

# Lenguajes, Tecnologías y Paradigmas de la programación (LTP)

## Práctica 5: Listas y Tipos algebraicos (Parte 1: Listas)



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

Sergio Pérez  
[serperu@dsic.upv.es](mailto:serperu@dsic.upv.es)

# SESIÓN 1: Listas

## OBJETIVOS DE LA PRÁCTICA

- Definir funciones básicas para el manejo de listas
- Introducir el orden superior (mediante `map` y `filter`)

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

`foo :: Int -> ...`

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

`foo :: [Int] -> ...`

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

```
foo :: [Int] -> ...
```

```
foo x = ...
```

```
foo (x:xs) = ...
```

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

`foo :: [Int] -> ...`

`foo [1, 2, 3]`

`foo x = ...`

`foo (x:xs) = ...`



# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

`foo :: [Int] -> ...`

`foo x = ...`

→

`foo [1, 2, 3]`

`x = [1, 2, 3]`

`foo (x:xs) = ...`

# Representación de una lista

- Representación entre corchetes: `[1, 2, 3, 4, 5]`
- ¿Cómo recibo listas en una función?

`foo :: [Int] -> ...`

`foo x = ...`

→

`foo [1, 2, 3]`

`x = [1, 2, 3]`

`foo (x:xs) = ...`

→

`x = 1`

`xs = [2, 3]`

# Operaciones con listas

- Operador “++” (Concatenar listas):      **Sintaxis:** (lista) ++ (lista)

# Operaciones con listas

- Operador “++” (Concatenar listas):      Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 =`

# Operaciones con listas

- Operador “++” (Concatenar listas):      Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l =`

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l = elem : l`



# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l = elem : l`

*¿Como lo añado al final?*

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l = elem : l`

*¿Como lo añado al final? ¿l : elem?*

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l = elem : l`

*¿Como lo añado al final? ¿l : elem?*

**!!!NO!!!** Violaría la sintaxis

# Operaciones con listas

- Operador “++” (Concatenar listas): Sintaxis: (lista) ++ (lista)

`union :: [Int] -> [Int] -> [Int]`

`union l1 l2 = l1 ++ l2`

- Operador “:” (Formar listas cabeza-cola): Sintaxis: elemento : (lista)

`addFirst :: Int -> [Int] -> [Int]`

`addFirst elem l = elem : l`

*¿Como lo añado al final? ¿l : elem?*

**!!!NO!!!** Violaría la sintaxis

**SOLUCIÓN:** `l ++ [elem]`

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)  
Sintaxis: (lista) !! (int)

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1, 2, 3] !! 2 →

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1, 2, 3] !! 2 → 3

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1, 2, 3] !! 2 → 3

- Operador “..” para crear listas (finitas o infinitas)



# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1, 2, 3] !! 2 → 3

- Operador “..” para crear listas (finitas o infinitas)

[1 .. 5] →

[1 .. ] →

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1,2,3] !! 2 → 3

- Operador “..” para crear listas (finitas o infinitas)

[1..5] → [1,2,3,4,5]

[1..] →

# Operaciones con listas

- Operador “!!” (Devolver el elemento de una posición de la lista)

Sintaxis: (lista) !! (int)

[1,2,3] !! 2 → 3

- Operador “..” para crear listas (finitas o infinitas)

[1..5] → [1,2,3,4,5]

[1..] → [1,2,3,4,5,6,...]

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

`long [] = 0` (1)

`long (x:xs) = 1 + long xs` (2)

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

`long [] = 0` (1)

`long (x:xs) = 1 + long xs` (2)

`long [1,2,3]`

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

`long [] = 0` (1)

`long (x:xs) = 1 + long xs` (2)

`long [1,2,3]`

`(2) → x = 1, xs = [2,3]`


# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

`long [] = 0` (1)

`long (x:xs) = 1 + long xs` (2)

`long [1,2,3]`  
  
`1 + long [2,3]`

(2)  $\rightarrow x = 1, xs = [2,3]$



# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$

$\text{long } [1,2,3]$



$1 + \text{long } [2,3]$

$(2) \rightarrow x = 1, xs = [2,3]$

$(2) \rightarrow x = 2, xs = [3]$

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$

$$\begin{array}{c} \text{long } [1,2,3] \\ \underbrace{\hspace{1.5cm}} \\ 1 + \text{long } [2,3] \\ \underbrace{\hspace{1.5cm}} \\ 1 + \text{long } [3] \end{array}$$

$$(2) \rightarrow x = 1, xs = [2,3]$$

$$(2) \rightarrow x = 2, xs = [3]$$

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

`long [] = 0` (1)

`long (x:xs) = 1 + long xs` (2)

`long [1,2,3]`  
└──┬──┘  
1 + `long [2,3]`  
    └──┬──┘  
    1 + `long [3]`

(2) → `x = 1, xs = [2,3]`

(2) → `x = 2, xs = [3]`

(2) → `x = 3, xs = []`

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$

$\text{long } [1,2,3]$

$1 + \text{long } [2,3]$

$1 + \text{long } [3]$

$1 + \text{long } []$

$(2) \rightarrow x = 1, xs = [2,3]$

$(2) \rightarrow x = 2, xs = [3]$

$(2) \rightarrow x = 3, xs = []$

# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$

$\text{long } [1,2,3]$	$(2) \rightarrow x = 1, xs = [2,3]$
$1 + \text{long } [2,3]$	$(2) \rightarrow x = 2, xs = [3]$
$1 + \text{long } [3]$	$(2) \rightarrow x = 3, xs = []$
$1 + \text{long } []$	$(1) \rightarrow 0$

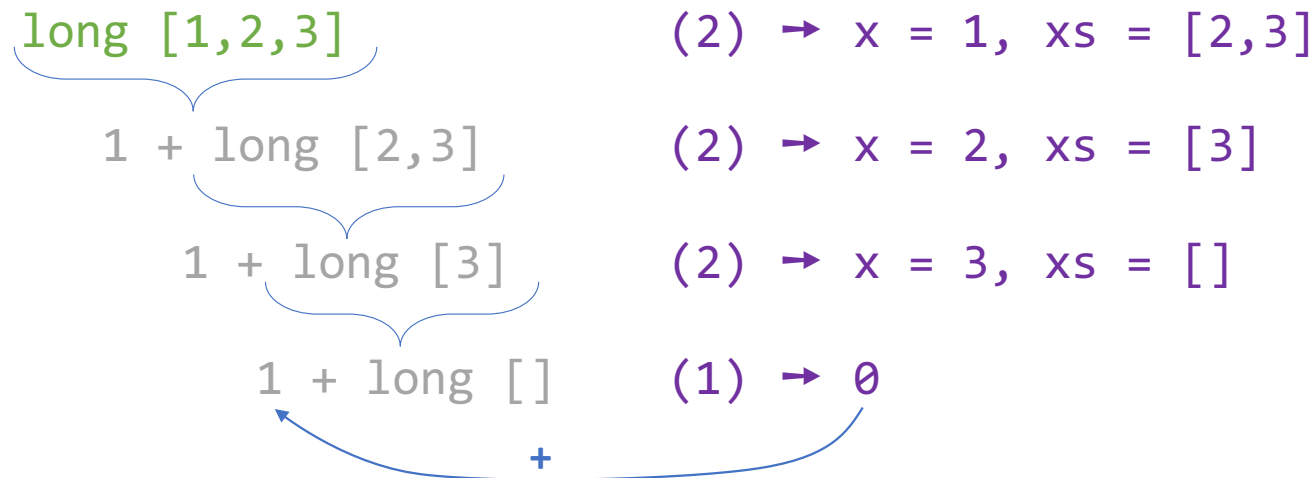
# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$



# Definición de funciones para listas

## Forma general

### Ejemplo: Función que calcula la longitud de una lista

$$\text{long } [] = \emptyset \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$

Diagram illustrating the recursive calculation of the length of the list `[1, 2, 3]`.

**Left side (Recursive Calls):**

- `long [1,2,3]` (initial call)
- `1 + long [2,3]` (call from `long [1,2,3]`)
- `1 + long [3]` (call from `1 + long [2,3]`)
- `1 + long []` (call from `1 + long [3]`)

**Right side (Return Values):**

- `(2) → x = 1, xs = [2,3]` (return from `long [2,3]`)
- `(2) → x = 2, xs = [3]` (return from `long [3]`)
- `(2) → x = 3, xs = []` (return from `long []`)
- `(1) → 0` (return from `long []`)

Blue arrows indicate the flow of data (return values) from the recursive calls back to the callers.

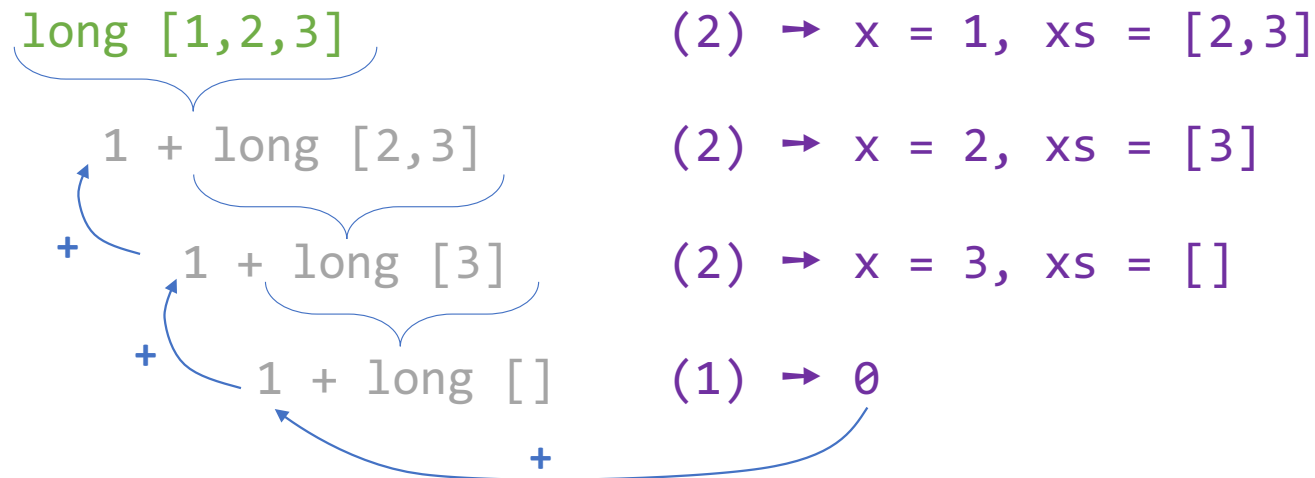
# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$





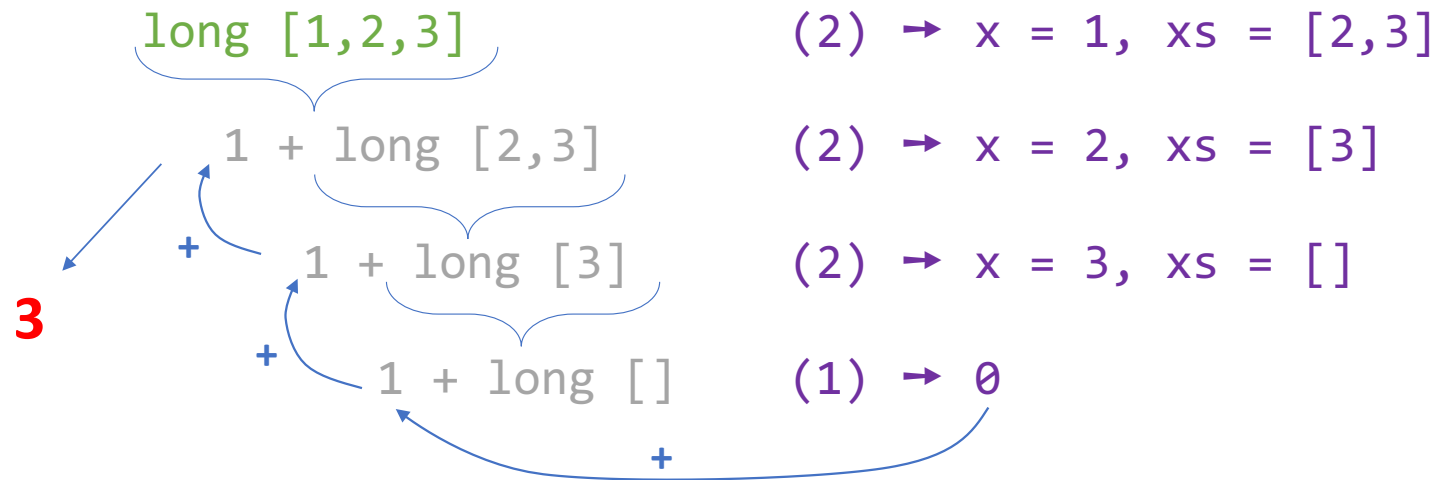
# Definición de funciones para listas

## Forma general

*Ejemplo: Función que calcula la longitud de una lista*

$$\text{long } [] = 0 \quad (1)$$

$$\text{long } (x:xs) = 1 + \text{long } xs \quad (2)$$



# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

expresión

generador

filtro(s)

**Sintaxis:** [ *expresión* | *variable* <- *lista* , *condicion(es)* ]

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

expresión

generador

filtro(s)

**Sintaxis:** [ *expresión* | *variable* <- *lista* , *condicion(es)* ]

¿Como se ejecuta?

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

expresión

generador

filtro(s)

**Sintaxis:** [ *expresión* | *variable* <- *lista* , *condicion(es)* ]

¿Como se ejecuta?

1) **GENERADOR:** Cojo 1 elemento de *lista* y lo mete en *variable*

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

expresión

generador

filtro(s)

**Sintaxis:** [ *expresión* | *variable* <- *lista* , *condicion(es)* ]

### ¿Como se ejecuta?

- 1) **GENERADOR:** Cojo 1 elemento de *lista* y lo mete en *variable*
- 2) **FILTRO(S):** Evalúa si se cumplen la(s) *condicion(es)*

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

expresión

generador

filtro(s)

**Sintaxis:** [ *expresión* | *variable* <- *lista* , *condicion(es)* ]

### ¿Como se ejecuta?

- 1) **GENERADOR:** Cojo 1 elemento de *lista* y lo mete en *variable*
- 2) **FILTRO(S):** Evalúa si se cumplen la(s) *condicion(es)*
- 3) **EXPRESIÓN:** Si *condicion(es)* se cumple, se ejecuta *expresión* y se guarda el resultado en la nueva lista resultado

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

Element 1: before = [], x = 1 → x \* x = 1 → after = [1]



# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

Element 1: before = [], x = 1 → x \* x = 1 → after = [1]

Element 2: before = [1], x = 2 → x \* x = 4 → after = [1,4]

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

Element 1: before = [], x = 1 → x \* x = 1 → after = [1]

Element 2: before = [1], x = 2 → x \* x = 4 → after = [1,4]

Element 3: before = [1,4], x = 3 → x \* x = 9 → after = [1,4,9]

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

Element 1: before = [], x = 1 → x \* x = 1 → after = [1]

Element 2: before = [1], x = 2 → x \* x = 4 → after = [1,4]

Element 3: before = [1,4], x = 3 → x \* x = 9 → after = [1,4,9]

Element 4: before = [1,4,9], x = 4 → x \* x = 16 → after = [1,4,9,16]

# Definición de funciones para listas

## Listas Intensionales (*list comprehensions*)

*Ejemplo: Eleva al cuadrado los elementos de una lista*

```
> [x * x | x <- [1,2,3,4]]  
[1,4,9,16]
```

Element 1: before = [], x = 1 → x \* x = 1 → after = [1]

Element 2: before = [1], x = 2 → x \* x = 4 → after = [1,4]

Element 3: before = [1,4], x = 3 → x \* x = 9 → after = [1,4,9]

Element 4: before = [1,4,9], x = 4 → x \* x = 16 → after = [1,4,9,16]

Result = [1,4,9,16]

# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

`(*2)` es una función que se aplicará a todos los elementos

# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

`(*2)` es una función que se aplicará a todos los elementos

Element 1: `before = [1,2,3] → 1 (*2) = 2 → after = [2,2,3]`

# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

`(*2)` es una función que se aplicará a todos los elementos

Element 1: before = [1,2,3] → 1 (\*2) = 2 → after = [2,2,3]

Element 2: before = [2,2,3] → 2 (\*2) = 4 → after = [2,4,3]



# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

`(*2)` es una función que se aplicará a todos los elementos

Element 1: before = [1,2,3] → 1 (\*2) = 2 → after = [2,2,3]

Element 2: before = [2,2,3] → 2 (\*2) = 4 → after = [2,4,3]

Element 3: before = [2,4,3] → 3 (\*2) = 6 → after = [2,4,6]

# Definición de funciones para listas

## Orden superior (*higher order*)

*Ejemplo: Función que multiplica por 2 los elementos de una lista*

```
> map (*2) [1..3]  
[2,4,6]
```

`(*2)` es una función que se aplicará a todos los elementos

Element 1: before = [1,2,3] → 1 (\*2) = 2 → after = [2,2,3]

Element 2: before = [2,2,3] → 2 (\*2) = 4 → after = [2,4,3]

Element 3: before = [2,4,3] → 3 (\*2) = 6 → after = [2,4,6]

Result = [2,4,6]

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

`map' ::`

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

`map' :: (a -> b)`

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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



# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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

`map' f [] = []`

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

# Definición de funciones para listas

- Definiendo funciones de orden superior
  - *Función que recibe una función y una lista y aplica la función a la lista (map')*

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

`map' f [] = []`

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

# Ejercicios

- Ejercicios Parte 1:
  - Ejercicios 1 - 10
- Ampliación Parte 1:
  - Ejercicios 17, 18 y 19