

Programación Funcional en Haskell

Primera parte

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

25 de marzo de 2025

Repaso: usando GHCi

Cómo empezar:

```
$
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci  
Loading ...  
Prelude>
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded:  Main.
*Main>
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded:  Main.
*Main>
```

Otros comandos útiles:

- Para recargar: `:r`
- Para cargar otro archivo: `:l archivo.hs`
- Para conocer el tipo de una expresión: `:t True`

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

¿Qué hacen estas funciones?

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

¿Qué hacen estas funciones?

Podría decirse que ambas “toman dos argumentos (x, y) y devuelven su producto”. Pero esto no es del todo así...

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- prod recibe una **tupla** de dos elementos.

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- prod recibe una tupla de dos elementos.
- ¿¿Y prod'??

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- `prod` recibe una tupla de dos elementos.
- `prod'` es una función que **toma un `x`** de tipo `Int` y **devuelve una función** de tipo `Int -> Int`, cuyo comportamiento es tomar un entero y multiplicarlo por `x`.

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- prod recibe una tupla de dos elementos.
- prod' es una función que **toma un x** de tipo Int y **devuelve una función** de tipo Int -> Int, cuyo comportamiento es tomar un entero y multiplicarlo por x .
En particular, (prod' 2) es la función que duplica.

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- `prod` recibe una tupla de dos elementos.
- `prod'` es una función que **toma un `x`** de tipo `Int` y **devuelve una función** de tipo `Int -> Int`, cuyo comportamiento es tomar un entero y multiplicarlo por `x`.

En particular, `(prod' 2)` es la función que duplica.

Una definición equivalente de `prod'` usando funciones anónimas:

```
prod' x = \y -> x*y
```

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- `prod` recibe una tupla de dos elementos.
- `prod'` es una función que **toma un `x`** de tipo `Int` y **devuelve una función** de tipo `Int -> Int`, cuyo comportamiento es tomar un entero y multiplicarlo por `x`.

En particular, `(prod' 2)` es la función que duplica.

Una definición equivalente de `prod'` usando funciones anónimas:

```
prod' x = \y -> x*y
```

Decimos que `prod'` es la versión **currificada** de `prod`.

Ejercicio

Definir las siguientes funciones:

- 1 $\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$
que devuelve la versión currificada de una función no currificada.
- 2 $\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)$
que devuelve la versión no currificada de una función currificada.

Los paréntesis en gris no son necesarios, pero es útil escribirlos cuando estamos aprendiendo y queremos ver más explícitamente que estamos devolviendo una función.

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
```

```
prod x y = x * y
```

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
```

```
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
```

```
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?
- 2 ¿Qué pasa si cambiamos la definición por `doble = prod 2`?

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
```

```
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?
- 2 ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
- 3 ¿Qué significa `(+) 1`?

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
```

```
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?
- 2 ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
- 3 ¿Qué significa `(+) 1`?
- 4 Definir las siguientes funciones de forma similar a `(+) 1`:

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int  
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?
- 2 ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
- 3 ¿Qué significa `(+) 1`?
- 4 Definir las siguientes funciones de forma similar a `(+)`1:
 - `triple :: Float -> Float`

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int  
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- ❶ ¿Cuál es el tipo de `doble`?
- ❷ ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
- ❸ ¿Qué significa `(+) 1`?
- ❹ Definir las siguientes funciones de forma similar a `(+) 1`:
 - `triple :: Float -> Float`
 - `esMayorDeEdad :: Int -> Bool`

Ejercicios

- 1 Implementar y dar los tipos de las siguientes funciones:

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- ❶ Implementar y dar los tipos de las siguientes funciones:
 - ❶ `(.)` que compone dos funciones. Por ejemplo:
 $((\backslash x \rightarrow x * 4) . (\backslash y \rightarrow y - 3))\ 10$ devuelve 28.

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- ❶ Implementar y dar los tipos de las siguientes funciones:
 - ❶ `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - ❷ `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- ❶ Implementar y dar los tipos de las siguientes funciones:
 - ❶ `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - ❷ `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.
 - ❸ `($)` que aplica una función a un argumento. Por ejemplo:
`id $ 6` devuelve 6.

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- ❶ Implementar y dar los tipos de las siguientes funciones:
 - ❶ `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - ❷ `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.
 - ❸ `($)` que aplica una función a un argumento. Por ejemplo:
`id $ 6` devuelve 6.
 - ❹ `const` que, dado un valor, retorna una función constante que devuelve siempre ese valor. Por ejemplo:
`const 5 'casa'` devuelve 5.

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- 1 Implementar y dar los tipos de las siguientes funciones:
 - a `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - b `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.
 - c `($)` que aplica una función a un argumento. Por ejemplo:
`id $ 6` devuelve 6.
 - d `const` que, dado un valor, retorna una función constante que devuelve siempre ese valor. Por ejemplo:
`const 5 'casa'` devuelve 5.
- 2 ¿Qué hace `flip ($)` 0?

Pueden ver más funciones útiles en la sección Útil del Campus.

Ejercicios

- ❶ Implementar y dar los tipos de las siguientes funciones:
 - ❶ `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - ❷ `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.
 - ❸ `($)` que aplica una función a un argumento. Por ejemplo:
`id $ 6` devuelve 6.
 - ❹ `const` que, dado un valor, retorna una función constante que devuelve siempre ese valor. Por ejemplo:
`const 5 'casa'` devuelve 5.
- ❷ ¿Qué hace `flip ($) 0`?
- ❸ ¿Y `(==0) . (flip mod 2)`?

Pueden ver más funciones útiles en la sección Útil del Campus.

Hay varias **macros** para definir listas:

Hay varias **macros** para definir listas:

- **Por extensión**

Esto es, dar la lista explícita, escribiendo todos sus elementos.

Por ejemplo: [4, 3, 3, 4, 6, 5, 4, 5, 4, 5].

Hay varias **macros** para definir listas:

- **Por extensión**

Esto es, dar la lista explícita, escribiendo todos sus elementos.
Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**

Son progresiones aritméticas en un rango particular.
Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

Hay varias **macros** para definir listas:

- **Por extensión**

Esto es, dar la lista explícita, escribiendo todos sus elementos.
Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**

Son progresiones aritméticas en un rango particular.
Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

- **Por comprensión**

Se definen de la siguiente manera:

`[expresión | selectores, condiciones]`

Por ejemplo: `[(x,y) | x <- [0..5], y <- [0..3], x+y==4]`

Hay varias **macros** para definir listas:

- **Por extensión**

Esto es, dar la lista explícita, escribiendo todos sus elementos.
Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**

Son progresiones aritméticas en un rango particular.
Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

- **Por comprensión**

Se definen de la siguiente manera:

`[expresión | selectores, condiciones]`

Por ejemplo: `[(x,y) | x <- [0..5], y <- [0..3], x+y==4]`
es la lista que tiene los pares (1,3), (2,2), (3,1) y (4,0).

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
0, 3, 6, 9, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
0, 3, 6, 9, ...
- `repeat "hola"`
"hola", "hola", "hola", "hola", ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
0, 3, 6, 9, ...
- `repeat "hola"`
"hola", "hola", "hola", "hola", ...
- `primos = [n | n <- [2..], esPrimo n]`
(asumiendo `esPrimo` definida) 2, 3, 5, 7, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
0, 3, 6, 9, ...
- `repeat 'hola'`
"hola", "hola", "hola", "hola", ...
- `primos = [n | n <- [2..], esPrimo n]`
(asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
1, 1, 1, 1, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
0, 3, 6, 9, ...
- `repeat 'hola'`
"hola", "hola", "hola", "hola", ...
- `primos = [n | n <- [2..], esPrimo n]`
(asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
1, 1, 1, 1, ...

¿Cómo es posible trabajar con listas infinitas sin que se cuelgue?

Evaluación lazy

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

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

Evaluación lazy

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

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Si ejecutamos `nUnos 2`...

Evaluación lazy

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

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Si ejecutamos nUnos 2...

```
nUnos 2 → take 2 infinitosUnos → take 2 (1:infinitosUnos)→
1 : take (2-1) infinitosUnos → 1 : take 1 infinitosUnos →
1 : take 1 (1:infinitosUnos)→ 1 : 1: take (1-1)
infinitosUnos → 1 : take 0 infinitosUnos → 1 : 1 : []
```

Evaluación lazy

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

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Si ejecutamos `nUnos 2`...

```
nUnos 2 → take 2 infinitosUnos → take 2 (1:infinitosUnos) →
1 : take (2-1) infinitosUnos → 1 : take 1 infinitosUnos →
1 : take 1 (1:infinitosUnos) → 1 : 1: take (1-1)
infinitosUnos → 1 : take 0 infinitosUnos → 1 : 1 : []
```

- ¿Qué sucedería si usáramos otra estrategia de reducción?

Evaluación lazy

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

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Si ejecutamos `nUnos 2`...

```
nUnos 2 → take 2 infinitosUnos → take 2 (1:infinitosUnos) →
1 : take (2-1) infinitosUnos → 1 : take 1 infinitosUnos →
1 : take 1 (1:infinitosUnos) → 1 : 1: take (1-1)
infinitosUnos → 1 : take 0 infinitosUnos → 1 : 1 : []
```

- ¿Qué sucedería si usáramos otra estrategia de reducción?
- Si para algún término existe una reducción finita, entonces la estrategia de reducción lazy termina.

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun ::`

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

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

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`
- Reescribir `maximo` y `listaMasCorta` en base a `mejorSegun`

Filtrar elementos de una lista

Filtrar elementos de una lista

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

Filtrar elementos de una lista

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


Filtrar elementos de una lista

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

Filtrar elementos de una lista

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

Ejercicios

- Definir usando filter:

Filtrar elementos de una lista

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

Ejercicios

- Definir usando filter:
 - deLongitudN :: Int -> [[a]] -> [[a]]

Filtrar elementos de una lista

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

Ejercicios

- Definir usando filter:
 - `deLongitudN :: Int -> [[a]] -> [[a]]`
 - `soloPuntosFijosEnN :: Int -> [Int->Int] -> [Int->Int]`
Dados un número n y una lista de funciones, deja las funciones que al aplicarlas a n dan n

Transformar elementos de una lista

Transformar elementos de una lista

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

Transformar elementos de una lista

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

Transformar elementos de una lista

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


Transformar elementos de una lista

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

Ejercicio

- Definir usando map:

Transformar elementos de una lista

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

Ejercicio

- Definir usando map:
 - `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo:
`reverseAnidado [‘‘quedate’’, ‘‘en’’, ‘‘casa’’]`
devuelve `[‘‘asac’’, ‘‘ne’’, ‘‘etadeuq’’]`.
Ayuda: ya existe la función `reverse` que invierte una lista.

Transformar elementos de una lista

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

Ejercicio

- Definir usando map:
 - `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo:
`reverseAnidado [‘‘quedate’’, ‘‘en’’, ‘‘casa’’]`
devuelve [‘‘asac’’, ‘‘ne’’, ‘‘etadeuq’’].
Ayuda: ya existe la función `reverse` que invierte una lista.
 - `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares y los impares sin modificar.

¿Hay similitud entre estas definiciones?

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

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

¿Hay similitud entre estas definiciones?

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

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

→ En el caso base devolvemos un valor determinado.

¿Hay similitud entre estas definiciones?

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

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

- En el caso base devolvemos un valor determinado.
En el caso recursivo devolvemos algo en función de:
 - La cabeza de la lista.

¿Hay similitud entre estas definiciones?

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

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

- En el caso base devolvemos un valor determinado.
En el caso recursivo devolvemos algo en función de:
 - La cabeza de la lista.
 - El llamado recursivo sobre la cola de la lista.

¿Hay similitud entre estas definiciones?

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

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

- En el caso base devolvemos un valor determinado.
En el caso recursivo devolvemos algo en función de:
 - La cabeza de la lista.
 - El llamado recursivo sobre la cola de la lista.

¿Se puede generalizar?


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

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

→ z es el valor que devolvemos para una lista vacía.

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

- z es el valor que devolvemos para una lista vacía.
- f es una función que computa el resultado sobre la lista entera a partir de:
 - La cabeza de la lista.

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

→ z es el valor que devolvemos para una lista vacía.

- f es una función que computa el resultado sobre la lista entera a partir de:

→ La cabeza de la lista.

→ El llamado recursivo sobre la cola de la lista.

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

→ z es el valor que devolvemos para una lista vacía.

- f es una función que computa el resultado sobre la lista entera a partir de:

→ La cabeza de la lista.

→ El llamado recursivo sobre la cola de la lista.

Ejemplo

Si $xs = [x1, x2, x3]$ entonces:

$foldr\ f\ z\ xs = f\ x1\ (f\ x2\ (f\ x3\ z))$

Equivalentemente con notación infija:

$foldr\ \star\ z\ xs = x1\ \star\ (x2\ \star\ (x3\ \star\ z))$

Reescribiendo filter y map con foldr

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

Reescribiendo filter y map con foldr

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

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

Reescribiendo filter y map con foldr

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

```
filter p xs =  
    foldr (\x r -> if p x then x:r else r) [] xs
```

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

```
map f xs = foldr (\x r -> f x : r) [] xs
```

¿Es necesario el argumento xs?

Reescribiendo filter y map con foldr

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

```
map :: (a -> b) -> [a] -> [b]
map f  = foldr (\x r -> f x :  r) []
```

Desplegando la macro de las listas por comprensión

Definir una expresión equivalente a las siguiente utilizando `map` y `filter`:

Ejercicio

```
listaComp :: (a -> Bool) -> (a -> b) -> [a] -> [b]
listaComp p f xs = [f x | x <- xs, p x]
```

i i i i i i i i i ? ? ? ? ? ? ? ?