - ❑ Permite definir funciones por *parametrización* y usarlas por *aplicación*
- ❑ **¡Abstraer** es detectar y aprovechar similaridades!

Parámetros y parametrización

- ❑ ¿Qué es un parámetro? ¿Cómo entenderlo?
 - ❑ Un parámetro es un mecanismo sintáctico y semántico de lenguajes de programación para expresar abstracción

Parámetros y órdenes superiores

- ❑ Parámetros y funciones currificadas
- ❑ Técnica de los recuadros

`f = \y-> 2 +y`

`g = \y-> 17 +y`

`h = \y-> 42 +y`

- ❑ Considerar las siguientes funciones

Parámetros y órdenes superiores

- ❑ Parámetros y funciones currificadas
- ❑ Técnica de los recuadros

$f = \lambda y \rightarrow \boxed{2} + y$

$g = \lambda y \rightarrow \boxed{17} + y$

$h = \lambda y \rightarrow \boxed{42} + y$

- ❑ Paso 1: recuadrar las diferencias

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

```
f = \y-> 2+y
g = \y-> 17+y
h = \y-> 42+y
```

- Paso 2: recortar los recuadros...

Parámetros y órdenes superiores

- ❑ Parámetros y funciones currificadas
- ❑ Técnica de los recuadros

$f = \lambda y. \boxed{y} + y$

$g = \lambda y. \boxed{y} + y$

$h = \lambda y. \boxed{y} + y$

2

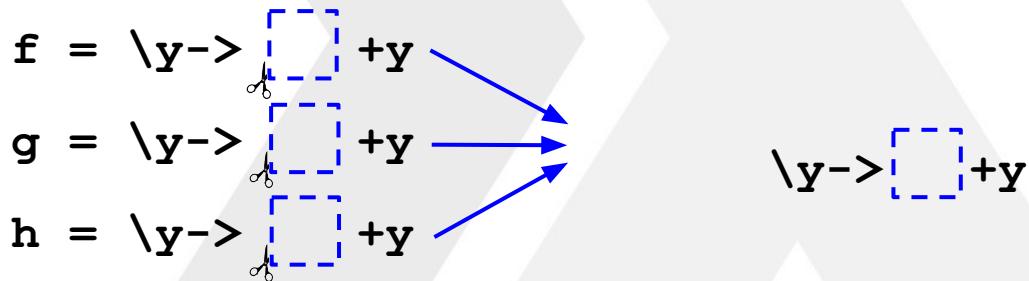
17

42

- ❑ Paso 2: recortar los recuadros y separarlos

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros



2

17

42

- Paso 3: identificar las partes comunes...

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

```
f = \y-> [ ] +y
g = \y-> [ ] +y
h = \y-> [ ] +y
```

sum = $\backslash y -> [] +y$

```
graph LR
    f["f = \y-> [ ] +y"] --> sum["sum = \y-> [ ] +y"]
    g["g = \y-> [ ] +y"] --> sum
    h["h = \y-> [ ] +y"] --> sum
```

2

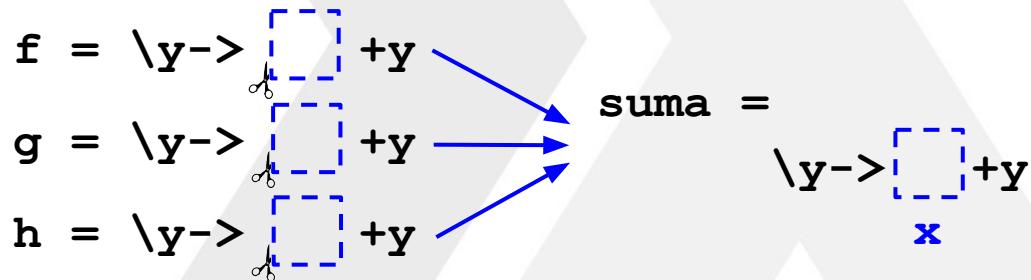
17

42

- Paso 3: identificar las partes comunes y darles nombre

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros



- Paso 4: nombrar el recuadro...

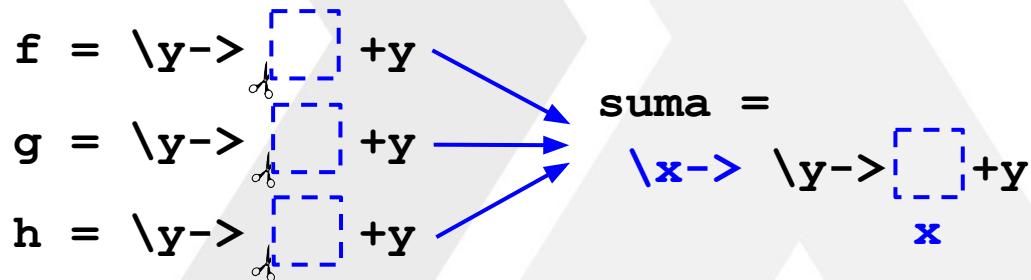
2

17

42

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros



2

17

42

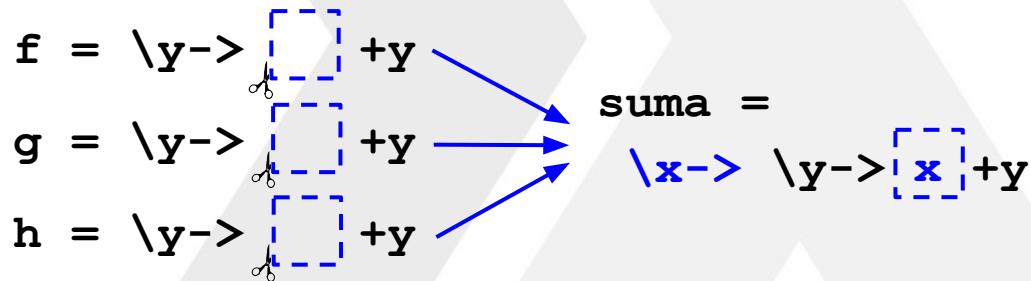
- Paso 4: nombrar el recuadro y ponerlo de parámetro

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

```
f = \y-> [y] +y
g = \y-> [y] +y
h = \y-> [y] +y
```

sum =
 $\lambda x. \lambda y. [x] + y$



2

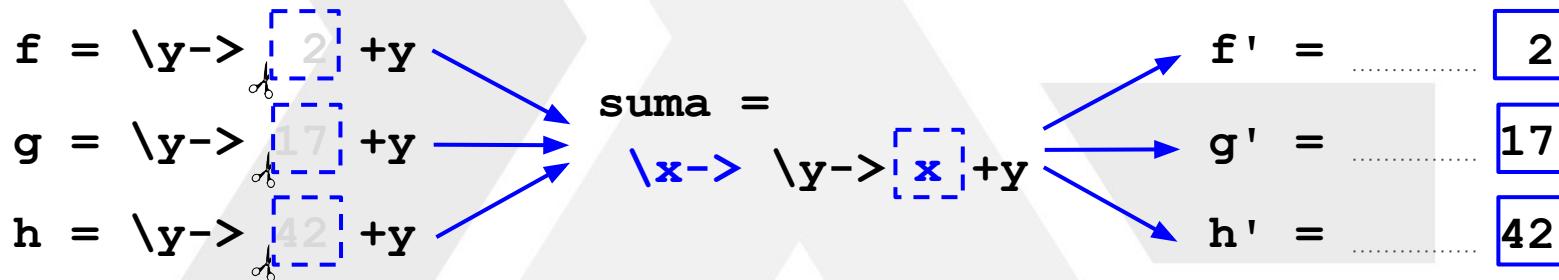
17

42

- Paso 4: nombrar el recuadro y ponerlo de parámetro

Parámetros y órdenes superiores

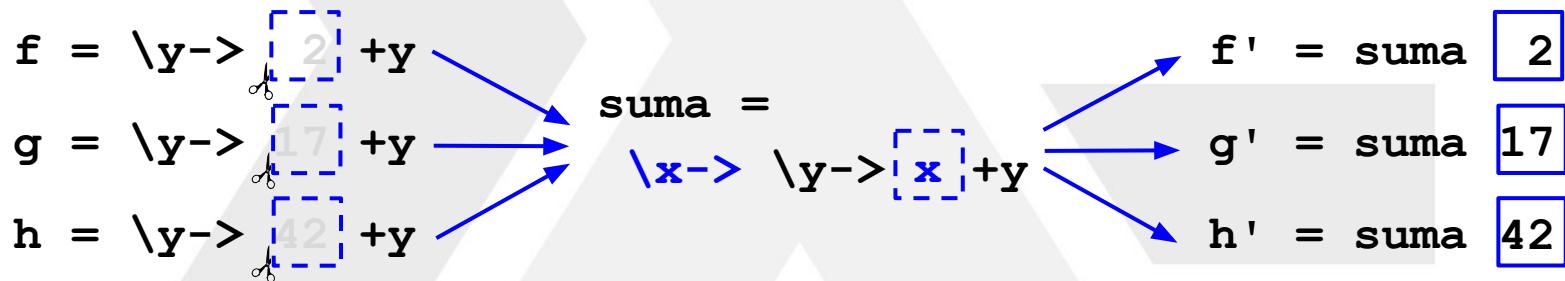
- Parámetros y funciones currificadas
- Técnica de los recuadros



- Paso 5: recuperar los datos...

Parámetros y órdenes superiores

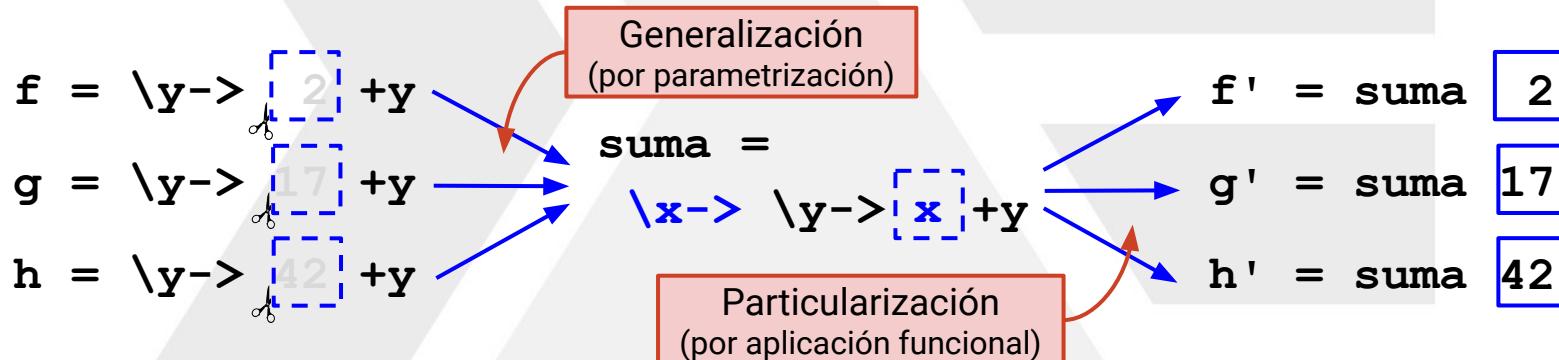
- Parámetros y funciones currificadas
- Técnica de los recuadros



- Paso 5: recuperar los datos usando la función obtenida

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros



- Nuevamente, **generalización por parametrización**
- No importa si los datos son funciones

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

```
f = \y-> 2 +y
```

```
g = \y-> 17 +y
```

```
h = \y-> 42 +y
```

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

```
f' = suma 2
```

```
g' = suma 17
```

```
h' = suma 42
```

- Si los datos son funciones,
 - la abstracción produce funciones currificadas, de orden superior

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

$f = \lambda y \rightarrow 2 + y$

$g = \lambda y \rightarrow 17 + y$

$h = \lambda y \rightarrow 42 + y$

suma $x =$
 $\lambda y \rightarrow x + y$

$f' = \text{suma } 2$

$g' = \text{suma } 17$

$h' = \text{suma } 42$

- La notación puede dificultar visualizar el proceso

Parámetros y órdenes superiores

- Parámetros y funciones currificadas
- Técnica de los recuadros

$f\ y = 2 + y$

$g\ y = 17 + y$

$h\ y = 42 + y$

suma $x\ y =$
 $x + y$

$f' = \text{suma } 2$

$g' = \text{suma } 17$

$h' = \text{suma } 42$

- La notación puede dificultar visualizar el proceso

Parámetros y órdenes su

- Parámetros y funciones cur
- Técnica de los recuadros

$$f(y) = 2 + y$$

$$g(y) = 17 + y$$

$$h(y) = 42 + y$$

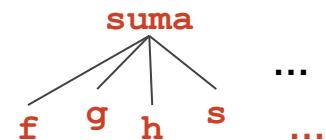
$$f' = \text{suma } 2$$

$$g' = \text{suma } 17$$

$$h' = \text{suma } 42$$



- La notación puede dificultar visualizar detalles
 - Se puede demostrar que $f = f'$, $g = g'$ y $h = h'$



Esquemas de funciones: presentación

Parámetros y esquemas de funciones

- ❑ Generalización (*abstracción*) mediante parametrización
 - ❑ Es una forma de aprovechar el orden superior
- ❑ ¿Se podrá utilizar en funciones recursivas de forma útil?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

□ Definir las siguientes funciones sobre listas

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
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 hacerlas?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

□ Luego los casos base...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
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? Probar la técnica de los recuadros

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

□ Técnica de los recuadros. Paso 1: recuadrar las diferencias

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
suucs :: [Int] -> [Int]           -- Suma uno a cada elemento
suucs []      = [1]
suucs (n:ns) = n + 1                suucs 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 []      = [1]                  -- que dice si es 0 o no
tests (x:xs) = x == 0              tests xs
```

□ ¿Qué problema tiene usarla así? Paso 2: recortar y separar

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
```

```
succs [] = []
succs (n:ns) = [ ] : succs ns
```

($n + 1$)

```
uppers :: [Char] -> [Char]
```

```
uppers [] = []
uppers (c:cs) = [ ] : uppers cs
```

(upper c)

```
tests :: [Int] -> [Bool]
```

```
tests [] = []
tests (x:xs) = [ ] : tests xs
```

($x == 0$)

□ ¿Qué problema tiene usarla así?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

($n + 1$)

(upper c)

($x == 0$)

□ ¿Qué problema tiene usarla así?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

(*n* + 1)

(upper *c*)

(*x* == 0)

□ ¿Qué problema tiene usarla así?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
```

```
succs [] = []
```

```
succs (n:ns) = [ ] : succs ns
```

```
uppers :: [Char] -> [Char]
```

```
uppers [] = []
```

```
uppers (c:cs) = [ ] : uppers cs
```

```
tests :: [Int] -> [Bool]
```

```
tests [] = []
```

```
tests (x:xs) = [ ] : tests xs
```

```
succs' = ..... (n + 1)
```

```
uppers' = ..... (upper c)
```

```
tests' = ..... (x == 0)
```

□ ¿Qué problema tiene usarla así?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
```

```
succs [] = []
```

```
succs (n:ns) = [ ] : succs ns
```

```
uppers :: [Char] -> [Char]
```

```
uppers [] = []
```

```
uppers (c:cs) = [ ] : uppers cs
```

```
tests :: [Int] -> [Bool]
```

```
tests [] = []
```

```
tests (x:xs) = [ ] : tests xs
```

```
succs' = ..... (n + 1)
```

```
uppers' = ..... (upper c)
```

```
tests' = ..... (x == 0)
```

¡El parámetro pierde su *ligadura*!

□ ¿Qué problema tiene usarla así?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

(*n* + 1)

(upper *c*)

(*x* == 0)

□ ¿Cómo se arregla? Volviendo para atrás...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

($n + 1$)

(upper c)

($x == 0$)

□ ¿Cómo se arregla? Volviendo para atrás...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
```

```
succs [] = []
succs (n:ns) = [ ] : succs ns
```

($n + 1$)

```
uppers :: [Char] -> [Char]
```

```
uppers [] = []
uppers (c:cs) = [ ] : uppers cs
```

(upper c)

```
tests :: [Int] -> [Bool]
```

```
tests [] = []
tests (x:xs) = [ ] : tests xs
```

($x == 0$)

□ ¿Cómo se arregla? Volviendo para atrás...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
sucess :: [Int] -> [Int]
success []      = [1]
success (n:ns) = n + 1 success 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? ...para separar bien las partes.

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
sucess :: [Int] -> [Int]
success []      = []
success (n:ns) = [ ] + n : success 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

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ 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...

Parámetros y esquemas de funciones

□ 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: nombrar los agujeros resultantes...

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

■ 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

```

Técnica extendida: ...y separarlos

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
succs :: [Int] -> [Int]
```

```
succs [] = []
```

```
succs (n:ns) = [ ] n : succs ns
```

$(\lambda x \rightarrow x + 1)$

```
uppers :: [Char] -> [Char]
```

```
uppers [] = []
```

```
uppers (c:cs) = [ ] c : uppers cs
```

$(\lambda x \rightarrow \text{upper } x)$

```
tests :: [Int] -> [Bool]
```

```
tests [] = []
```

```
tests (x:xs) = [ ] x : tests xs
```

$(\lambda n \rightarrow n == 0)$

□ Técnica extendida: ...y separarlos

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

(\x-> x + 1)

(\x-> upper x)

(\n-> n == 0)

□ Técnica extendida: ...y separarlos

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ 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: ...y separarlos

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
suucs :: [Int] -> [Int]
suucs []      = []
suucs (n:ns) = n : suucs ns
uppers :: [Char] -> [Char]
uppers []     = []
uppers (c:cs) = c : uppers cs
tests :: [Int] -> [Bool]
tests []      = []
tests (x:xs) = x : tests xs
```

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

□ Técnica extendida: Identificar las partes comunes...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
suucs :: [Int] -> [Int]
suucs [] = []
suucs (n:ns) = n : suucs ns
uppers :: [Char] -> [Char]
uppers [] = []
uppers (c:cs) = c : uppers cs
tests :: [Int] -> Bool
tests [] = False
tests (x:xs) = x > 0 && tests xs
```

suucs' = **(\x-> x + 1)**

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

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

□ Técnica extendida: Identificar las partes comunes...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

```
types :: [Tahar] -> Bofchar  
types [] = []  
types (n:ns) = [ ] n : types ns
```

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

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

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

□ Técnica extendida: Identificar las partes comunes...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

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

□ Técnica extendida: Identificar las partes comunes y nombrarlas

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

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

□ Técnica extendida: Nombrar el recuadro...

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

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

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

□ Técnica extendida: ...y ponerlo de parámetro

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

success' = **(\x-> x + 1)**

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

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

□ Técnica extendida: ...y ponerlo de parámetro

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

```
success' = map (\x-> x + 1)
```

```
uppers' = map (\x-> upper x)
```

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

□ Técnica extendida: Recuperar los datos con la función hecha

Parámetros y esquemas de funciones

□ 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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

□ ¿Qué tipo tiene el parámetro?

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

Parámetros y esquemas de funciones

□ Parámetros y funciones recursivas sobre listas

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

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

Esquema de map

□ Comparar las dos versiones

```
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)
```

```
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
```

- Las partes verdes y negras se redistribuyeron
- Las partes azules proveen las conexiones
- ¡Observar la importancia del polimorfismo!

Esquema de map

□ Comparar las dos versiones

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

succs' = map (+1)
uppers' = map upper
tests' = map (==0)
```

```
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
```

- A veces la notación puede confundir algunas cosas
- Se puede demostrar que **succs** = **succs'**, etc.

Esquema de map

□ *Esquema de map*

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

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

$\text{map } f [x_1, x_2, \dots, x_n] = ??$

Versión no currificada

- Dada una función y una lista, transforma esa lista elemento a elemento, según la función dada

Esquema de map

□ *Esquema de map*

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

`map f [] = []`

`map f (x:xs) = f x : map f xs`

Versión no currificada

`map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]`

- Dada una función y una lista, transforma esa lista elemento a elemento, según la función dada
- ¿Cómo se llama este esquema en Smalltalk y/o Java?

Esquema de map

□ *Esquema de map*

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

`map f [] = []`

`map f (x:xs) = f x : map f xs`

Versión no currificada

`map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]`

- Dada una función y una lista, transforma esa lista elemento a elemento, según la función dada
- ¿Cómo se llama este esquema en Smalltalk y/o Java?
 - Es el método **collect:** de las colecciones de Smalltalk y **map** de los Streams en Java

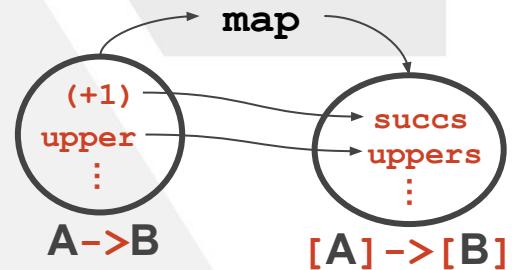
Esquema de map

□ Esquema de map

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

Versión currificada

- Interpretado en forma currificada, es más que **collect**:
 - Es una *función* que transforma funciones de elementos en funciones de listas

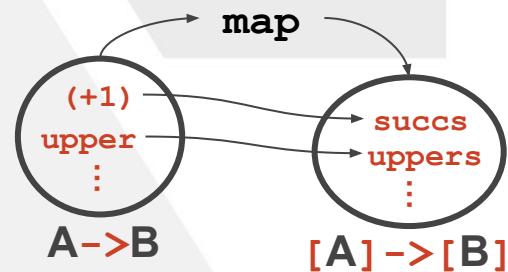


Esquema de map

□ Esquema de map

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

- Interpretado en forma currificada, es más que **collect**:
 - Es una *función* que transforma funciones de elementos en funciones de listas
 - No hay datos de orden 0 en este concepto



Esquema de map

□ *Esquema de map*

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

□ La función **map** permite ver en común a muchas otras

```
type Empresa = [Empleado]
sueldo :: Empleado -> Int
sueldos :: Empresa -> [Int]      -- Calcula el sueldo de cada empleado
sueldos ...
```

Esquema de map

Esquema de map

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

La función **map** permite ver en común a muchas otras

```
type Empresa = [Empleado]
sueldo :: Empleado -> Int
sueldos :: Empresa -> [Int]      -- Calcula el sueldo de cada empleado
sueldos = map sueldo
liquidaciones :: [Empresa] -> [[[Int]]] -- Liquida sueldos de todas
liquidaciones ...                  -- las empresas
```

Esquema de map

□ *Esquema de map*

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

□ La función **map** permite ver en común a muchas otras

```
type Empresa = [Empleado]
sueldo :: Empleado -> Int
sueldos :: Empresa -> [Int]      -- Calcula el sueldo de cada empleado
sueldos = map sueldo

liquidaciones :: [Empresa] -> [[[Int]]] -- Liquida sueldos de todas
liquidaciones = map sueldos           -- las empresas
```

Esquema de map

```
liquidaciones = map (map sueldo)
```

□ Esquema de map

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

□ La función **map** permite ver en común a muchas otras

```
type Empresa = [Empleado]
sueldo :: Empleado -> Int
sueldos :: Empresa -> [Int]      -- Calcula el sueldo de cada empleado
sueldos = map sueldo

liquidaciones :: [Empresa] -> [[[Int]]] -- Liquida sueldos de todas
liquidaciones = map sueldos           -- las empresas
```

Esquema de map

□ *Esquema de map*

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



- La función **map** permite ver en común a muchas otras
- Esta expresividad es una de las ventajas de los esquemas



Esquema de map

- *Esquema de map*: algunas propiedades

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

`map f [] = []`

`map f (x:xs) = f x : map f xs`

- Prop: ¿ `map id = id` ?
- Prop: ¿ para todo `f`. `length . map f = length` ?
- Prop: ¿ para todo `f`. para todo `xs`. para todo `ys`.
`map f (xs ++ ys) = map f xs ++ map f ys` ?

Esquema de map

- ❑ *Esquema de map*: algunas propiedades

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

`map f [] = []`

`map f (x:xs) = f x : map f xs`

- ❑ Las propiedades son verdaderas para todas las funciones que son instancia
- ❑ Esa es otra de las ventajas de los esquemas

Esquemas de funciones: filter

Parámetros y esquemas de funciones

- ❑ Generalización (**abstracción**) mediante **parametrización**
 - ❑ Es una forma de aprovechar el orden superior
- ❑ ¿Se podrá utilizar en funciones recursivas de forma útil?
 - ❑ Se definió el esquema de map
 - ❑ ¿Habrá otros esquemas interesantes?

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]    -- Solamente los números mayores que cero  
masQueCero ...
```

```
digitos :: [Char] -> [Char]      -- Solamente los caracteres que son dígitos  
digitos ...
```

```
noVacias :: [[a]] -> [[a]]        -- Solamente las listas que no están vacías  
noVacias ...
```

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]    -- Solamente los números mayores que cero
masQueCero []      = ...
masQueCero (n:ns) = ... n ... masQueCero ns ...
```

```
digitos :: [Char] -> [Char]    -- Solamente los caracteres que son dígitos
digitos []        = ...
digitos (c:cs)   = ... c ... digitos cs ...
```

```
noVacias :: [[a]] -> [[a]]    -- Solamente las listas que no están vacías
noVacias []       = ...
noVacias (xs:xss) = ... xs ... noVacias xss ...
```

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]    -- Solamente los números mayores que cero
masQueCero []      = ...
masQueCero (n:ns) = if n>0 then n : masQueCero ns
                    else masQueCero ns

digitos :: [Char] -> [Char]    -- Solamente los caracteres que son dígitos
digitos []          = ...
digitos (c:cs)      = if isDigit c then c : digitos cs
                    else digitos cs

noVacias :: [[a]] -> [[a]]      -- Solamente las listas que no están vacías
noVacias []          = ...
noVacias (xs:xss) = if null xs then noVacias xss
                    else xs : noVacias xss
```

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]    -- Solamente los números mayores que cero
masQueCero []      = []
masQueCero (n:ns) = if n>0 then n : masQueCero ns
                    else masQueCero ns

digitos :: [Char] -> [Char]    -- Solamente los caracteres que son dígitos
digitos []        = []
digitos (c:cs)   = if isDigit c then c : digitos cs
                    else digitos cs

noVacias :: [[a]] -> [[a]]    -- Solamente las listas que no están vacías
noVacias []       = []
noVacias (xs:xss) = if null xs then noVacias xss
                        else xs : noVacias xss
```

Repaso de la “técnica de los recuadros”

- ❑ **Técnica de los recuadros** para parametrizar (revisada)
 - ❑ **Paso 1:** recuadrar las diferencias
 - ❑ **Paso 2:** *aplicar la técnica a esos recuadros, para independizarlos*
 - ❑ **Paso 3:** recortar los recuadros y separarlos
 - ❑ **Paso 4:** identificar las partes comunes y nombrarlas
 - ❑ **Paso 5:** nombrar el recuadro y ponerlo de parámetro
 - ❑ **Paso 6:** recuperar los datos usando la función obtenida
- ❑ Generalización (**abstracción**) mediante **parametrización**

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if n > 0 then n : masQueCero ns
                    else masQueCero ns
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit c then c : digitos cs
                    else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if null xs then noVacias xss
                           else xs : noVacias xss
```

- Paso 1: se recuadran las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if n > 0 then n : masQueCero ns
                     else masQueCero ns
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit c then c : digitos cs
                     else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if null xs then noVacias xss
                           else xs : noVacias xss
```

¡Debe ajustarse el código para evidenciar la similaridad!

- Paso 1: se recuadran las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if n > 0
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit c
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if null xs
```

```
then n : masQueCero ns
else masQueCero ns
```

```
then c : digitos cs
else digitos cs
```

```
then noVacias xss
else xs : noVacias xss
```

¡Debe ajustarse el código para evidenciar la similaridad!

- Paso 1: recuadrar las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if n > 0 then n : masQueCero ns
                    else masQueCero ns
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit c then c : digitos cs
                    else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if not (null xs) then xs : noVacias xss
                                else noVacias xss
```

¡Debe ajustarse el código para evidenciar la similaridad!

- Paso 1: recuadrar las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if n > 0 then n : masQueCero ns
                    else masQueCero ns
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit c then c : digitos cs
                    else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if not (null xs) then xs : noVacias xss
                                else noVacias xss
```

- Paso 2: independizar los recuadros (paso 1)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) = if (n > 0) then n : masQueCero ns
                    else masQueCero ns
```

```
digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) = if isDigit(c) then c : digitos cs
                    else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) = if not (null(xs)) then xs : noVacias xss
                                else noVacias xss
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if ( ) > 0
```

n then n : masQueCero ns
else masQueCero ns

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if isDigit ( )
```

c then c : digitos cs
else digitos cs

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if not (null ( ))
```

xs then xs : noVacias xss
else noVacias xss

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if _____ > 0
```

```
n then n : masQueCero ns  
else masQueCero ns
```

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if _____ isDigit c
```

```
c then c : digitos cs  
else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if _____ not (null xs)
```

```
xs then xs : noVacias xss  
else noVacias xss
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if _____ > 0  
                      x
```

n then n : masQueCero ns
else masQueCero ns

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if _____ isDigit(c)  
                      x
```

c then c : digitos cs
else digitos cs

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if _____ not (null(xs))  
                           xs  
                           ys
```

xs then xs : noVacias xss
else noVacias xss

- Paso 2: independizar los recuadros (paso 5)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if (\x-> (x)> 0)
```

n then n : masQueCero ns
else masQueCero ns

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if (\x-> isDigit(x))
```

c then c : digitos cs
else digitos cs

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if (\ys-> not(null(ys)))
```

xs then xs : noVacias xss
else noVacias xss

- Paso 2: independizar los recuadros (paso 5)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if (\x-> x > 0)
```

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if (\x-> isDigit x)
```

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if (\ys-> not(null ys))
```

```
n then n : masQueCero ns  
else masQueCero ns
```

```
c then c : digitos cs  
else digitos cs
```

```
xs then xs : novacias xss  
else novacias xss
```

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero []      = []
```

```
masQueCero (n:ns) = if
```



```
(\x-> x > 0)
```

```
n then n : masQueCero ns  
else masQueCero ns
```

```
digitos :: [Char] -> [Char]
```

```
digitos []      = []
```

```
digitos (c:cs) = if
```



```
(\x-> isDigit x)
```

```
c then c : digitos cs  
else digitos cs
```

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias []      = []
```

```
noVacias (xs:xss) = if
```



```
(\ys-> not(null ys))
```

```
xs then xs : noVacias xss  
else noVacias xss
```

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
masQueCero :: [Int] -> [Int]
```

```
masQueCero [] = []
```

```
masQueCero (n:ns) = if [ ] n then n : masQueCero ns  
                      else masQueCero ns
```

(\x-> x > 0)

```
digitos :: [Char] -> [Char]
```

```
digitos [] = []
```

```
digitos (c:cs) = if [ ] c then c : digitos cs  
                      else digitos cs
```

(\x-> isDigit x)

```
noVacias :: [[a]] -> [[a]]
```

```
noVacias [] = []
```

```
noVacias (xs:xss) = if [ ] xs then xs : noVacias xss  
                           else noVacias xss
```

(\ys-> not(null ys))

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
filter :: [a] -> [a]
filter [] = []
filter (x:xs) = if x then x : filter xs
                else filter xs
```

(\x-> x > 0)

(\x-> isDigit x)

(\ys-> not(null ys))

- Paso 4: identificar las partes comunes y nombrarlas

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
filter :: [a] -> [a]
filter [] = []
filter (x:xs) = if x then x : filter xs
                else filter xs
```

(\x-> x > 0)

(\x-> isDigit x)

(\ys-> not(null ys))

- Paso 5: nombrar el recuadro y ponerlo de parámetro

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
filter :: ??      -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

(\x-> x > 0)

(\x-> isDigit x)

(\ys-> not(null ys))

- Paso 5: nombrar el recuadro y ponerlo de parámetro

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
filter :: ??      -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

```
masQueCero' = filter (\x-> x > 0)
digitos'    = filter (\x-> isDigit x)
noVacias'  = filter (\ys-> not(null ys))
```

- Paso 6: recuperar los datos usando la función obtenida

Esquema de filter

- Definir las siguientes funciones sobre listas

```
filter :: ??      -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

```
masQueCero' = filter (\x-> x > 0)
digitos'    = filter (\x-> isDigit x)
noVacias'   = filter (\ys-> not(null ys))
```

- ¿Qué tipo tiene el parámetro?

Esquema de filter

- Definir las siguientes funciones sobre listas

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

```
masQueCero' = filter (\x-> x > 0)
digitos'     = filter (\x-> isDigit x)
noVacias'   = filter (\ys-> not(null ys))
```

- ¿Qué tipo tiene el parámetro?
 - Nuevamente, una función

Esquema de filter

□ *Esquema de filter*

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs

masQueCero' = filter (\x-> x > 0)
digitos'    = filter (\x-> isDigit x)
noVacias'   = filter (\ys-> not(null ys))
```

- El esquema de filter elimina elementos que no cumplen la condición dada

Esquema de filter

□ Comparar las dos versiones

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) =
    if p x then x : filter p xs
              else filter p xs

masQueCero' = filter (\x-> x > 0)
digitos'    = filter (\x-> isDigit x)
noVacias'   = filter (\ys-> not(null ys))
```

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) =
    if n>0 then n : masQueCero ns
              else masQueCero ns

digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) =
    if isDigit c then c : digitos cs
              else digitos cs

noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) =
    if null xs then noVacias xss
              else xs : noVacias xss
```

- Las partes verdes y negras se redistribuyeron
- Las partes azules proveen las conexiones
- ¡Observar la importancia del polimorfismo!

Esquema de filter

□ Comparar las dos versiones

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) =
  if p x then x : filter p xs
            else filter p xs

masQueCero' = filter (>0)
digitos'    = filter isDigit
noVacias'   = filter (not . null)
```

```
masQueCero :: [Int] -> [Int]
masQueCero []      = []
masQueCero (n:ns) =
  if n>0 then n : masQueCero ns
            else masQueCero ns

digitos :: [Char] -> [Char]
digitos []      = []
digitos (c:cs) =
  if isDigit c then c : digitos cs
            else digitos cs

noVacias :: [[a]] -> [[a]]
noVacias []      = []
noVacias (xs:xss) =
  if null xs then noVacias xss
            else xs : noVacias xss
```

- La notación puede ocultar ciertas ideas
- Se puede demostrar que **masQueCero = masQueCero'**

Esquema de filter

□ *Esquema de filter*

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) =
  if p x then x : filter p xs
            else filter p xs
```

Versión no currificada

- Dada una función de comparación y una lista, describir la lista con aquellos elementos que cumplan la propiedad
- ¿Cómo se llama este esquema en Smalltalk y/o Java?
 - Es el método **select**: de las colecciones de Smalltalk y el **filter** de los Streams de Java

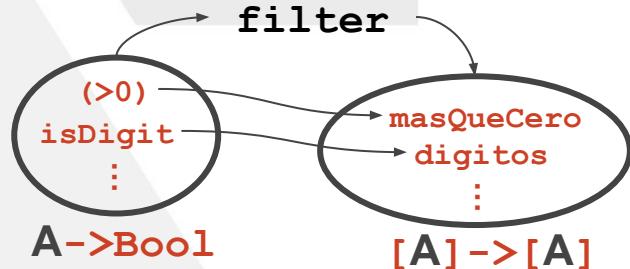
Esquema de filter

□ Esquema de filter

```
filter :: (a->Bool) -> ([a] -> [a])
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

Versión currificada

- Interpretado en forma currificada, es más que **select**:
 - Es una *función* que transforma propiedades sobre elementos en funciones de listas
 - No hay elementos de orden 0 en este concepto



Esquema de filter

□ Esquema de filter

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

□ La función filter permite ver en común a muchas otras

```
type Empresa = [Empleado]
ausencias :: Empleado -> Int
losSarmientos :: Empresa -> [Empleado]      -- Solamente empleados sin ausencias
losSarmientos ...
```

Esquema de filter

□ Esquema de filter

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

□ La función filter permite ver en común a muchas otras

```
type Empresa = [Empleado]
ausencias :: Empleado -> Int
losSarmientos :: Empresa -> [Empleado]
losSarmientos = filter ((==0) . ausencias)
sinPresentismos :: [Empresa] -> [Empresa] -- Empresas, pero sin personal sin ausencias
sinPresentismos ...
```

\emp -> ausencias emp == 0

-- Solamente empleados sin ausencias

Esquema de filter

□ Esquema de filter

```
filter :: (a->Bool) -> [a] -> [a]
```

```
filter p []      = []
```

```
filter p (x:xs) = if p x then x : filter p xs  
                   else filter p xs
```

```
\e -> null (losSarmientos e)
```

□ La función filter permite ver en común a muchas otras

```
type Empresa = [Empleado]
```

```
ausencias :: Empleado -> Int
```

```
losSarmientos :: Empresa -> [Empleado]      -- Solamente empleados sin ausencias
```

```
losSarmientos = filter ((==0) . ausencias)
```

```
sinPresentismos :: [Empresa] -> [Empresa] -- Empresas, pero sin personal sin ausencias
```

```
sinPresentismos = filter (null . losSarmientos)
```

Esquema de filter

```
filter (null . (filter ((==0) . ausencias)))
```

□ Esquema de filter

```
filter :: (a->Bool) -> [a] -> [a]
```

```
filter p [] = []
```

```
filter p (x:xs) = if p x then x : filter p xs  
                   else filter p xs
```

```
\e -> null (losSarmientos e)
```

□ La función filter permite ver en común a muchas otras

```
type Empresa = [Empleado]
```

```
ausencias :: Empleado -> Int
```

```
losSarmientos :: Empresa -> [Empleado]      -- Solamente empleados sin ausencias
```

```
losSarmientos = filter ((==0) . ausencias)
```

```
sinPresentismos :: [Empresa] -> [Empresa] -- Empresas, pero sin personal sin ausencias
```

```
sinPresentismos = filter (null . losSarmientos)
```

Esquema de filter

□ Esquema de filter

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

- La función filter permite ver en común a muchas otras
- Una ventaja de los esquemas: **expresividad**



Esquema de filter

□ *Esquema de filter*: algunas propiedades

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

- Prop: $\text{filter} (\text{const True}) = \text{id}$?
- Prop: $\text{para todo } p. \text{length} . \text{filter } p \leq \text{length}$?
- Prop: $\text{para todo } p. \text{para todo } xs. \text{para todo } ys.$
 $\text{filter } p (xs ++ ys) = \text{filter } p xs ++ \text{filter } p ys$?
- Prop: $\text{para todo } p. \text{para todo } f.$
 $\text{filter } (p . f) = \text{filter } p . \text{map } f$?

Esquema de filter

□ *Esquema de filter*: algunas propiedades

```
filter :: (a->Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

- Una vez más, las propiedades son verdaderas para todas las funciones que son instancia
- Otra de las ventajas: *propiedades generales*

Esquema de recursión estructural

Parámetros y esquemas de funciones

- ❑ Generalización (*abstracción*) mediante **parametrización**
 - ❑ Es una forma de aprovechar el orden superior
- ❑ ¿Se podrá utilizar en funciones recursivas de forma útil?
 - ❑ Se definieron los esquemas de **map** y **filter**
 - ❑ ¿Habrá otros esquemas interesantes?

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool          -- Indica si todos los números son 5
sonCincos ...
```

```
cantTotal :: [[a]] -> Int           -- Cuántos elementos hay en total
cantTotal ...
```

```
concat :: [[a]] -> [a]              -- El append de todas las listas dadas
concat ...
```

- Se decide usar recursión estructural

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool          -- Indica si todos los números son 5
sonCincos []           = ...
sonCincos (n:ns)       = ... n ... sonCincos ns ...

cantTotal :: [[a]] -> Int            -- Cuántos elementos hay en total
cantTotal []           = ...
cantTotal (xs:xss)     = ... xs ... cantTotal xss ...

concat :: [[a]] -> [a]              -- El append de todas las listas dadas
concat []             = ...
concat (xs:xss)       = ... xs ... concat xss ...
```

- Se plantea el esquema

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool          -- Indica si todos los números son 5
sonCincos []           = ...
sonCincos (n:ns)       = n==5 && sonCincos ns

cantTotal :: [[a]] -> Int            -- Cuántos elementos hay en total
cantTotal []           = ...
cantTotal (xs:xss)     = length xs + cantTotal xss

concat :: [[a]] -> [a]              -- El append de todas las listas dadas
concat []             = ...
concat (xs:xss)       = xs ++ concat xss
```

- Se definen los casos inductivos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool          -- Indica si todos los números son 5
sonCincos []           = True
sonCincos (n:ns)       = n==5 && sonCincos ns

cantTotal :: [[a]] -> Int            -- Cuántos elementos hay en total
cantTotal []            = 0
cantTotal (xs:xss)      = length xs + cantTotal xss

concat :: [[a]] -> [a]               -- El append de todas las listas dadas
concat []              = []
concat (xs:xss)        = xs ++ concat xss
```

- Se completa con los casos base (elementos neutros)

Repaso de la “técnica de los recuadros”

- ❑ ¿Se podrá encontrar un esquema en este caso?
- ❑ Recordar: **Técnica de los recuadros** (revisada)
 - ❑ **Paso 1:** recuadrar las diferencias
 - ❑ **Paso 2:** aplicar la técnica a esos recuadros, para independizarlos
 - ❑ **Paso 3:** recortar los recuadros y separarlos
 - ❑ **Paso 4:** identificar las partes comunes que quedaron
 - ❑ **Paso 5:** nombrar el recuadro y ponerlo de parámetro
 - ❑ **Paso 6:** recuperar los datos usando la función obtenida
- ❑ Generalización (**abstracción**) mediante **parametrización**

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = n==5 && sonCincos ns
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = length xs + cantTotal xss
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

- Paso 1: recuadrar las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = n==5 && sonCincos ns
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = length xs + cantTotal xss
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

¡Se recuadró
TODO!
¿Se parecen
en algo?

- Paso 1: recuadrar las diferencias

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = n==5 && sonCincos ns
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = length xs + cantTotal xss
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

¡Falta
independizar
los recuadros!

- Paso 2: independizar los recuadros (paso 1)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = (n==5 && sonCincos ns)
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = length xs + cantTotal xss
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True n sonCincos ns
```

```
sonCincos (n:ns) = (n==5 && sonCincos ns)
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0 xs cantTotal xss
```

```
cantTotal (xs:xss) = length xs + cantTotal xss
```

```
concat :: [[a]] -> [a]
```

```
concat [] = [] xs concat xss
```

```
concat (xs:xss) = xs ++ concat xss
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = (n==5 && sonCincos ns)
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = (length xs + cantTotal xss)
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = (xs ++ concat xss)
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = ( ==5 && ( ) ) n (sonCincos ns)
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = ( length( ) + ( ) ) xs (cantTotal xss)
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = ( ( ) ++ ( ) ) xs (concat xss)
```

- Paso 2: independizar los recuadros (paso 3)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = ( ( ==5 && ( ) ) n (sonCincos ns) )
```

x b

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = ( length( ) + ( ) ) xs (cantTotal xss)
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = ( ( ) ++ ( ) ) ys zs xs (concat xss)
```

- Paso 2: independizar los recuadros (paso 5)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = (x == 5 && b) n (sonCincos ns)
```

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = (length ys + n) xs (cantTotal xss)
```

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = (ys ++ zs) xs (concat xss)
```

- Paso 2: independizar los recuadros (paso 5)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
```

```
sonCincos [] = True
```

```
sonCincos (n:ns) = (\x b-> x==5 && b)
```

n (sonCincos ns)

```
cantTotal :: [[a]] -> Int
```

```
cantTotal [] = 0
```

```
cantTotal (xs:xss) = (\ys n-> length ys + n)
```

xs (cantTotal xss)

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = (\ys zs->(ys)++(zs))
```

xs (concat xss)

- Paso 2: independizar los recuadros (paso 5)

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool
sonCincos [] = True
sonCincos (n:ns) = (\x b-> x==5 && b) n (sonCincos ns)

cantTotal :: [[a]] -> Int
cantTotal [] = 0
cantTotal (xs:xss) = (\ys n-> length ys + n) xs (cantTotal xss)

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = (\ys zs-> ys ++ zs) xs (concat xss)
```

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
sonCincos :: [Int] -> Bool True
sonCincos [] = λ b -> x == 5 && b
sonCincos (n:ns) = λ b -> n (sonCincos ns)

cantTotal :: [[a]] -> Int 0
cantTotal [] = λ ys -> length ys + n
cantTotal (xs:xss) = λ ys -> xs (cantTotal xss)

concat :: [[a]] -> [a] []
concat [] = λ ys -> ys ++ zs
concat (xs:xss) = λ ys -> xs (concat xss)
```

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`sonCincos :: [Int] -> Bool`

`sonCincos []`

`sonCincos (n:ns)`

`cantTotal :: [[a]] -> Int`

`cantTotal []`

`cantTotal (xs:xss)`

`concat :: [[a]] -> [a]`

`concat []`

`concat (xs:xss)`

`True`

`(\x b-> x==5 && b)`

`n (sonCincos ns)`

`0`

`(\ys n-> length ys + n)`

`xs (cantTotal xss)`

`[]`

`(\ys zs-> ys ++ zs)`

`xs (concat xss)`

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`sonCincos :: [Int] -> Bool`

`sonCincos []`

= 

`True`

`(\x b-> x==5 && b)`

`sonCincos (n:ns)`

= 

`n (sonCincos ns)`

`cantTotal :: [[a]] -> Int`

`cantTotal []`

= 

`0`

`(\ys n-> length ys + n)`

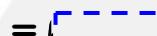
`cantTotal (xs:xss)`

= 

`xs (cantTotal xss)`

`concat :: [[a]] -> [a]`

`concat []`

= 

`[]`

`(\ys zs-> ys ++ zs)`

`concat (xs:xss)`

= 

`xs (concat xss)`

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`sonCincos :: [Int] -> Bool`

`sonCincos []`

= []

True

$(\lambda x \ b \rightarrow x == 5 \ \&\& \ b)$

`sonCincos (n:ns)`

= []

n (sonCincos ns)

`cantTotal :: [[a]] -> Int`

`cantTotal []`

= []

0

$(\lambda ys \ n \rightarrow \text{length } ys + n)$

`cantTotal (xs:xss)`

= []

xs (cantTotal xss)

`concat :: [[a]] -> [a]`

`concat []`

= []

[]

$(\lambda ys \ zs \rightarrow ys ++ zs)$

`concat (xs:xss)`

= []

xs (concat xss)

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`sonCincos :: [Int] -> Bool`

`sonCincos []`

=
[]

`(\x b-> x==5 && b)`

`True`

`sonCincos (n:ns)`

=
n (sonCincos ns)

`cantTotal :: [[a]] -> Int`

`cantTotal []`

=
[]

`(\ys n-> length ys + n)`

`0`

`cantTotal (xs:xss)`

=
xs (cantTotal xss)

`concat :: [[a]] -> [a]`

`concat []`

=
[]

`(\ys zs-> ys ++ zs)`

`[]`

`concat (xs:xss)`

=
xs (concat xss)

- Paso 3: recortar los recuadros y separarlos

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

foldr :: [a] -> b

foldr [] = λ []

foldr (x:xs) = λ [] x (foldr xs)

(\x b-> x==5 && b)

True

(\ys n-> length ys + n)

0

(\ys zs-> ys ++ zs)

[]

- Paso 4: identificar las partes comunes y nombrarlas

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

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

(\x b-> x==5 && b)	True
(\ys n-> length ys + n)	0
(\ys zs-> ys ++ zs)	[]

- Paso 5: nombrar los recuadros y ponerlos de parámetro

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`foldr :: [a] -> b`

`foldr [] = z`

`foldr (x:xs) = f x (foldr xs)`

<code>(\x b-> x==5 && b)</code>	<code>True</code>
<code>(\ys n-> length ys + n)</code>	<code>0</code>
<code>(\ys zs-> ys ++ zs)</code>	<code>[]</code>

- Paso 5: nombrar los recuadros y ponerlos de parámetro

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

`foldr :: ?? -> ?? -> [a] -> b`

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

<code>(\x b-> x==5 && b)</code>	<code>True</code>
--	-------------------

<code>(\ys n-> length ys + n)</code>	<code>0</code>
---	----------------

<code>(\ys zs-> ys ++ zs)</code>	<code>[]</code>
-------------------------------------	-----------------

- Paso 5: nombrar los recuadros y ponerlos de parámetro

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
foldr :: ??      -> ?? -> [a] -> b
```

```
foldr f z []     = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

```
sonCincos' =      (\x b-> x==5 && b) True
```

```
cantTotal' =      (\ys n-> length ys + n) 0
```

```
concat' =         (\ys zs-> ys ++ zs) []
```

- Paso 6: recuperar los datos usando la función obtenida

Parámetros y esquemas de funciones

- Definir las siguientes funciones sobre listas

```
foldr :: ??      -> ?? -> [a] -> b
```

```
foldr f z []     = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

```
sonCincos' = foldr (\x b-> x==5 && b) True
```

```
cantTotal' = foldr (\ys n-> length ys + n) 0
```

```
concat'     = foldr (\ys zs-> ys ++ zs) []
```

- Paso 6: recuperar los datos usando la función obtenida

Esquema de recursión estructural

□ *Esquema de recursión estructural (foldr)*

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

sonCincos' = foldr (\x  b-> x==5 && b)      True
cantTotal' = foldr (\ys n-> length ys + n) 0
concat'    = foldr (\ys zs-> ys ++ zs) []
```

□ ¿Qué tipo tienen los parámetros? ¡Son dos!

Esquema de recursión estructural

□ *Esquema de recursión estructural (foldr)*

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

sonCincos' = foldr (\x  b-> x==5 && b)      True
cantTotal' = foldr (\ys  n-> length ys + n) 0
concat'    = foldr (\ys zs-> ys ++ zs)        []
```

- ¿Qué tipo tienen los parámetros? ¡Son dos!
 - Una función...

Esquema de recursión estructural

□ *Esquema de recursión estructural (foldr)*

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

sonCincos' = foldr (\x  b-> x==5 && b)      True
cantTotal' = foldr (\ys  n-> length ys + n) 0
concat'    = foldr (\ys zs-> ys ++ zs)        []
```

- ¿Qué tipo tienen los parámetros? ¡Son dos!
 - Una función y un valor “base”

Esquema de recursión estructural

□ Esquema de recursión estructural (funciones)

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

```
sonCincos' = foldr (\x b-> x==5 && b) True
```

```
cantTotal' = foldr (\ys n-> length ys + n) 0
```

```
concat' = foldr (\ys zs-> ys ++ zs) []
```

```
sonCincos :: [Int] -> Bool
sonCincos []      = True
sonCincos (n:ns)  =
    n==5 && sonCincos ns

cantTotal :: [[a]] -> Int
cantTotal []      = 0
cantTotal (xs:xss) =
    length xs + cantTotal xss

concat :: [[a]] -> [a]
concat []          = []
concat (xs:xss)   =
    xs ++ concat xss
```

- Las partes verdes y negras se redistribuyeron
- Las partes azules proveen las conexiones
- ¡Observar la importancia del polimorfismo!

Esquema de recursión estructural

❑ Esquema de recursión estructural (foldr)

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

sonCincos' = foldr ((&&) . (==5)) True
cantTotal' = foldr ((+) . length) 0
concat'    = foldr (++) []
```

```
sonCincos :: [Int] -> Bool
sonCincos []      = True
sonCincos (n:ns)  =
  n==5 && sonCincos ns

cantTotal :: [[a]] -> Int
cantTotal []      = 0
cantTotal (xs:xss) =
  length xs + cantTotal xss

concat :: [[a]] -> [a]
concat []          = []
concat (xs:xss)   =
  xs ++ concat xss
```

- ❑ La notación puede ocultar ciertas ideas
- ❑ Se puede demostrar que **cantTotal = cantTotal'**

Esquema de recursión estructural

Versión no currificada

❑ *Esquema de recursión estructural (foldr)*

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

- ❑ Dada una función de incorporación de elementos, un valor básico y una lista, describe el resultado de procesar e incorporar todos los elementos de esa lista al valor básico
- ❑ ¿Cómo se llama este esquema en Smalltalk y/o Java?

Esquema de recursión estructural

Versión no currificada

❑ *Esquema de recursión estructural (foldr)*

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

- ❑ Dada una función de incorporación de elementos, un valor básico y una lista, describe el resultado de procesar e incorporar todos los elementos de esa lista
- ❑ ¿Cómo se llama este esquema en Smalltalk y/o Java?
 - ❑ Es el método **inject:into:** de las colecciones de Smalltalk y el **reduce** de los Streams de Java

Esquema de recursión estructural

Versión “más” currificada

❑ *Esquema de recursión estructural (foldr)*

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

- ❑ Interpretada currificada, es más que **inject:into:**
 - ❑ Es una función que produce funciones recursivas estructurales (en DOS pasos)
 - ❑ *¡La función foldr ES la expresión en código de la noción de recursión estructural sobre listas!*

Esquema de recursión estructural

Versión “más” currificada

❑ *Esquema de recursión estructural (foldr)*

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

- ❑ Interpretada currificada, es más que **inject:into:**
 - ❑ Es una función que produce funciones recursivas estructurales (en DOS pasos)
 - ❑ *¡La función foldr ES la expresión en código de la noción de recursión estructural sobre listas!*

Esquema de recursión estructural

❑ *Esquema de recursión estructural (**foldr**)*

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

foldr f z [x₁, x₂, ..., x_n] = ??

❑ O sea, ¡TODAS las funciones recursivas estructurales sobre listas pueden expresarse como resultado de un **foldr**!

Esquema de recursión estructural

❑ Esquema de recursión estructural (*foldr*)

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

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

`foldr f z [x1, x2, ..., xn] = f x1 (f x2 (... f xn z ...))`

La función **f**
asocia a derecha
en esta operación

❑ O sea, ¡TODAS las funciones recursivas estructurales sobre listas pueden expresarse como resultado de un **foldr**!

- ❑ El nombre foldr viene de “doblar” (en el sentido de *plegar, to fold*) desde la derecha (*right*): fold-r (*to fold to the right*)
- ❑ También se la conoce como **reduce**, pues *reduce* la lista dada

Esquema de recursión estructural

❑ Esquema de recursión estructural (`foldr`)

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

$\text{foldr } f \ z \ [x_1, x_2, \dots, x_n] = f \ x_1 \ (f \ x_2 \ (\dots \ f \ x_n \ z \ \dots))$

La función `f` asocia a derecha en esta operación

❑ La función `foldr` permite ver en común a muchas más

```
sum      = foldr (+) 0
```

```
prod     = foldr (*) 1
```

```
all      = foldr (&&) True
```

```
concat   = foldr (++) []
```

```
map f    = foldr ... (???)
```

```
filter p = foldr ... (???)
```

```
cantTotal = foldr ((+) . length) 0
```

```
sonCincos = foldr ((&&) . (==5)) True
```

```
sumSqr   = foldr ((+) . (^2)) 0
```

```
nFaltas  = foldr ((+) . ausencias) 0
```

```
gSearch q = foldr ((++) . cumplen q) []
```

```
append   = foldr ... (???)
```

Esquema de recursión es



Esquema de recursión estructural

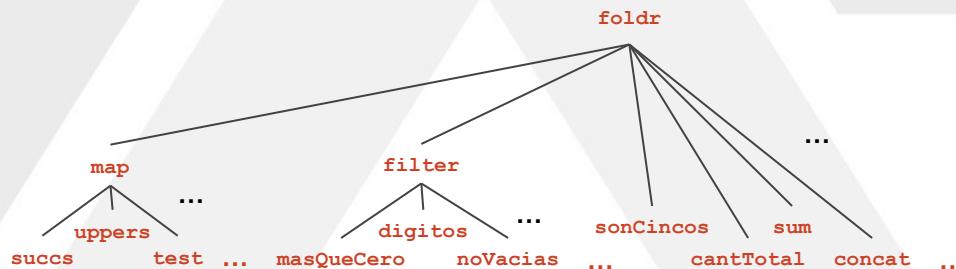
`foldr :: (a -> b -> b) -> b -> [a]`

foldr f z [] = z

`foldr f z (x:xs) = f x (foldr f z xs)`

foldr $f z [x_1, x_2, \dots, x_n] = f x_1 (f x_2 (\dots f x_n z \dots))$

La función **foldr** permite ver en común a muchas más



Ventajas de los esquemas

- ❑ Los esquemas tienen un montón de ventajas
 - ❑ **Expresividad**
 - ❑ Permiten definir muchísimas funciones en forma clara y concisa
 - ❑ **Modularidad**
 - ❑ Permiten la definición de pequeños módulos que pueden combinarse para obtener soluciones más complejas
 - ❑ **Propiedades generales**
 - ❑ Permiten demostrar propiedades que son válidas para TODAS las instancias del esquema

Propiedades del esquema de recursión estructural

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ Pero usa recursión estructural...

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

```
append :: [a] -> [a] -> [a]
append []      ys = ys
append (x:xs) ys = x : append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ Pero usa recursión estructural...

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

```
append :: [a] -> [a] -> [a]
append []      ys = ys
append (x:xs) ys = x : append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ Se explicita que el resultado es una función

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

```
append :: [a] -> ([a] -> [a])
append []        = \ys -> ys
append (x:xs)   = \ys -> x : append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ Y se procede con la técnica de los recuadros

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

```
append :: [a] -> ([a] -> [a])
append []      = \ys -> ys
append (x:xs) = \ys -> x : append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ Y se procede con la técnica de los recuadros
 - ❑ ¡El 2do argumento del caso recursivo es una función!

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

```
append :: [a] -> ([a] -> [a])
append []      = \ys -> ys
append (x:xs) = \ys -> x : append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ El 2do argumento del caso recursivo es una función
 - ❑ Es esperable, pues es el resultado recursivo

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

```
append :: [a] -> ([a] -> [a])
append [] = \ys -> ys
append (x:xs) = \ys -> x :: append xs ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ El 2do argumento del caso recursivo es una función
 - ❑ Es esperable, pues es el resultado recursivo

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

```
append :: [a] -> ([a] -> [a])
append [] = \ys -> ys
append (x:xs) = (\x h-> \ys -> x : h ys) x (append xs)
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ El 2do argumento del caso recursivo es una función
 - ❑ Es esperable, pues es el resultado recursivo

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

```
append :: [a] -> ([a] -> [a])
append []      = \ys -> ys
append (x:xs) = (\x h-> \ys -> x : h ys) x (append xs)
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ El caso recursivo queda con 3 parámetros...
 - ❑ ¿Es correcto eso?

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

```
append :: [a] -> ([a] -> [a])
```

```
append []      = \ys -> ys
```

```
append (x:xs) = (\x h-> \ys -> x : h ys) x (append xs)
```

```
append' :: [a] -> [a] -> [a]
```

```
append' = foldr (\x h-> \ys -> x : h ys) (\ys -> ys)
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Es posible expresar **append** usando **foldr**?
 - ❑ ¡El resultado es una función!
 - ❑ El caso recursivo queda con 3 parámetros...
 - ❑ Es correcto, porque el resultado es una función

```
append :: [a] -> ([a] -> [a])
```

```
append [] = \ys -> ys
```

```
append (x:xs) = (\x h-> \ys -> x : h ys) x (append xs)
```

```
append' :: [a] -> [a] -> [a]
```

```
append' = foldr (\x h-> \ys -> x : h ys) (\ys -> ys)  
          :: a :: b           :: b           :: b
```

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

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace

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

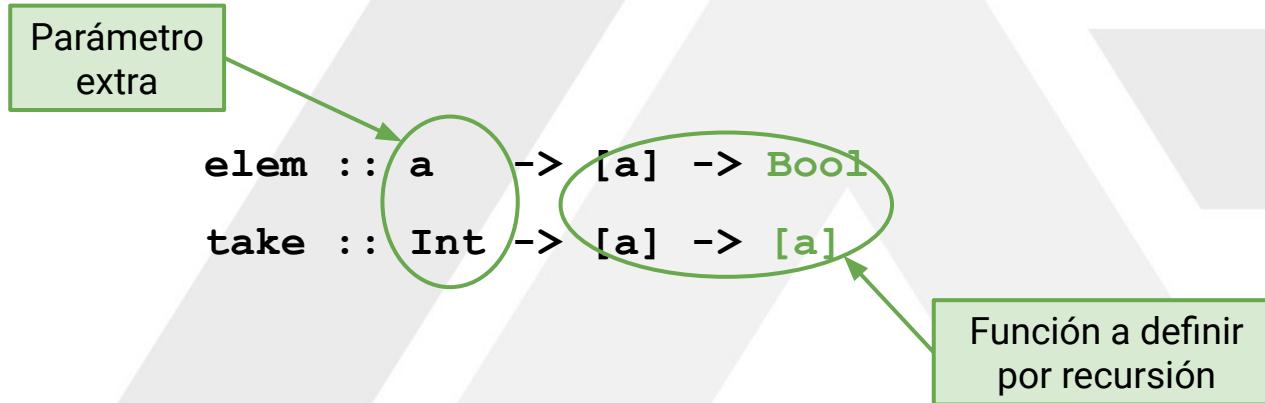
```
elem :: a    -> [a] -> Bool
```

```
take :: Int -> [a] -> [a]
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace



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

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que no cambia:

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

```
elem :: a -> ([a] -> Bool)
elem x []      = False
elem x (y:ys) = x==y || elem x ys
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que no cambia:
 - ❑ El parámetro extra se usa en forma “global”

```
elem :: a -> ([a] -> Bool)
elem x []      = False
elem x (y:ys) = x==y || elem x ys

elem' :: a -> [a] -> Bool
elem' x = foldr (\y b-> x==y || b) False
```

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

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que no cambia:
 - ❑ El parámetro extra se usa en forma “global”

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

```
elem :: a -> ([a] -> Bool) ←
elem x []      = False
elem x (y:ys) = x==y || elem x ys
elem' :: a -> [a] -> Bool
elem' x = foldr (\y b-> x==y || b) False
```

La función que se define es
 $(\text{elem } x)(y \text{ NO } \text{elem})$

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que cambia:

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

```
take :: Int -> ([a] -> [a])
take n []      = []
take n (x:xs) = if n==0 then [] else x : take (n-1) xs
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que cambia:
 - ❑ El parámetro extra se pasa al resultado (el foldr es de orden superior)

```
take :: Int -> ([a] -> [a])
take n []      = []
take n (x:xs) = if n==0 then [] else x : take (n-1) xs
take :: Int -> [a] -> [a]
take = flip (foldr ctk (\n-> []))
where ctk x h = \n-> if n==0 then [] else x : h (n-1)
```

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

Propiedades del esquema de *foldr*

```
flip :: (a -> b -> c) -> b -> a -> c  
flip f x y = f y x
```

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

❑ Expresividad

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que cambia:
 - ❑ El parámetro extra se pasa al resultado (el foldr es de orden superior)

Observar el uso de
flip

```
take :: Int -> ([a] -> [a])  
take n []      = []  
take n (x:xs) = if n==0 then [] else x : take (n-1) xs  
  
take :: Int -> [a] -> [a]  
take = flip (foldr ctk (\n-> []))  
where ctk x h = \n-> if n==0 then [] else x : h (n-1)
```

:: [a] -> $\overbrace{\text{Int} -> [\text{a}]}^b$

```
flip :: (a -> b -> c) -> b -> a -> c  
flip f x y = f y x
```

Propiedades del esquema de *foldr*

❑ *Expresividad*

- ❑ ¿Y si el parámetro extra no es parte del resultado?
 - ❑ Hay dos posibilidades: no cambia nunca, o sí lo hace
 - ❑ En el caso que cambia:
 - ❑ El parámetro extra se pasa al resultado (el foldr es de orden superior)

```
take :: Int -> ([a] -> [a])  
take n []      = []  
take n (x:xs) = if n==0 then [] else x : take (n-1) xs
```

```
take :: Int -> [a] -> [a] ::b  
take = flip (foldr ctk (\n-> []))  
where ctk x h = \n-> if n==0 then [] else x : h (n-1)  
      ::a ::b  
      ::b
```

```
:: [a] -> Int -> [a]  
::b
```

Propiedades del esquema de *foldr*

□ Propiedades generales

- Considerar la definición dada de la función **cantTotal**

```
cantTotal :: [[a]] -> Int
cantTotal []          = 0
cantTotal (xs:xss)   = length xs + cantTotal xss
```

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

Propiedades del esquema de *foldr*

□ Propiedades generales

- Considerar la definición dada de la función **cantTotal**

```
cantTotal :: [[a]] -> Int
cantTotal []          = 0
cantTotal (xs:xss)   = length xs + cantTotal xss

cantTotal = foldr (\xs n-> length xs + n) 0
cantTotal = foldr ((+) . length) 0
```

- Se busca analizar su eficiencia, y ver cómo mejorarla
- Es mejor pensar con una versión más concreta...

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
```

```
monedasEnElGrupo [] = 0
```

```
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps
```

```
monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
```

```
monedasEnElGrupo = foldr ((+) . monedas) 0
```

- Se busca analizar su eficiencia, y ver cómo mejorarla
- Es mejor pensar con una versión más concreta...

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
monedasEnElGrupo []      = 0
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps

monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
monedasEnElGrupo = foldr ((+) . monedas) 0
```

- ¿Cuánto se tardaría en contar siguiendo esta estrategia?
 - Suponer que una persona tarda 1 min en contar sus monedas

Propiedades del esquema de *foldr*

❑ Propiedades generales

❑ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
```

```
monedasEnElGrupo [] = 0
```

```
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps
```

```
monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
```

```
monedasEnElGrupo = foldr ((+) . monedas) 0
```

- ❑ ¿Cuánto se tardaría en contar siguiendo esta estrategia?
 - ❑ Suponer que una persona tarda 1 min en contar sus monedas
 - ❑ ¡Tantos minutos como personas haya!

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
monedasEnElGrupo []      = 0
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps

monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
monedasEnElGrupo = foldr ((+) . monedas) 0
```

□ ¿Se puede hacer más rápido?

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
```

```
monedasEnElGrupo [] = 0
```

```
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps
```

```
monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
```

```
monedasEnElGrupo = foldr ((+) . monedas) 0
```

□ ¿Se puede hacer más rápido?

- ¡Sí! ¡Que todos cuenten al mismo tiempo, y sumar aparte!

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar la definición dada de `monedasEnElGrupo`

```
monedasEnElGrupo :: [Persona] -> Int
```

```
monedasEnElGrupo [] = 0
```

```
monedasEnElGrupo (p:ps) = monedas p + monedasEnElGrupo ps
```

```
monedasEnElGrupo = foldr (\p n-> monedas p + n) 0
```

```
monedasEnElGrupo = foldr ((+) . monedas) 0
```

□ ¿Se puede hacer más rápido?

- ¡Sí! ¡Que todos cuenten al mismo tiempo, y sumar aparte!
- ¿Cómo se expresaría esa estrategia?

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar definición alternativa de `monedasEnElGrupo`

```
monedasEnElGrupo' :: [Persona] -> Int
monedasEnElGrupo' ps = sum ( ?? ps)
  where sum = foldr (+) 0
        todosCuentanSusMonedas
```

¿`monedasEnElGrupo' = monedasEnElGrupo`?

`monedasEnElGrupo = foldr ((+) . monedas) 0`

□ ¿Se puede hacer más rápido?

- ¡Sí! ¡Que todos cuenten al mismo tiempo, y sumar aparte!
- ¿Cómo se expresaría esa estrategia?

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar definición alternativa de `monedasEnElGrupo`

```
monedasEnElGrupo' :: [Persona] -> Int
monedasEnElGrupo' ps = sum (map monedas ps)
  where sum = foldr (+) 0
         todosCuentanSusMonedas
```

¿`monedasEnElGrupo' = monedasEnElGrupo`?

`monedasEnElGrupo = foldr ((+) . monedas) 0`

□ Un `map` seguido de un `foldr (reduce)`

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar definición alternativa de `monedasEnElGrupo`

```
monedasEnElGrupo' :: [Persona] -> Int
monedasEnElGrupo' ps = sum (map monedas ps)
  where sum = foldr (+) 0
         todosCuentanSusMonedas
```

¿`monedasEnElGrupo' = monedasEnElGrupo`?

`monedasEnElGrupo = foldr ((+) . monedas) 0`

□ Un `map` seguido de un `foldr (reduce)`

- Esta estrategia se conoce como “*map-reduce*”
- El `map` podría parallelizarse, y así mejoraría la velocidad

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

Propiedades del esquema de *foldr*

□ Propiedades generales

□ Considerar definición alternativa de `monedasEnElGrupo`

```
monedasEnElGrupo' :: [Persona] -> Int
monedasEnElGrupo' ps = sum (map monedas ps)
  where sum = foldr (+) 0
         todosCuentanSusMonedas
```

¿`monedasEnElGrupo' = monedasEnElGrupo`?

`monedasEnElGrupo = foldr ((+) . monedas) 0`

□ ¿Y esta forma es equivalente a la anterior?

□ Debe demostrarse

□ ¡Este esquema puede demostrarse en forma general!

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

Propiedades del esquema de *foldr*

□ Propiedades generales

- Prop: para todo f. para todo g. para todo z.

foldr (f . g) z = foldr f z . map g

- Corolario: (consecuencia de la propiedad)

monedasEnElGrupo' = foldr (+) 0 . map monedas

- Dem: aplicar la propiedad a la siguiente equivalencia

monedasEnElGrupo = foldr ((+) . monedas) 0



Propiedades del esquema de *foldr*

- ❑ *Propiedades generales*: más propiedades
 - ❑ Prop: para todo f. para todo g. para todo z.
$$\text{foldr } (f \ . \ g) \ z = \text{foldr } f \ z \ . \ \text{map } g$$
 - ❑ Toda función definida por **foldr** termina
 - ❑ Prop: $\text{foldr } (:) \ [] = \text{id}$ -- :: [a] -> [a]
 - ❑ Prop: para todo f. para todo z.
para todo xs. para todo ys.
$$\begin{aligned} \text{foldr } f \ z \ (xs \ ++ \ ys) \\ = \text{foldr } f \ (\text{foldr } f \ z \ ys) \ xs \end{aligned}$$

Esquema de recursión primitiva

Recursión primitiva

- ❑ No todas las funciones recursivas son estructurales...

```
insert :: a -> [a] -> [a] -- El elemento agregado en orden
insert x []      = [x]
insert x (y:ys) = if x <= y
                  then x : y : ys
                  else y : insert x ys

maximum :: [a] -> a          -- El máximo de los elementos
maximum []       = error "No hay elementos"
maximum (x:xs)  = if null xs
                  then x
                  else x `max` maximum xs
```

- ❑ ¡No se pueden definir con **foldr**!

Recursión primitiva

- ❑ ¿Cuál es el problema para expresarlas con **foldr**?

```
insert :: a -> [a] -> [a] -- El elemento agregado en orden
```

```
insert x [] = [x]
```

```
insert x (y:ys) = if x <= y  
                  then x : y : ys  
                  else y : insert x ys
```

```
maximum :: [a] -> a -- El máximo de los elementos
```

```
maximum [] = error "No hay elementos"
```

```
maximum (x:xs) = if null xs  
                   then x  
                   else x `max` maximum xs
```

No son parte del esquema, pero tampoco de los recuadros

- ❑ ¡Vuelve a haber parámetros que pierden su ligadura!

Recursión primitiva

- Solución: incluir esos datos en el esquema

```
insert :: a -> [a] -> [a] -- El elemento agregado en orden
```

```
insert x [] = [x]
```

```
insert x (y:ys) = if x <= y  
then x : y : ys  
else y : insert x ys
```

Ahora pasan a ser parte del esquema

```
maximum :: [a] -> a -- El máximo de los elementos
```

```
maximum [] = error "No hay elementos"
```

```
maximum (x:xs) = if null xs  
then x  
else x `max` maximum xs
```

- El segundo recuadro tendrá ahora 3 argumentos

Esquema de recursión primitiva

□ *Esquema de recursión primitiva (recr)*

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

- Es un esquema similar al de **foldr**
- Para no mezclarlos, el orden de los parámetros es inverso
- La función parámetro ahora tiene un parámetro más

Esquema de recursión primitiva

□ *Esquema de recursión primitiva (recr)*

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

□ Se pueden expresar bien las funciones anteriores

```
insert x = recr [x] (\y ys rs -> if x < y
                        then x:y:ys
                        else y:rs)
maximum   = recr (error "") (\x xs m -> if null xs
                           then x
                           else max x m)
```

Esquema de recursión primitiva

□ *Esquema de recursión primitiva (recr)*

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

□ Este esquema permite ver en común a muchas más aún

```
init      = recr (error "") (\x xs rs -> if null xs
                           then []
                           else x:rs)
tail     = recr (error "") (\_ xs _  -> xs)

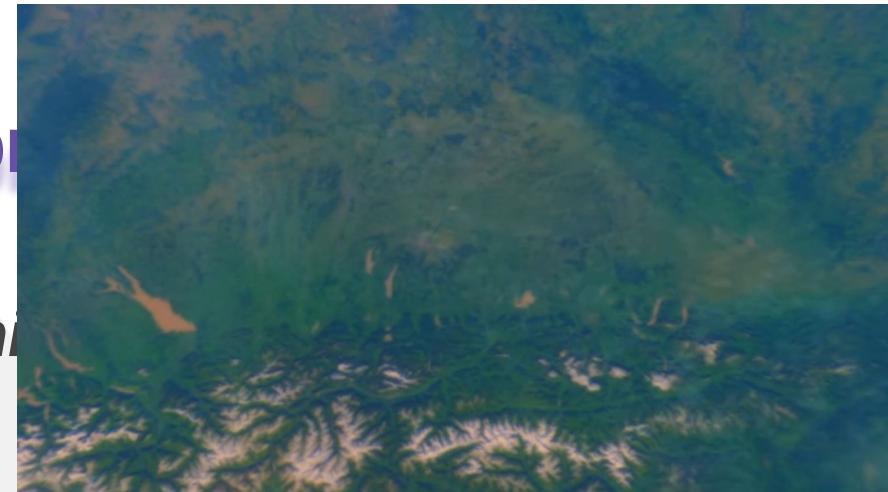
foldr f z = recr ... (???)
```

Esquema de recursión primaria

□ *Esquema de recursión primaria*

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

- Este esquema permite ver en común a muchas más aún



Esquema de recursión primitiva

□ *Esquema de recursión primitiva (recr)*

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

Hay que elegir una de las variantes (cualquiera sirve)

- Es sencillo definir **foldr** usando **recr**
- Lo sorprendente es que es posible escribir **recr** usando **foldr** más algún ajuste
 - Se puede ajustar de varias formas

```
recr z f = h (foldr ????) where h =
```

{\f x -> f x x
appDup
fst
snd
...}

John C. Reynolds



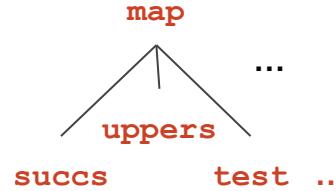
John Charles Reynolds

(1 de junio 1935 – 28 de abril 2013) es un científico de la computación estadounidense que enseñó computación en la Universidad de Carnegie Mellon.

Se focalizó en el diseño de lenguajes de programación y también de lenguajes de especificación. Fue el inventor del lambda cálculo polimórfico (System F – descubierto en forma independiente por Jean Yves Girard) y formuló el concepto semántico de parametricidad. También trabajó en el concepto de *continuaciones*, una forma de representar datos en base a las maneras de usarlos en el futuro, implementadas mediante funciones de orden superior. Trabajó en la lógica de separación para expresar corrección de programas con estructuras mutables de datos que comparten memoria.

Esquemas y la abstracción

Abstracción



- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura

Abstracción



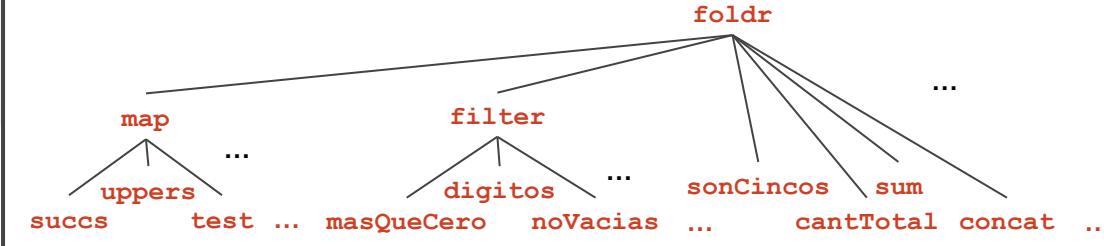
- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura

Abstracción



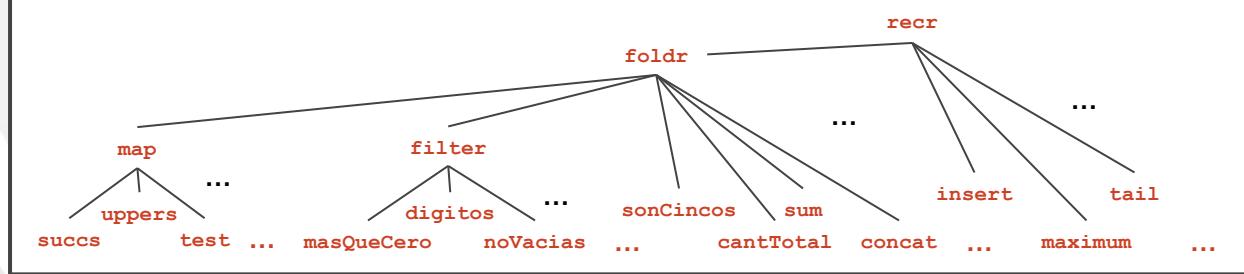
- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura

Abstracción



- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura

Abstracción



- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura
 - ❑ Metafóricamente, es como hacer “zoom out” en un terreno, subiendo en forma vertical

Parámetros y la “técnica

- ❑ ¿Qué es un parámetro? ¿Cómo se usa?
- ❑ *Técnica de los recuadros para*

$$a = 1 + 1$$

$$b = 1 + 16$$

$$c = 1 + 41$$

$$s = \text{y} \rightarrow 1 + \text{y}$$

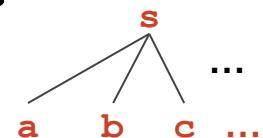
$$a' = s \ 1$$

$$b' = s \ 16$$

$$c' = s \ 41$$



- ❑ Se puede demostrar que $a = a'$, $b = b'$ y $c = c'$
 - ❑ Se definió una *función s*



Parámetros y órdenes su

- Parámetros y funciones cur
- Técnica de los recuadros

$$f \ y = 2 + y$$

$$g \ y = 17 + y$$

$$h \ y = 42 + y$$

$$\text{suma } x \ y = x + y$$

$$f' = \text{suma } 2$$

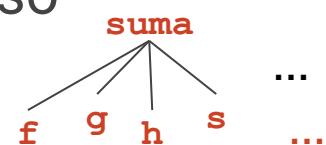
$$g' = \text{suma } 17$$

$$h' = \text{suma } 42$$



- La notación puede dificultar visualizar el proceso

- Se puede demostrar que $f = f'$, $g = g'$ y $h = h'$



Esquema de map

□ *Esquema de map*

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



- La función **map** permite ver en común a muchas otras
- Esta expresividad es una de las ventajas de los esquemas



Esquema de recursión estructural

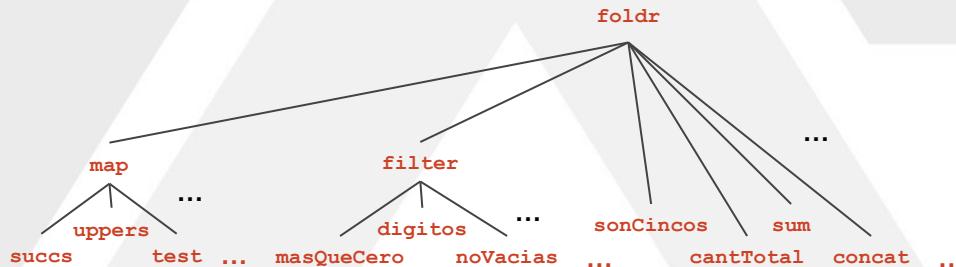
□ Esquema de recursión estructural

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

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

□ La función foldr permite ver en común a muchas más



Esquema de recursión primaria



❑ Esquema de recursión primaria

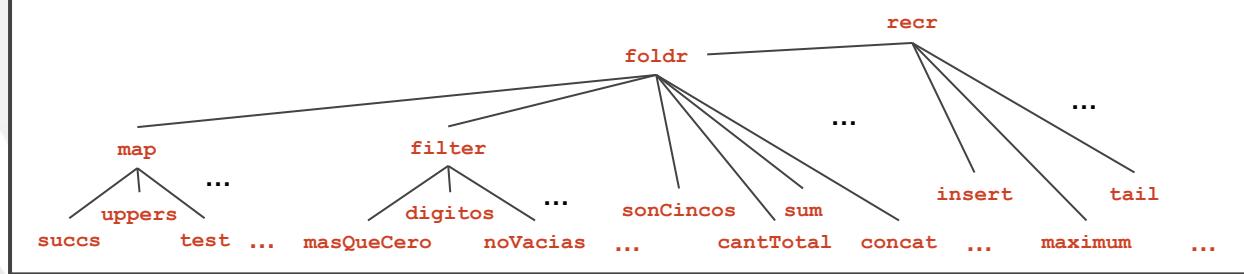
```
recr :: b -> (a->[a]->b->b) ->  
recr z f []      = z  
recr z f (x:xs) = f x xs (recr z f xs)
```

❑ La función recr permite ver en común a muchas más aún





Abstracción



- ❑ Abstraer es detectar similaridades (ignorando diferencias) y aprovecharlas
- ❑ Los parámetros son una forma sintáctica de expresar abstracción
- ❑ Es como “subir” y mirar desde mayor altura
 - ❑ ¿Se podrá subir más aún? ¿Cuánto?
 - ❑ ¿Hay otras formas sintácticas para expresar abstracción?
 - ❑ Esto sigue en la próxima clase...

Resumen

Resumen

- ❑ Parametrización
- ❑ Esquemas de funciones
 - ❑ map, filter
 - ❑ propiedades
- ❑ Esquemas de recursión
 - ❑ foldr, recr
 - ❑ propiedades

