Programación Funcional en Haskell

Primera parte

Paradigmas de Lenguajes de Programación

Clase virtual por Carolina Lucía González

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

16 de abril de 2020

Introducción

En esta clase veremos una introducción a la programación funcional utilizando Haskell.

Introducción

En esta clase veremos una introducción a la programación funcional utilizando Haskell.

Si bien en Taller de Álgebra I ya trabajaron un poco con este tema, haremos un breve repaso y también veremos temas nuevos.

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
  $\longrightarrow$ basado en funciones matemáticas, sin asignación de variables, etc.

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
  $\longrightarrow$ los desarrollaremos más adelante.

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

$\longrightarrow$ lo desarrollaremos más adelante.

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

Utilizaremos **GHC** y su entorno interactivo GHCi.

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

Utilizaremos **GHC** y su entorno interactivo GHCi.

Curiosidad: el logo de Haskell está formado por los símbolos λ y $\gg=$ (este último se usa al trabajar con **mónadas**, un concepto que verán más adelante).

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Lambda

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

Utilizaremos **GHC** y su entorno interactivo GHCi.

Curiosidad: el logo de Haskell está formado por los símbolos λ y $\gg=$ (este último se usa al trabajar con **mónadas**, un concepto que verán más adelante).

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Bind

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

Utilizaremos **GHC** y su entorno interactivo GHCi.

Curiosidad: el logo de Haskell está formado por los símbolos λ y $\gg=$ (este último se usa al trabajar con **mónadas**, un concepto que verán más adelante).

Sobre Haskell

- Nombre en honor a

Haskell Brooks Curry
Matemático
12/09/1900 – 01/09/1982

Haskell

- Lenguaje de programación funcional (puro)
- Tipado estático, polimorfismo y alto orden
- Evaluación lazy

Utilizaremos **GHC** y su entorno interactivo GHCi.

Curiosidad: el logo de Haskell está formado por los símbolos λ y $\gg=$ (este último se usa al trabajar con **mónadas**, un concepto que verán más adelante).

Repaso: usando GHCi

Cómo empezar:

Repaso: usando GHCi

Cómo empezar:

```
$
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
```

Repaso: usando GHCi

Cómo empezar:

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

Repaso: usando GHCi

Cómo empezar:

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

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
```

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`

Repaso: tipado en Haskell

- Tipado estático:

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: "si tipa entonces funciona bien"

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: "si tipa entonces funciona bien"

BUSTED

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.

- Mito: "si tipa entonces funciona bien"

BUSTED

Si bien el hecho de que tipe es una buena señal, el programa puede andar mal por otros motivos. Por ejemplo:

```
suma1 :: Int -> Int
suma1 x = x+2
```

tipa pero no hace lo que uno espera.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.

- Mito: "si tipa entonces funciona bien"

BUSTED

- Inferencia de tipos:

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: "si tipa entonces funciona bien"

BUSTED

- Inferencia de tipos:
  - No es necesario indicar <u>todos</u> los tipos, Haskell tiene mecanismos para inferirlos (veremos más sobre este tema en λ–Cálculo).

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: "si tipa entonces funciona bien"

BUSTED

- Inferencia de tipos:
  - No es necesario indicar <u>todos</u> los tipos, Haskell tiene mecanismos para inferirlos (veremos más sobre este tema en λ–Cálculo).
  - Sin embargo, indicar todos los tipos es una **buena práctica**.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: “si tipa entonces funciona bien”

  BUSTED

- Inferencia de tipos:
  - No es necesario indicar <u>todos</u> los tipos, Haskell tiene mecanismos para inferirlos (veremos más sobre este tema en λ–Cálculo).
  - Sin embargo, indicar todos los tipos es una **buena práctica**.
- Para conocer el tipo de una expresión podemos usar `:t` en GHCi.

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.
- Mito: "si tipa entonces funciona bien"

BUSTED

- Inferencia de tipos:
  - No es necesario indicar <u>todos</u> los tipos, Haskell tiene mecanismos para inferirlos (veremos más sobre este tema en λ–Cálculo).
  - Sin embargo, indicar todos los tipos es una **buena práctica**.
- Para conocer el tipo de una expresión podemos usar `:t` en GHCi.

```
> :t True
```

Repaso: tipado en Haskell

- Tipado estático:
  - El tipo de cada expresión se conoce en tiempo de compilación.
  - Código más seguro: si no tipa no compila.
  - Todo tiene un tipo.

- Mito: “si tipa entonces funciona bien”

BUSTED

- Inferencia de tipos:
  - No es necesario indicar <u>todos</u> los tipos, Haskell tiene mecanismos para inferirlos (veremos más sobre este tema en λ–Cálculo).
  - Sin embargo, indicar todos los tipos es una **buena práctica**.

- Para conocer el tipo de una expresión podemos usar `:t` en GHCi.

```
> :t True
True :: Bool
```

Repaso: tipos básicos

- `Bool`: `True`, `False`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `'a'`, `'A'`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: 3, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `'a'`, `'A'`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `'a'`, `'A'`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `'a'`, `'A'`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: 3, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `'a'`, `'A'`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `('A', 'T', 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: 3, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`
  - `‘‘Lovelace’’ :: [Char]`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: 3, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`
  - `‘‘Lovelace’’ :: [Char]`
  - `[(1,‘a’), (2,‘b’)] :: [(Int, Char)]`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`
  - `‘‘Lovelace’’ :: [Char]`
  - `[(1,‘a’), (2,‘b’)] :: [(Int, Char)]`
  - `[[0],[0,1],[0,1,1],[1,1,2],[1,2,3],[2,3,5]] :: [[Int]]`

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`
  - `‘‘Lovelace’’ :: [Char]`
  - `[(1,‘a’), (2,‘b’)] :: [(Int, Char)]`
  - `[[0],[0,1],[0,1,1],[1,1,2],[1,2,3],[2,3,5]] :: [[Int]]`
- *Funciones*:

Repaso: tipos básicos

- `Bool`: `True`, `False`
- `Char`: `‘a’`, `‘A’`
- `Int`: `3`, `-4`
  (También está `Integer`, que es no-acotado y permite representar números muy grandes, pero `Int` es más eficiente.)
- `Float`: `3.14`, `-1.618`
- *Tuplas*:
  - `(7, True) :: (Int, Bool)`
  - `(‘A’, ‘T’, 23.06, 1912) :: (Char, Char, Float, Int)`
- *Listas*:
  - `[11,8,2,7,10,13] :: [Int]`
  - `[‘A’, ‘d’, ‘a’] :: [Char]`
  - `‘‘Lovelace’’ :: [Char]`
  - `[(1,‘a’), (2,‘b’)] :: [(Int, Char)]`
  - `[[0],[0,1],[0,1,1],[1,1,2],[1,2,3],[2,3,5]] :: [[Int]]`
- *Funciones*:
  - `not :: Bool -> Bool`

Repaso: alto orden

En Haskell el tipo de las funciones <u>no es</u> "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

En Haskell el tipo de las funciones no es "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

Por ejemplo, definamos una función que toma una función de enteros a enteros y devuelve el doble de su aplicación en 0:

Repaso: alto orden

En Haskell el tipo de las funciones no es "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

Por ejemplo, definamos una función que toma una función de enteros a enteros y devuelve el doble de su aplicación en 0:

```
aplicaEn0x2 :: (Int -> Int) -> Int
aplicaEn0x2 f = 2*(f 0)
```

Repaso: alto orden

En Haskell el tipo de las funciones no es "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

Por ejemplo, definamos una función que toma una función de enteros a enteros y devuelve el doble de su aplicación en 0:

```haskell
aplicaEn0x2 :: (Int -> Int) -> Int
aplicaEn0x2 f = 2*(f 0)
```

De esta manera, si tenemos `suma1 x = x+1`:

Repaso: alto orden

En Haskell el tipo de las funciones no es "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

Por ejemplo, definamos una función que toma una función de enteros a enteros y devuelve el doble de su aplicación en 0:

```
aplicaEn0x2 :: (Int -> Int) -> Int
aplicaEn0x2 f = 2*(f 0)
```

De esta manera, si tenemos `suma1 x = x+1`:

```
> aplicaEn0x2 suma1
```

Repaso: alto orden

En Haskell el tipo de las funciones no es "más especial" que los demás.

Esto permite trabajar con facilidad con listas de funciones, funciones que toman o devuelven otras funciones, etc.

Por ejemplo, definamos una función que toma una función de enteros a enteros y devuelve el doble de su aplicación en 0:

```
aplicaEn0x2 :: (Int -> Int) -> Int
aplicaEn0x2 f = 2*(f 0)
```

De esta manera, si tenemos `suma1 x = x+1`:

```
> aplicaEn0x2 suma1
2
```

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

```
\x -> x+1
```

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

```
\x -> x+1
```

`\x` indica que `x` es el argumento de la función y todo lo que está a la derecha de `->` es el cuerpo de la función (en este caso, `x+1`).

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

```
\x -> x+1
```

`\x` indica que `x` es el argumento de la función y todo lo que está a la derecha de `->` es el cuerpo de la función (en este caso, `x+1`).

Esto es particularmente útil si queremos definir una función que vamos a usar en un solo lugar del código, en general para pasar como argumento a otra función.

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

```
\x -> x+1
```

`\x` indica que `x` es el argumento de la función y todo lo que está a la derecha de `->` es el cuerpo de la función (en este caso, `x+1`).

Esto es particularmente útil si queremos definir una función que vamos a usar en un solo lugar del código, en general para pasar como argumento a otra función.

Por ejemplo:

```
> aplicaEn0x2 (\x -> 4*x+3)
```

Repaso: funciones anónimas (abstracciones λ)

Podemos usar funciones sin darles un nombre. Por ejemplo:

```
\x -> x+1
```

`\x` indica que `x` es el argumento de la función y todo lo que está a la derecha de `->` es el cuerpo de la función (en este caso, `x+1`).

Esto es particularmente útil si queremos definir una función que vamos a usar en un solo lugar del código, en general para pasar como argumento a otra función.

Por ejemplo:

```
> aplicaEn0x2 (\x -> 4*x+3)
6
```

Repaso: analizando por casos

Los siguientes son ejemplos para resaltar las diferencias entre las distintas expresiones.
Hay mejores formas de definir esta función. No hagan esto en un parcial!!

Repaso: analizando por casos

Los siguientes son <u>ejemplos</u> para resaltar las diferencias entre las distintas expresiones. Hay mejores formas de definir esta función. No hagan esto en un parcial!!

If

```
esCero n = if n == 0
           then True
           else False
```

Repaso: analizando por casos

Los siguientes son ejemplos para resaltar las diferencias entre las distintas expresiones.
Hay mejores formas de definir esta función. No hagan esto en un parcial!!

If

```
esCero n = if n == 0
           then True
           else False
```

Guardas

```
esCero n | n == 0 = True
         | otherwise = False
```

Repaso: analizando por casos

Los siguientes son ejemplos para resaltar las diferencias entre las distintas expresiones. Hay mejores formas de definir esta función. No hagan esto en un parcial!!

If

```
esCero n = if n == 0
           then True
           else False
```

Guardas

```
esCero n | n == 0 = True
         | otherwise = False
```

Pattern matching

```
esCero 0 = True
esCero n = False
```

Repaso: analizando por casos

Los siguientes son ejemplos para resaltar las diferencias entre las distintas expresiones.

Hay mejores formas de definir esta función. No hagan esto en un parcial!!

If

```
esCero n = if n == 0
           then True
           else False
```

Guardas

```
esCero n | n == 0 = True
         | otherwise = False
```

Pattern matching

```
esCero 0 = True
esCero n = False
```

Case

```
esCero n = case n of
             0 -> True
             _ -> False
```

Repaso: recursión

Funciones que se llaman a sí mismas. Por ejemplo:

```
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

(Así se hace el pattern matching en listas.)

Repaso: recursión

Funciones que se llaman a sí mismas. Por ejemplo:

```
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

(Así se hace el pattern matching en listas.)

También podemos tener funciones mutuamente recursivas, sin necesidad de definirlas de una forma especial. Por ejemplo:

```
par 0 = True
par n = impar (n-1)

impar 0 = False
impar n = par (n-1)
```

(De nuevo, estos sólo son ejemplos para entender el concepto, hay mejores formas de definir las funciones anteriores.)

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre.
Veamos un ejemplo.

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ?
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ?
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Primero notemos que `longitud` es una **función**, así que "toma algo" y "devuelve algo".

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ? -> ?
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Primero notemos que `longitud` es una **función**, así que "toma algo" y "devuelve algo".

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ? -> ?
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Sigamos por lo fácil... ¿qué devuelve?

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ? -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Sigamos por lo fácil... ¿qué devuelve?

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ? -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: ? -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?
Bueno, es una lista...

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [?] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?
Bueno, es una lista...

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [?] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?
Bueno, es una lista...
¿De qué tipo?

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre.
Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [?] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?
Bueno, es una lista...
¿De qué tipo?
De cualquier tipo!

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

¿Qué tipo tiene el argumento?
Bueno, es una lista...
¿De qué tipo?
De cualquier tipo!

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

El sistema de tipos de Haskell permite definir funciones para ser usadas con más de un tipo. Su tipo se expresa con **variables de tipo**.

Repaso: polimorfismo paramétrico

Este es un concepto que ya vieron, pero probablemente no con este nombre. Veamos un ejemplo.

¿Cuál es el tipo de la función `longitud`?

```
longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

El sistema de tipos de Haskell permite definir funciones para ser usadas con más de un tipo. Su tipo se expresa con **variables de tipo**.
Para esto usamos letras minúsculas.

(Van a notar que se usa mucho a, pero cualquier letra es válida.)

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales ? = ?
```

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales ? = ?
```

Empecemos por la definición.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales ? = ?
```

Empecemos por la definición.

SPOILER ALERT! Pensar la definición antes de seguir.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales ? = ?
```

Empecemos por la definición.

Como es usual al trabajar con listas, separemos en casos y ¡hagamos pattern matching!

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = ?
```

Empecemos por la definición.

Como es usual al trabajar con listas, separemos en casos y ¡hagamos pattern matching!
Hay dos casos base. ¿Cómo queda el caso recursivo?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Empecemos por la definición.

Como es usual al trabajar con listas, separemos en casos y ¡hagamos pattern matching!
Hay dos casos base. ¿Cómo queda el caso recursivo?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ? -> ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ? -> ?
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ? -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: ? -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?
¿Y qué recibe?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: [?] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?
¿Y qué recibe?
Un lista, ok. ¿De qué tipo?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: [?] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?
¿Y qué recibe?
Un lista, ok. ¿De qué tipo?
¿De cualquier tipo? ¿Puede ser una lista de funciones?

Más polimorfismo

Definir y dar el tipo de la función `todosIguales` que dada una lista determina si todos sus elementos son iguales.

```
todosIguales :: [¿a?] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Pasemos al tipo.
Hagamos un análisis parecido al que hicimos con `longitud`.
Sabemos que `todosIguales` es una función.
¿Qué devuelve?
¿Y qué recibe?
Un lista, ok. ¿De qué tipo?
¿De cualquier tipo? ¿Puede ser una lista de funciones?
¿Para qué tipos está definida la igualdad?

Polimorfismo ad hoc (typeclasses)

Las **clases de tipos (typeclasses)** de Haskell permiten agrupar tipos de acuerdo a algunas operaciones que definen.

Polimorfismo ad hoc (typeclasses)

Las **clases de tipos (typeclasses)** de Haskell permiten agrupar tipos de acuerdo a algunas operaciones que definen.

Por ejemplo:

- `Eq` (==, /=)
- `Ord` (`(<)`, `(<=)`, `(>=)`, `(>)`, ...)
- `Num` (`(+)`, `(-)`, `(*)`, ...)
- `Show` (`show`, ...)
- `Monad` (`(>>=)`, `return`, ...)

Polimorfismo ad hoc (typeclasses)

Las **clases de tipos (typeclasses)** de Haskell permiten agrupar tipos de acuerdo a algunas operaciones que definen.

Por ejemplo:

- `Eq` (==, /=)
- `Ord` (`(<)`, `(<=)`, `(>=)`, `(>)`, ...)
- `Num` (`(+)`, `(-)`, `(*)`, ...)
- `Show` (`show`, ...)
- `Monad` (`(>>=)`, `return`, ...)

Definición de una instancia en Haskell:

```
instance Show Bool where
    show True = ``True''
    show False = ``False''
```

Usamos `instance` para indicar qué a clase queremos que pertenezca un tipo, y luego definimos las funciones de esa clase para ese tipo.

Polimorfismo ad hoc (typeclasses)

Las **clases de tipos (typeclasses)** de Haskell permiten agrupar tipos de acuerdo a algunas operaciones que definen.

Por ejemplo:

- `Eq` (==, /=)
- `Ord` (`(<)`, `(<=)`, `(>=)`, `(>)`, ...)
- `Num` (`(+)`, `(-)`, `(*)`, ...)
- `Show` (`show`, ...)
- `Monad` (`(>>=)`, `return`, ...)

Definición de una instancia en Haskell:

```
instance Show Bool where
    show True = ''True''
    show False = ''False''
```

Usamos `instance` para indicar qué a clase queremos que pertenezca un tipo, y luego definimos las funciones de esa clase para ese tipo.

Más info sobre typeclasses **acá.**

Polimorfismo ad hoc (typeclasses)

Volviendo al ejercicio anterior, nos quedaría

```
todosIguales :: Eq a => [a] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Con "`Eq a =>`" estamos expresando que el tipo a tiene que pertenecer a la clase `Eq`.

Polimorfismo ad hoc (typeclasses)

Volviendo al ejercicio anterior, nos quedaría

```
todosIguales :: Eq a => [a] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (y:x:xs) = y==x && todosIguales (x:xs)
```

Con "`Eq a =>`" estamos expresando que el tipo a tiene que pertenecer a la clase `Eq`.

De esta manera, podemos usar esta función en listas de cualquier tipo que tenga definida la igualdad (por ejemplo: enteros, booleanos, tuplas de tipos con igualdad). Pero no podemos usarla, por ejemplo, en listas de funciones (porque, como ya sabrán, no podemos definir una función que en tiempo finito determine si dos funciones son iguales).

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

```haskell
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. `curry :: ((a,b) -> c) -> (a -> b -> c)`
   que devuelve la versión currificada de una función no currificada.
2. `uncurry :: (a -> b -> c) -> ((a,b) -> 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.

curry – uncurry

Ejercicio

Definir las siguientes funciones:

1. `curry :: ((a,b) -> c) -> (a -> b -> c)`
   que devuelve la versión currificada de una función no currificada.
2. `uncurry :: (a -> b -> c) -> ((a,b) -> 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.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Ejercicio

Definir las siguientes funciones:

1. `curry :: ((a,b) -> c) -> (a -> b -> c)`
   que devuelve la versión currificada de una función no currificada.
2. `uncurry :: (a -> b -> c) -> ((a,b) -> 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.

Soluciones

```haskell
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = \a b -> f (a,b)
```

curry – uncurry

Ejercicio

Definir las siguientes funciones:

1. `curry :: ((a,b) -> c) -> (a -> b -> c)`
   que devuelve la versión currificada de una función no currificada.
2. `uncurry :: (a -> b -> c) -> ((a,b) -> 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.

Soluciones

```haskell
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = \a b -> f (a,b)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f = \(a,b) -> f a b
```

Aplicación parcial

Ejercicios

Si definimos `doble x = prod’ 2 x`

1. ¿Cuál es el tipo de `doble`?

Aplicación parcial

Ejercicios

Si definimos `doble x = prod' 2 x`

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

Aplicación parcial

Ejercicios

Si 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`?

Aplicación parcial

Ejercicios

Si 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`:

Aplicación parcial

Ejercicios

Si 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`

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Soluciones

1. `Int -> Int`

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Soluciones

1. `Int -> Int`
2. Nada. La función se comporta igual. No hace falta pasarle argumentos a las funciones para poder definirlas.

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Soluciones

1. `Int -> Int`
2. Nada. La función se comporta igual. No hace falta pasarle argumentos a las funciones para poder definirlas.
3. Es la función que toma un número n y devuelve $1 + n$. Ponerle paréntesis a un operador infijo hace que se comporte como prefijo.

Aplicación parcial

Ejercicios

Si 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`
   - `esMayorDeEdad :: Int -> Bool`

Soluciones

1. `Int -> Int`
2. Nada. La función se comporta igual. No hace falta pasarle argumentos a las funciones para poder definirlas.
3. Es la función que toma un número n y devuelve $1 + n$. Ponerle paréntesis a un operador infijo hace que se comporte como prefijo.
4. `triple = (*)3` Otras formas: `triple = (*3)` y `triple = (3*)`
   `esMayorDeEdad = (<=)18` Otra forma: `esMayorDeEdad = (18<=)`

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

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

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

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

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

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

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
    1. `(.)` que compone dos funciones. Por ejemplo:
       `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
    2. `flip` que invierte los argumentos de una función. Por ejemplo:
       `flip (\x y -> x - y) 1 5` devuelve 4.
    3. `($)` que aplica una función a un argumento. Por ejemplo:
       `prod $ (2,3)` devuelve 6.

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Soluciones

1. 1.
      ```
      (.) :: (b -> c) -> (a -> b) -> a -> c
      f . g = \x -> f (g x)
      ```

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Soluciones

1.
   1. ```
      (.) :: (b -> c) -> (a -> b) -> a -> c
      f . g = \x -> f (g x)
      ```
   2. ```
      flip :: (a -> b -> c) -> b -> a -> c
      flip f a b = f b a
      ```

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Soluciones

1.
   1. ```
      (.) :: (b -> c) -> (a -> b) -> a -> c
      f . g = \x -> f (g x)
      ```
   2. ```
      flip :: (a -> b -> c) -> b -> a -> c
      flip f a b = f b a
      ```
   3. ```
      ($) :: (a -> b) -> a -> b
      f $ x = f x
      ```
      ```

Funciones muy útiles (aunque no lo parezcan)

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   1. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   2. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   3. `($)` que aplica una función a un argumento. Por ejemplo:
      `prod $ (2,3)` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Soluciones

1.
   1.
      ```
      (.) :: (b -> c) -> (a -> b) -> a -> c
      f . g = \x -> f (g x)
      ```
   2.
      ```
      flip :: (a -> b -> c) -> b -> a -> c
      flip f a b = f b a
      ```
   3.
      ```
      ($) :: (a -> b) -> a -> b
      f $ x = f x
      ```
2. `flip ($) 0` es una función que toma una función f y devuelve $f(0)$.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\, mod\, 26)$.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\, mod\, 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\ mod\ 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\, mod\, 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ?
```

Pista: pensar paso por paso qué operaciones se quieren hacer.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\, mod\, 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . ord
```

Primero queremos aplicar ord.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\ mod\ 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . ord
```

A ese número le queremos sumar 13...

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\; mod\; 26)$.

Implementar `rot13 :: Char -> Char` <u>sin escribir variables</u>.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . (+ (13 - ord 'a')) . ord
```

...y restarle la posición de 'a', para "contar desde 0".

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\ mod\ 26)$.

Implementar `rot13 :: Char -> Char` <u>sin escribir variables</u>.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . (+ (13 - ord 'a')) . ord
```

A todo esto hay que calcularle el resto módulo 26.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\, mod\, 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . ¿mod 26? . (+ (13 - ord 'a')) . ord
```

Como los argumentos de `mod` no están en el orden que necesitamos...

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13) \, mod \, 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . (flip mod 26) . (+ (13 - ord 'a')) . ord
```

...los damos vuelta con `flip`.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\ mod\ 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = ? . (flip mod 26) . (+ (13 - ord 'a')) . ord
```

Y finalmente vemos qué letra es.

Ejercicios

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada letra con la que aparece 13 lugares después en el alfabeto. Básicamente, $ROT13(\ell) = char((\#\ell + 13)\ mod\ 26)$.

Implementar `rot13 :: Char -> Char` sin escribir variables.
Suponemos que esta función sólo la usamos con letras minúsculas.

Ayudas:

- `mod`: dados dos números x y m devuelve el resto de x en la división por m.
- `ord::Char->Int`: dado un carácter ASCII devuelve su número asociado.
- `chr::Int->Char`: dado un número devuelve su carácter ASCII asociado.

  (Para usar estas dos últimas funciones en Haskell hay que poner `import Data.Char`.)
- Los números ASCII de las letras minúsculas empiezan en la 'a' y siguen consecutivamente hasta la 'z'. El número de la 'a' no es 0.

Solución

```
rot13 = chr . (+ ord 'a') . (flip mod 26) . (+ (13 - ord 'a')) . ord
```

Voilà!

Ejercicios

Definir las siguientes funciones. Precondición: la lista tiene al menos un elemento.

1. `maximo :: Ord a => [a] -> a`
2. `minimo :: Ord a => [a] -> a`
3. `listaMasCorta :: [[a]] -> [a]`

Ejercicios

Definir las siguientes funciones. Precondición: la lista tiene al menos un elemento.

1. `maximo :: Ord a => [a] -> a`
2. `minimo :: Ord a => [a] -> a`
3. `listaMasCorta :: [[a]] -> [a]`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Ejercicios

Definir las siguientes funciones. Precondición: la lista tiene al menos un elemento.

1. `maximo :: Ord a => [a] -> a`
2. `minimo :: Ord a => [a] -> a`
3. `listaMasCorta :: [[a]] -> [a]`

Soluciones

```haskell
maximo [x] = x
maximo (x:xs) =
     let m = maximo xs
     in if x > m then x else m

minimo [x] = x
minimo (x:xs) =
     let m = minimo xs
     in if x < m then x else m

listaMasCorta [l] = l
listaMasCorta (l:ls) =
     let lmc = listaMasCorta xs
     in if length l < length lmc then l else lmc
```

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Soluciones

```
mejorSegun _ [x] = x
mejorSegun esMejor (x:xs) =
   let m = mejorSegun esMejor xs
   in if esMejor x m then x else m
```

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Soluciones

```
mejorSegun _ [x] = x
mejorSegun esMejor (x:xs) =
   let m = mejorSegun esMejor xs
   in if esMejor x m then x else m

maximo = mejorSegun (>)
```

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Soluciones

```
mejorSegun _ [x] = x
mejorSegun esMejor (x:xs) =
   let m = mejorSegun esMejor xs
   in if esMejor x m then x else m

maximo = mejorSegun (>)

minimo = mejorSegun (<)
```

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento.
2. Reescribir `maximo`, `minimo` y `listaMasCorta` usando `mejorSegun`.

Soluciones

```
mejorSegun _ [x] = x
mejorSegun esMejor (x:xs) =
   let m = mejorSegun esMejor xs
   in if esMejor x m then x else m

maximo = mejorSegun (>)

minimo = mejorSegun (<)

listaMasCorta= mejorSegun (\l1 l2 -> length l1 < length l2)
```

Listas

Repasemos algunos conceptos de listas que ya vieron en Taller de Álgebra I y veamos otros nuevos.

Listas

Repasemos algunos conceptos de listas que ya vieron en Taller de Álgebra I y veamos otros nuevos.
Hay varias formas de definir una lista:

Listas

Repasemos algunos conceptos de listas que ya vieron en Taller de Álgebra I y veamos otros nuevos.

Hay varias formas de definir una lista:

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

Listas

Repasemos algunos conceptos de listas que ya vieron en Taller de Álgebra I y veamos otros nuevos.

Hay varias formas de definir una lista:

- **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.

Listas

Repasemos algunos conceptos de listas que ya vieron en Taller de Álgebra I y veamos otros nuevos.

Hay varias formas de definir una lista:

- **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:

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, ...

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]`

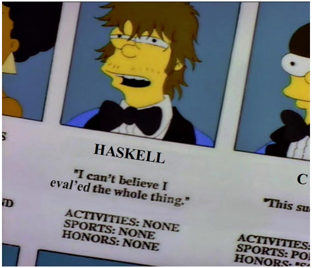
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...

- `infinitosUnos = 1 : infinitosUnos`

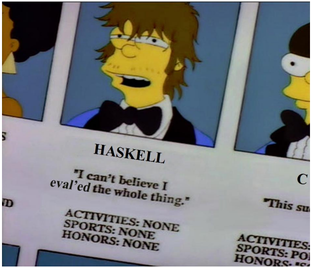
  1, 1, 1, 1, ...

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

Evaluación Lazy

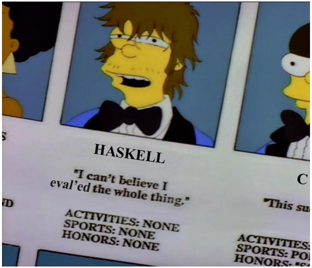


Evaluación Lazy



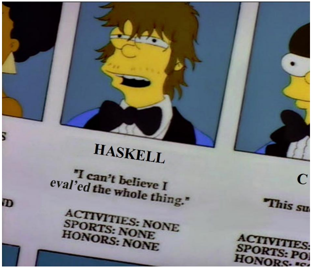
- Las expresiones son evaluadas **sólo cuando es necesario**. Además, se evita repetir la evaluación en caso de aparecer varias veces.

Evaluación Lazy



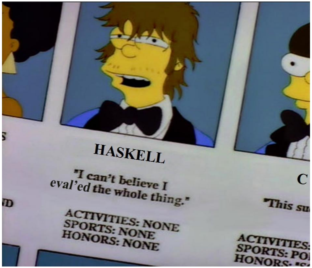
- Las expresiones son evaluadas **sólo cuando es necesario**. Además, se evita repetir la evaluación en caso de aparecer varias veces.
- En cada paso se reduce la **expresión reducible (redex: reducible expression)** más externa y más a la izquierda.

Evaluación Lazy



- Las expresiones son evaluadas **sólo cuando es necesario**. Además, se evita repetir la evaluación en caso de aparecer varias veces.
- En cada paso se reduce la **expresión reducible (redex: reducible expression)** más externa y más a la izquierda.
- Básicamente: primero las funciones más externas y luego los argumentos (sólo si se necesitan).

Evaluación Lazy



- Las expresiones son evaluadas **sólo cuando es necesario**. Además, se evita repetir la evaluación en caso de aparecer varias veces.
- En cada paso se reduce la **expresión reducible (redex: reducible expression)** más externa y más a la izquierda.
- Básicamente: primero las funciones más externas y luego los argumentos (sólo si se necesitan).
- Sus ventajas van más allá de la eficiencia temporal.

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

Ejercicios

```haskell
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

SPOILER ALERT!

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

```
nUnos 2
```

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

`nUnos 2`
$\rightarrow$ **`take 2 infinitosUnos`**

```
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

nUnos 2
→ **take 2 infinitosUnos**
→ **take 2 (1:infinitosUnos)**

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

```
nUnos 2
→ take 2 infinitosUnos
→ take 2 (1:infinitosUnos)
→ 1 : take (2-1) infinitosUnos
```

Ejercicios

```haskell
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
```

Mostrar los pasos necesarios para reducir `nUnos 2`.

```
nUnos 2
→ take 2 infinitosUnos
→ take 2 (1:infinitosUnos)
→ 1 : take (2-1) infinitosUnos
→ 1 : take 1 infinitosUnos
```

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `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)
```

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `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
```

Ejercicios

```
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
```

Mostrar los pasos necesarios para reducir `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 : 1 : take 0 infinitosUnos
```

Ejercicios

```haskell
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
```

Mostrar los pasos necesarios para reducir `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 : 1 : take 0 infinitosUnos
→ 1 : 1 : [] = [1,1]
```

Ejercicios

```haskell
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
```

¿Qué sucedería si usáramos otra estrategia de reducción?

Ejercicios

```
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
```

¿Qué sucedería si usáramos otra estrategia de reducción?

SPOILER ALERT!

Ejercicios

```haskell
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
```

¿Qué sucedería si usáramos otra estrategia de reducción?

Si usamos una estrategia que primero reduce `infinitosUnos`, no terminaría nunca.

Ejercicios

```
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
```

¿Existe algún término que admita una reducción finita pero para el cual la estrategia lazy no termine?

Ejercicios

```
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
```

¿Existe algún término que admita una reducción finita pero para el cual la estrategia lazy no termine?

SPOILER ALERT!

Ejercicios

```
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
```

¿Existe algún término que admita una reducción finita pero para el cual la estrategia lazy no termine?

No. Siempre que el término admita una reducción finita, la reducción lazy termina en una cantidad finita de pasos.

Ejercicios

```haskell
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 un término admite otra reducción finita además de la dada por reducción lazy, ¿el resultado de ambas coincide?

Ejercicios

```haskell
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 un término admite otra reducción finita además de la dada por reducción lazy, ¿el resultado de ambas coincide?

SPOILER ALERT!

Ejercicios

```
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 un término admite otra reducción finita además de la dada por reducción lazy, ¿el resultado de ambas coincide?

Sí.

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

- toma una función que sabe cómo convertir un elemento de tipo `a` en otro de tipo `b`, y

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

- toma una función que sabe cómo convertir un elemento de tipo `a` en otro de tipo `b`, y
- devuelve una función que sabe cómo convertir una lista de `a` en una lista de `b`.

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

- toma una función que sabe cómo convertir un elemento de tipo `a` en otro de tipo `b`, y
- devuelve una función que sabe cómo convertir una lista de `a` en una lista de `b`.

Ejercicio

Definir `map`.

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

- toma una función que sabe cómo convertir un elemento de tipo `a` en otro de tipo `b`, y
- devuelve una función que sabe cómo convertir una lista de `a` en una lista de `b`.

Ejercicio

Definir `map`.

Solución

SPOILER ALERT!

Esquemas de recursión sobre listas: map

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

Permite aplicar una transformación a cada elemento de una lista.
O, dicho de otra forma, la función `map`

- toma una función que sabe cómo convertir un elemento de tipo `a` en otro de tipo `b`, y
- devuelve una función que sabe cómo convertir una lista de `a` en una lista de `b`.

Ejercicio

Definir `map`.

Solución

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

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`
2. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
   Ayuda: `l !! n` devuelve el elemento de `l` en la posición `n`.

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`
2. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
   Ayuda: `l !! n` devuelve el elemento de `l` en la posición `n`.

Soluciones

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`
2. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
   Ayuda: `l !! n` devuelve el elemento de `l` en la posición `n`.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`
2. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
   Ayuda: `l !! n` devuelve el elemento de `l` en la posición `n`.

Soluciones

```
longitudes = map length
```

Esquemas de recursión sobre listas: map

Ejercicios

Definir usando `map`:

1. `longitudes :: [[a]] -> [Int]`
2. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
   Ayuda: `l !! n` devuelve el elemento de `l` en la posición `n`.

Soluciones

```
longitudes = map length

shuffle indices l = map (l !!) indices
```

Esquemas de recursión sobre listas: filter

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

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Esquemas de recursión sobre listas: filter

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

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Ejercicio

Definir `filter`.

Esquemas de recursión sobre listas: filter

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

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Ejercicio

Definir `filter`.

Solución

SPOILER ALERT!

Esquemas de recursión sobre listas: filter

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

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Ejercicio

Definir `filter`.

Solución

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

Esquemas de recursión sobre listas: filter

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

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Ejercicio

Definir `filter`.

Solución

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

Otra forma:

```
filter p l = [x | x <- l, p x]
```

Ejercicios

Definir usando `filter`

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.
3. `quickSort :: Ord a => [a] -> [a]`

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.
3. `quickSort :: Ord a => [a] -> [a]`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.
3. `quickSort :: Ord a => [a] -> [a]`

Soluciones

```
deLongitudN n = filter ((== n).length)
```

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.
3. `quickSort :: Ord a => [a] -> [a]`

Soluciones

```
deLongitudN n = filter ((== n).length)

soloPuntosFijosEnN n = filter ((== n).(flip ($) n))
```

Esquemas de recursión sobre listas: filter

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `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.
3. `quickSort :: Ord a => [a] -> [a]`

Soluciones

```
deLongitudN n = filter ((== n).length)

soloPuntosFijosEnN n = filter ((== n).(flip ($) n))

quickSort [] = []
quickSort (x:xs) = let chicos = quickSort (filter (x >=) xs)
                       grandes = quickSort (filter (x <) xs)
                   in chicos ++ [x] ++ grandes
```

Ejercicios con map y filter

Ejercicios

Definir sin utilizar recursión explícita:

1. `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.

Ejercicios con map y filter

Ejercicios

Definir sin utilizar recursión explícita:

1. `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.
2. `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

Ejercicios con map y filter

Ejercicios

Definir sin utilizar recursión explícita:

1. `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.
2. `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Ejercicios con map y filter

Ejercicios

Definir sin utilizar recursión explícita:

1. `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.
2. `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

Soluciones

```
reverseAnidado = reverse . (map reverse)
```

Ejercicios con map y filter

Ejercicios

Definir sin utilizar recursión explícita:

1. `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.
2. `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

Soluciones

```
reverseAnidado = reverse . (map reverse)

paresCuadrados = (map cuadrado) . (filter esPar)
                 where esPar = (== 0) . (flip mod 2)
                       cuadrado = \x -> x*x
```

Tarea

Tarea

(pero es fácil y no se entrega)

Tarea

Tarea

1) Crear un archivo test.hs con algunas de las funciones dadas en clase. Abrirlo con GHCi y jugar un poco, en especial con la aplicación parcial.

Tarea

1) Crear un archivo test.hs con algunas de las funciones dadas en clase. Abrirlo con GHCi y jugar un poco, en especial con la aplicación parcial.

2) Entre las primeras diapositivas vimos definiciones "feas" de `esCero`, `impar` y `par`. Definirlas de forma elegante, aprovechando todos los conceptos nuevos vistos en clase.

¿Preguntas?

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?

Estaremos respondiendo consultas por mail y Discord durante el horario de la materia :)