Programación Funcional en Haskell

Segunda parte

Paradigmas de Lenguajes de Programación

Clase virtual por Carolina Lucía González

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

23 de abril de 2020

Generación infinita

En la clase anterior vimos que gracias a la **evaluación lazy** podemos trabajar con **estructuras de datos infinitas**.

Generación infinita

En la clase anterior vimos que gracias a la **evaluación lazy** podemos trabajar con **estructuras de datos infinitas**.

Por ejemplo...

`[0..]` denota la lista de todos los números enteros no negativos y la podemos usar (con cuidado) como una lista cualquiera, para hacer cosas como `take 10 [0..]` o `map (*2) [0..]`.

Generación infinita

En la clase anterior vimos que gracias a la **evaluación lazy** podemos trabajar con **estructuras de datos infinitas**.

Por ejemplo...

`[0..]` denota la lista de todos los números enteros no negativos y la podemos usar (con cuidado) como una lista cualquiera, para hacer cosas como `take 10 [0..]` o `map (*2) [0..]`.

Veamos cómo construir listas infinitas de elementos más complejos.

Generación infinita

En la clase anterior vimos que gracias a la **evaluación lazy** podemos trabajar con **estructuras de datos infinitas**.

Por ejemplo...

`[0..]` denota la lista de todos los números enteros no negativos y la podemos usar (con cuidado) como una lista cualquiera, para hacer cosas como `take 10 [0..]` o `map (*2) [0..]`.

Veamos cómo construir listas infinitas de elementos más complejos.

(Sugerencia: para probar en Haskell si efectivamente definimos la lista infinita que queríamos, lógicamente no podemos imprimirla entera, así que una buena idea es usar `take`.)

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

En un primer intento se nos podría ocurrir definirla como

```
[(x,y) | x <- [0..], y <- [0..]]
```

Pero esto no está bien, ¿por qué?

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

En un primer intento se nos podría ocurrir definirla como

```
[(x,y) | x <- [0..], y <- [0..]]
```

Pero esto no está bien, ¿por qué?
Porque nos genera la lista [(0,0), (0,1), (0,2), (0,3), ...] y entonces no podríamos acceder nunca (en tiempo finito), por ejemplo, al elemento (1,1).

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Teniendo en cuenta que tenemos que poder acceder (en tiempo finito) a cada uno de los pares de la lista, ¿cómo hacemos para definirla?

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Teniendo en cuenta que tenemos que poder acceder (en tiempo finito) a cada uno de los pares de la lista, ¿cómo hacemos para definirla?
Pista: repasar conjuntos numerables. ¿Cómo hacíamos para probar que $\mathbb{N}\times\mathbb{N}$ es numerable?

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Teniendo en cuenta que tenemos que poder acceder (en tiempo finito) a cada uno de los pares de la lista, ¿cómo hacemos para definirla?
Pista: repasar conjuntos numerables. ¿Cómo hacíamos para probar que $\mathbb{N}\times\mathbb{N}$ es numerable?
Una forma es pensar en la suma de ambas coordenadas. Dado un número s, sólo hay finitos pares que suman s. Entonces una buena idea es buscar todos los pares que sumen s para cada s de 0 a infinito.

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Teniendo en cuenta que tenemos que poder acceder (en tiempo finito) a cada uno de los pares de la lista, ¿cómo hacemos para definirla?
Pista: repasar conjuntos numerables. ¿Cómo hacíamos para probar que $\mathbb{N}\times\mathbb{N}$ es numerable?
Una forma es pensar en la suma de ambas coordenadas. Dado un número s, sólo hay finitos pares que suman s. Entonces una buena idea es buscar todos los pares que sumen s para cada s de 0 a infinito.

Solución

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Generación infinita

Ejercicio

Definir una lista (infinita) `pares :: [(Int, Int)]` que contenga todos los pares de números enteros no negativos (sin repetir).

Vamos a definirla usando listas por comprensión.

Teniendo en cuenta que tenemos que poder acceder (en tiempo finito) a cada uno de los pares de la lista, ¿cómo hacemos para definirla?
Pista: repasar conjuntos numerables. ¿Cómo hacíamos para probar que $\mathbb{N} \times \mathbb{N}$ es numerable?
Una forma es pensar en la suma de ambas coordenadas. Dado un número s, sólo hay finitos pares que suman s. Entonces una buena idea es buscar todos los pares que sumen s para cada s de 0 a infinito.

Solución

```
pares = [(x,suma-x) | suma<-[0..], x<-[0..suma]]
```

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Al igual que antes, hay que pensar bien cómo hacemos para que la lista permita acceder (en tiempo finito) a cada elemento.

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Al igual que antes, hay que pensar bien cómo hacemos para que la lista permita acceder (en tiempo finito) a cada elemento.
Pista: en este caso puede complicarse más porque son triplas, pero la idea es similar a la de pares.

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Al igual que antes, hay que pensar bien cómo hacemos para que la lista permita acceder (en tiempo finito) a cada elemento.
Pista: en este caso puede complicarse más porque son triplas, pero la idea es similar a la de pares.
Pista 2: tenemos que pensar en más sumas.

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Al igual que antes, hay que pensar bien cómo hacemos para que la lista permita acceder (en tiempo finito) a cada elemento.
Pista: en este caso puede complicarse más porque son triplas, pero la idea es similar a la de pares.
Pista 2: tenemos que pensar en más sumas.

Solución

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Generación infinita

Ejercicio

Definir una lista (infinita) `triplas :: [(Int, Int, Int)]` que contenga todas las triplas de números enteros no negativos (sin repetir).

Al igual que antes, hay que pensar bien cómo hacemos para que la lista permita acceder (en tiempo finito) a cada elemento.
Pista: en este caso puede complicarse más porque son triplas, pero la idea es similar a la de pares.
Pista 2: tenemos que pensar en más sumas.

Solución

```
triplas = [(x,y,z) | sumaXYZ<-[0..], sumaXY<-[0..sumaXYZ],
x<-[0..sumaXY], let y=sumaXY-x, let z=sumaXYZ-sumaXY]
```

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

Pista 1: la lista vacía suma 0.

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

Pista 1: la lista vacía suma 0.

Pista 2: para cualquier n, `listasQueSuman n` es una lista finita.

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

Pista 1: la lista vacía suma 0.
Pista 2: para cualquier n, `listasQueSuman n` es una lista finita.
Pista 3: todos los elementos de `listasQueSuman n` tienen que estar entre 1 y n.

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

Pista 1: la lista vacía suma 0.
Pista 2: para cualquier n, `listasQueSuman n` es una lista finita.
Pista 3: todos los elementos de `listasQueSuman n` tienen que estar entre 1 y n.
Pista 4 (última): si `x:xs` pertenece a `listasQueSuman n`, ¿qué podemos decir de `xs`?

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.
2. `listasPositivas :: [[Int]]` que contenga todas las listas finitas de enteros mayores o iguales que 1.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.
2. `listasPositivas :: [[Int]]` que contenga todas las listas finitas de enteros mayores o iguales que 1.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.
2. `listasPositivas :: [[Int]]` que contenga todas las listas finitas de enteros mayores o iguales que 1.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

Pista 1: usar `listasQueSuman`.

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.
2. `listasPositivas :: [[Int]]` que contenga todas las listas finitas de enteros mayores o iguales que 1.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

Pista 1: usar `listasQueSuman`.
Pista 2 (última): recordar la idea de `pares`.

Generación infinita

Ejercicios

Definir:

1. `listasQueSuman :: Int -> [[Int]]` que, dado un número entero $n \geq 0$, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n.
2. `listasPositivas :: [[Int]]` que contenga todas las listas finitas de enteros mayores o iguales que 1.

Soluciones

```
listasQueSuman 0 = [[]]
listasQueSuman n = [x:xs | x <- [1..n], xs <-
listasQueSuman (n-x)]
```

```
listasPositivas = [xs | n <- [1..], xs <- listasQueSuman n]
```

Ejercicios para resolver en el aire

Definir las siguientes funciones sin usar recursión explícita:

- `negar :: [[Char]] -> [[Char]]` que, dada una lista de palabras, le agrega "in" adelante a todas. Por ejemplo `negar [''util'', ''creible'']` da `[''inutil'', ''increible'']`
- `sinVacias :: [[a]] -> [[a]]` que, dada una lista de listas, devuelve las que no son vacías (en el mismo orden).

Ejercicios para resolver en el aire

Definir las siguientes funciones sin usar recursión explícita:

- `negar :: [[Char]] -> [[Char]]` que, dada una lista de palabras, le agrega "in" adelante a todas. Por ejemplo `negar [''util'', ''creible'']` da `[''inutil'', ''increible'']`
- `sinVacias :: [[a]] -> [[a]]` que, dada una lista de listas, devuelve las que no son vacías (en el mismo orden).

Soluciones

```
negar = map (''in''++)

sinVacias = filter (not . null)
```

Cuando map y filter no alcanzan

¿Y si queremos definir las siguientes funciones?

- `length :: [a] -> Int`, que calcula la longitud de una lista.
- `all :: (a -> Bool) -> [a] -> Bool`, que decide si todos los elementos de una lista cumplen una cierta propiedad.
- `concat :: [[a]] -> [a]`, que dada una lista de listas, devuelve la lista que resulta de concatenarlas en orden.

Cuando map y filter no alcanzan

¿Y si queremos definir las siguientes funciones?

- `length :: [a] -> Int`, que calcula la longitud de una lista.
- `all :: (a -> Bool) -> [a] -> Bool`, que decide si todos los elementos de una lista cumplen una cierta propiedad.
- `concat :: [[a]] -> [a]`, que dada una lista de listas, devuelve la lista que resulta de concatenarlas en orden.

Sólo con map y filter no podemos.

Cuando map y filter no alcanzan

¿Y si queremos definir las siguientes funciones?

- `length :: [a] -> Int`, que calcula la longitud de una lista.
- `all :: (a -> Bool) -> [a] -> Bool`, que decide si todos los elementos de una lista cumplen una cierta propiedad.
- `concat :: [[a]] -> [a]`, que dada una lista de listas, devuelve la lista que resulta de concatenarlas en orden.

Sólo con map y filter no podemos.

Veamos cómo resolverlas usando recursión explícita.

Soluciones

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

all _ [] = True
all p (x:xs) = p x && all p xs

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

¿Qué estamos haciendo en todos estos casos?

Cuando map y filter no alcanzan

Soluciones

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

all _ [] = True
all p (x:xs) = p x && all p xs

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

¿Qué estamos haciendo en todos estos casos? Tenemos:

Cuando map y filter no alcanzan

Soluciones

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

all _ [] = True
all p (x:xs) = p x && all p xs

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

¿Qué estamos haciendo en todos estos casos? Tenemos:

- Un valor para el caso base (lista vacía).

Cuando map y filter no alcanzan

Soluciones

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

all _ [] = True
all p (x:xs) = p x && all p xs

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

¿Qué estamos haciendo en todos estos casos? Tenemos:

- Un valor para el caso base (lista vacía).
- Y para el caso recursivo: una función que utiliza la cabeza de la lista y un llamado recursivo con la cola de la lista.

Cuando map y filter no alcanzan

Soluciones

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

all _ [] = True
all p (x:xs) = p x && all p xs

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

¿Qué estamos haciendo en todos estos casos? Tenemos:

- Un valor para el caso base (lista vacía).
- Y para el caso recursivo: una función que utiliza la cabeza de la lista y un llamado recursivo con la cola de la lista.

¿Cómo podemos generalizarlo?

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

- toma una función que representa el caso recursivo (noten que esta función toma la cabeza de la lista y un llamado recursivo sobre la cola de la lista) y también

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

- toma una función que representa el caso recursivo (noten que esta función toma la cabeza de la lista y un llamado recursivo sobre la cola de la lista) y también
- toma un valor que representa el caso base, y

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

- toma una función que representa el caso recursivo (noten que esta función toma la cabeza de la lista y un llamado recursivo sobre la cola de la lista) y también
- toma un valor que representa el caso base, y
- devuelve una función que sabe cómo reducir listas de a a un valor de tipo b.

foldr

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

- toma una función que representa el caso recursivo (noten que esta función toma la cabeza de la lista y un llamado recursivo sobre la cola de la lista) y también
- toma un valor que representa el caso base, y
- devuelve una función que sabe cómo reducir listas de a a un valor de tipo b.

Veamos un ejemplo.

foldr

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

> let suma = foldr (+) 0

foldr

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

```
> let suma = foldr (+) 0
> suma [1,2,3]
```

foldr

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

```
> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
```

⟶ Por definición de `suma`.

foldr

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

```
> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
⟶ 1 + (foldr (+) 0 [2,3])
```

⟶ Entramos al segundo caso de `foldr` con la lista `1:[2,3]`.

foldr

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

> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
⟶ 1 + (foldr (+) 0 [2,3])
⟶ 1 + (2 + (foldr (+) 0 [3]))

⟶ Porque para resolver la operación + primero hay que evaluar sus argumentos, en particular el `foldr` aplicado a la lista `2:[3]`.

foldr

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

```
> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
⟶ 1 + (foldr (+) 0 [2,3])
⟶ 1 + (2 + (foldr (+) 0 [3]))
⟶ 1 + (2 + (3 + (foldr (+) 0 [ ])))
```

⟶ De nuevo lo mismo.

foldr

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

> let suma = foldr (+) 0
> suma [1,2,3]
$\longrightarrow$ foldr (+) 0 [1,2,3]
$\longrightarrow$ 1 + (foldr (+) 0 [2,3])
$\longrightarrow$ 1 + (2 + (foldr (+) 0 [3]))
$\longrightarrow$ 1 + (2 + (3 + (foldr (+) 0 [ ])))
$\longrightarrow$ 1 + (2 + (3 + 0))

$\longrightarrow$ Porque llegamos al caso base de `foldr`.

foldr

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

> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
⟶ 1 + (foldr (+) 0 [2,3])
⟶ 1 + (2 + (foldr (+) 0 [3]))
⟶ 1 + (2 + (3 + (foldr (+) 0 [ ])))
⟶ 1 + (2 + (3 + 0))
⟶ 1 + (2 + 3)

⟶ Y resolvemos las sumas.

foldr

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

```
> let suma = foldr (+) 0
> suma [1,2,3]
⟶ foldr (+) 0 [1,2,3]
⟶ 1 + (foldr (+) 0 [2,3])
⟶ 1 + (2 + (foldr (+) 0 [3]))
⟶ 1 + (2 + (3 + (foldr (+) 0 [ ])))
⟶ 1 + (2 + (3 + 0))
⟶ 1 + (2 + 3)
⟶ 1 + 5
```

foldr

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

> let suma = foldr (+) 0
> suma [1,2,3]
$\longrightarrow$ foldr (+) 0 [1,2,3]
$\longrightarrow$ 1 + (foldr (+) 0 [2,3])
$\longrightarrow$ 1 + (2 + (foldr (+) 0 [3]))
$\longrightarrow$ 1 + (2 + (3 + (foldr (+) 0 [ ])))
$\longrightarrow$ 1 + (2 + (3 + 0))
$\longrightarrow$ 1 + (2 + 3)
$\longrightarrow$ 1 + 5
6

foldr

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

> let suma = foldr (+) 0
> suma [1,2,3]
$\longrightarrow$ foldr (+) 0 [1,2,3]
$\longrightarrow$ 1 + (foldr (+) 0 [2,3])
$\longrightarrow$ 1 + (2 + (foldr (+) 0 [3]))
$\longrightarrow$ 1 + (2 + (3 + (foldr (+) 0 [ ])))
$\longrightarrow$ 1 + (2 + (3 + 0))
$\longrightarrow$ 1 + (2 + 3)
$\longrightarrow$ 1 + 5
6

Fíjense cómo quedaron desplegadas las operaciones `(+)`. Por esta razón decimos que `foldr` acumula el resultado desde la derecha.

foldr

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

Ejercicios

Definir usando `foldr`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`
4. `concat :: [[a]] -> [a]`
5. `map :: (a -> b) -> [a] -> [b]`
6. `filter :: (a -> Bool) ->[a] -> [a]`

foldr

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

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

foldr

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

Soluciones

```haskell
producto = foldr (*) 1
```

foldr

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

Soluciones

```haskell
producto = foldr (*) 1
length = foldr ((+).(const 1)) 0
```

foldr

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

Soluciones

```
producto = foldr (*) 1
length = foldr ((+).(const 1)) 0
all p = foldr ((&&).p) True
```

foldr

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

Soluciones

```haskell
producto = foldr (*) 1
length = foldr ((+).(const 1)) 0
all p = foldr ((&&).p) True
concat = foldr (++) []
```

foldr

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

Soluciones

```
producto = foldr (*) 1
length = foldr ((+).(const 1)) 0
all p = foldr ((&&).p) True
concat = foldr (++) []
map f = foldr ((:).f) []
```

foldr

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

Soluciones

```haskell
producto = foldr (*) 1
length = foldr ((+).(const 1)) 0
all p = foldr ((&&).p) True
concat = foldr (++) []
map f = foldr ((:).f) []
filter p = foldr (\x r -> if p x then x:r else r) []
```

foldl

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

La función `foldl` es muy similar a `foldr` pero acumula desde la izquierda.

foldl

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

> let suma = foldl (+) 0

foldl

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

```
> let suma = foldl (+) 0
> suma [1,2,3]
```

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
$\longrightarrow$ foldl (+) 0 [1,2,3]

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
$\longrightarrow$ foldl (+) 0 [1,2,3]
$\longrightarrow$ foldl (+) (0+1) [2,3]

foldl

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

```
> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
```

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
$\longrightarrow$ foldl (+) 0 [1,2,3]
$\longrightarrow$ foldl (+) (0+1) [2,3]
$\longrightarrow$ foldl (+) ((0+1)+2) [3]
$\longrightarrow$ foldl (+) (((0+1)+2)+3) [ ]

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
⟶ foldl (+) (((0+1)+2)+3) [ ]
⟶ ((0+1)+2)+3

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
⟶ foldl (+) (((0+1)+2)+3) [ ]
⟶ ((0+1)+2)+3
⟶ (1+2)+3

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
⟶ foldl (+) (((0+1)+2)+3) [ ]
⟶ ((0+1)+2)+3
⟶ (1+2)+3
⟶ 3+3

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
⟶ foldl (+) (((0+1)+2)+3) [ ]
⟶ ((0+1)+2)+3
⟶ (1+2)+3
⟶ 3+3
6

foldl

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

> let suma = foldl (+) 0
> suma [1,2,3]
⟶ foldl (+) 0 [1,2,3]
⟶ foldl (+) (0+1) [2,3]
⟶ foldl (+) ((0+1)+2) [3]
⟶ foldl (+) (((0+1)+2)+3) [ ]
⟶ **((0+1)+2)+3**
⟶ (1+2)+3
⟶ 3+3
6

Noten la diferencia con `foldr` que hacía **1+(2+(3+0))**.

foldl

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

Ejercicios

Definir usando `foldl`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`

foldl

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

Ejercicios

Definir usando `foldl`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

foldl

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

Ejercicios

Definir usando `foldl`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`

Soluciones

```haskell
producto = foldl (*) 1
```

foldl

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

Ejercicios

Definir usando `foldl`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`

Soluciones

```
producto = foldl (*) 1
length = foldl (\r _ -> r+1) 0
```

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

Ejercicios

Definir usando `foldl`:

1. `producto :: [Int] -> Int`
2. `length :: [a] -> Int`
3. `all :: (a -> Bool) -> [a] -> Bool`

Soluciones

```haskell
producto = foldl (*) 1
length = foldl (\r _ -> r+1) 0
all p = foldl (\r x -> r && p x) True
```

foldr y foldl

Ejercicios

¿Qué computan estas funciones?

1.
```haskell
f :: [a] -> [a]
f = foldr (:) []
```
2.
```haskell
g :: [a] -> [a]
g = foldl (flip (:)) []
```

foldr y foldl

Ejercicios

¿Qué computan estas funciones?

1. ```haskell
   f :: [a] -> [a]
   f = foldr (:) []
   ```
2. ```haskell
   g :: [a] -> [a]
   g = foldl (flip (:)) []
   ```

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Pista: ejecutar, a mano, paso por paso en algún caso particular.

foldr y foldl

Ejercicios

¿Qué computan estas funciones?

1. ```haskell
   f :: [a] -> [a]
   f = foldr (:) []
   ```
2. ```haskell
   g :: [a] -> [a]
   g = foldl (flip (:)) []
   ```

Soluciones

1. Devuelve la misma lista.

foldr y foldl

Ejercicios

¿Qué computan estas funciones?

1. ```
   f :: [a] -> [a]
   f = foldr (:) []
   ```
2. ```
   g :: [a] -> [a]
   g = foldl (flip (:)) []
   ```

Soluciones

1. Devuelve la misma lista.
2. Devuelve la lista invertida (equivalente a `reverse`).

Ejercicios

¿Qué computan estas funciones?

1. ```
   f :: [a] -> [a]
   f = foldr (:) []
   ```
2. ```
   g :: [a] -> [a]
   g = foldl (flip (:)) []
   ```
   $\leftarrow$ Importante!

Soluciones

1. Devuelve la misma lista.
2. Devuelve la lista invertida (equivalente a `reverse`).

Este es un buen ejemplo para ver las diferencias entre `foldr` y `foldl`.

foldr y foldl

Ejercicios

¿Qué computan estas funciones?

1. ```
   f :: [a] -> [a]
   f = foldr (:) []
   ```
2. ```
   g :: [a] -> [a]
   g = foldl (flip (:)) []  ← Importante!
   ```

Soluciones

1. Devuelve la misma lista.
2. Devuelve la lista invertida (equivalente a `reverse`).

Este es un buen ejemplo para ver las diferencias entre `foldr` y `foldl`.
Pero tienen otra diferencia muy importante...

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?

¿Cómo reduce `all even [0..]`?

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?

¿Cómo reduce `all even [0..]`?

Usando foldr

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`
$\longrightarrow$ `(even 0) && (foldr ((&&).even) True [1..])`

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`
$\longrightarrow$ `(even 0) && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `True && (foldr ((&&).even) True [1..])`

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`
$\longrightarrow$ `(even 0) && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `True && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `foldr ((&&).even) True [1..]`

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

```
⟶ foldr ((&&).even) True [0..]
⟶ (even 0) && (foldr ((&&).even) True [1..])
⟶ True && (foldr ((&&).even) True [1..])
⟶ foldr ((&&).even) True [1..]
⟶ (even 1) && (foldr ((&&).even) True [2..])
```

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`
$\longrightarrow$ `(even 0) && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `True && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `foldr ((&&).even) True [1..]`
$\longrightarrow$ `(even 1) && (foldr ((&&).even) True [2..])`
$\longrightarrow$ `False && (foldr ((&&).even) True [2..])`

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?

¿Cómo reduce `all even [0..]`?

Usando foldr

```
⟶ foldr ((&&).even) True [0..]
⟶ (even 0) && (foldr ((&&).even) True [1..])
⟶ True && (foldr ((&&).even) True [1..])
⟶ foldr ((&&).even) True [1..]
⟶ (even 1) && (foldr ((&&).even) True [2..])
⟶ False && (foldr ((&&).even) True [2..])
⟶ False
```

Usando foldl

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

```
⟶ foldr ((&&).even) True [0..]
⟶ (even 0) && (foldr ((&&).even) True [1..])
⟶ True && (foldr ((&&).even) True [1..])
⟶ foldr ((&&).even) True [1..]
⟶ (even 1) && (foldr ((&&).even) True [2..])
⟶ False && (foldr ((&&).even) True [2..])
⟶ False
```

Usando foldl

```
⟶ foldl (\r x -> r&&(even x)) True [0..]
```

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

```
⟶ foldr ((&&).even) True [0..]
⟶ (even 0) && (foldr ((&&).even) True [1..])
⟶ True && (foldr ((&&).even) True [1..])
⟶ foldr ((&&).even) True [1..]
⟶ (even 1) && (foldr ((&&).even) True [2..])
⟶ False && (foldr ((&&).even) True [2..])
⟶ False
```

Usando foldl

```
⟶ foldl (\r x -> r&&(even x)) True [0..]
⟶ foldl (\r x -> r&&(even x)) (True&&(even 0)) [1..]
```

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?

¿Cómo reduce `all even [0..]`?

Usando foldr

$\longrightarrow$ `foldr ((&&).even) True [0..]`
$\longrightarrow$ `(even 0) && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `True && (foldr ((&&).even) True [1..])`
$\longrightarrow$ `foldr ((&&).even) True [1..]`
$\longrightarrow$ `(even 1) && (foldr ((&&).even) True [2..])`
$\longrightarrow$ `False && (foldr ((&&).even) True [2..])`
$\longrightarrow$ `False`

Usando foldl

$\longrightarrow$ `foldl (\r x -> r&&(even x)) True [0..]`
$\longrightarrow$ `foldl (\r x -> r&&(even x)) (True&&(even 0)) [1..]`
$\longrightarrow$ `foldl (\r x -> r&&(even x)) (True&&(even 0)&&(even 1)) [2..]`

Esquemas de recursión sobre listas infinitas

¿Qué sucede al usar `foldr` y `foldl` en las listas infinitas?
¿Cómo reduce `all even [0..]`?

Usando foldr

```
⟶ foldr ((&&).even) True [0..]
⟶ (even 0) && (foldr ((&&).even) True [1..])
⟶ True && (foldr ((&&).even) True [1..])
⟶ foldr ((&&).even) True [1..]
⟶ (even 1) && (foldr ((&&).even) True [2..])
⟶ False && (foldr ((&&).even) True [2..])
⟶ False
```

Usando foldl

```
⟶ foldl (\r x -> r&&(even x)) True [0..]
⟶ foldl (\r x -> r&&(even x)) (True&&(even 0)) [1..]
⟶ foldl (\r x -> r&&(even x)) (True&&(even 0)&&(even 1)) [2..]
...
```

Esquemas de recursión sobre listas infinitas

Conclusión:

Esquemas de recursión sobre listas infinitas

Conclusión:

- `foldl` reduce a otro `foldl` en el caso recursivo.

Esquemas de recursión sobre listas infinitas

Conclusión:

- `foldl` reduce a otro `foldl` en el caso recursivo.

  Por lo tanto, si la lista es infinita entonces nunca termina.

Conclusión:

- `foldl` reduce a otro `foldl` en el caso recursivo.
  Por lo tanto, si la lista es infinita entonces nunca termina.

- En cambio, con `foldr` tenemos

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

Esquemas de recursión sobre listas infinitas

Conclusión:

- `foldl` reduce a otro `foldl` en el caso recursivo.

  Por lo tanto, si la lista es infinita entonces nunca termina.

- En cambio, con `foldr` tenemos

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

  Como la función "de más afuera" es `f`, primero intenta reducir `f` antes de meterse con el otro `foldr`.

Esquemas de recursión sobre listas infinitas

Conclusión:

- `foldl` reduce a otro `foldl` en el caso recursivo.

  Por lo tanto, si la lista es infinita entonces nunca termina.

- En cambio, con `foldr` tenemos

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

  Como la función "de más afuera" es `f`, primero intenta reducir `f` antes de meterse con el otro `foldr`.

  Por lo tanto, `foldr` puede trabajar bien con listas infinitas.

  (Con algunas restricciones dependiendo de `f` y el contenido de la lista que reciba.)

foldr1 y foldl1

¿Podemos usar `foldr` para obtener el máximo de una lista (no vacía)?
¿Cuál sería el caso base?

foldr1 y foldl1

¿Podemos usar `foldr` para obtener el máximo de una lista (no vacía)? ¿Cuál sería el caso base?

Para este tipo de situaciones tenemos las funciones `foldr1` y `foldl1`, que permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el último elemento de la lista.
- `foldl1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el primer elemento de la lista.

Para ambas, la lista no debe ser vacía y el tipo del resultado debe ser el de los elementos de la lista.

foldr1 y foldl1

- `foldr1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el último elemento de la lista.
- `foldl1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el primer elemento de la lista.

Ejercicios

Definir las siguientes funciones:

1. `last :: [a] -> a`
2. `maximum :: Ord a => [a] -> a`

foldr1 y foldl1

- `foldr1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el último elemento de la lista.
- `foldl1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el primer elemento de la lista.

Ejercicios

Definir las siguientes funciones:

1. `last :: [a] -> a`
2. `maximum :: Ord a => [a] -> a`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

foldr1 y foldl1

- `foldr1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el último elemento de la lista.
- `foldl1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el primer elemento de la lista.

Ejercicios

Definir las siguientes funciones:

1. `last :: [a] -> a`
2. `maximum :: Ord a => [a] -> a`

Soluciones

```
last = foldr1 (\_ r -> r)
```

foldr1 y foldl1

- `foldr1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el último elemento de la lista.
- `foldl1 :: (a -> a -> a) -> [a] -> a`
  toma como caso base el primer elemento de la lista.

Ejercicios

Definir las siguientes funciones:

1. `last :: [a] -> a`
2. `maximum :: Ord a => [a] -> a`

Soluciones

```
last = foldr1 (\_ r -> r)
maximum = foldr1 max
```

¿Se puede definir usando sólo `foldr` la función

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
```

que, dado un elemento x y una lista ℓ ordenada de forma creciente, inserte x en ℓ preservando el orden?

recr

¿Se puede definir usando sólo `foldr` la función

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
```

que, dado un elemento x y una lista ℓ ordenada de forma creciente, inserte x en ℓ preservando el orden?

Sí, pero la solución es un tanto rebuscada (ejercicio opcional). Nos sería mucho más fácil (y capturaría mejor la esencia de la función) si tuviésemos acceso a la cola de la lista original, no sólo al resultado de la llamada recursiva sobre ella (como tenemos con fold).

recr

¿Se puede definir usando sólo `foldr` la función

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
```

que, dado un elemento x y una lista ℓ ordenada de forma creciente, inserte x en ℓ preservando el orden?

Sí, pero la solución es un tanto rebuscada (ejercicio opcional). Nos sería mucho más fácil (y capturaría mejor la esencia de la función) si tuviésemos acceso a la cola de la lista original, no sólo al resultado de la llamada recursiva sobre ella (como tenemos con fold).

Para este tipo de situaciones consideramos otro esquema de recursión:

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

recr

¿Se puede definir usando sólo `foldr` la función

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
```

que, dado un elemento x y una lista ℓ ordenada de forma creciente, inserte x en ℓ preservando el orden?

Sí, pero la solución es un tanto rebuscada (ejercicio opcional). Nos sería mucho más fácil (y capturaría mejor la esencia de la función) si tuviésemos acceso a la cola de la lista original, no sólo al resultado de la llamada recursiva sobre ella (como tenemos con fold).

Para este tipo de situaciones consideramos otro esquema de recursión:

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

Notemos que ahora la función para el caso recursivo también toma como parámetro la cola de la lista.

recr

¿Se puede definir usando sólo `foldr` la función

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
```

que, dado un elemento x y una lista ℓ ordenada de forma creciente, inserte x en ℓ preservando el orden?

Sí, pero la solución es un tanto rebuscada (ejercicio opcional). Nos sería mucho más fácil (y capturaría mejor la esencia de la función) si tuviésemos acceso a la cola de la lista original, no sólo al resultado de la llamada recursiva sobre ella (como tenemos con fold).

Para este tipo de situaciones consideramos otro esquema de recursión:

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

Notemos que ahora la función para el caso recursivo también toma como parámetro la cola de la lista.

(Nota: `recr` no viene definido en el Standard Prelude de Haskell, como sí vienen `foldr` y `foldl`. De hecho, `recr` se puede definir en términos de `foldr`, pero este ejercicio ya se escapa de los objetivos de la materia.)

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.
Hay pistas para `take`.

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```haskell
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
```

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
```

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
```

Pista para `take n xs`**:** pensar en un fold que "devuelva" una función que dado un entero n sepa tomar n elementos de una lista determinada.

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
```

Otra pista: ¿cuál es el tipo de ese fold?

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
```

Otra pista: ¿cuál es el tipo de ese fold?
En este caso "b" sería `Int -> [a]`. Así que queda:
`(a -> (Int -> [a]) -> Int -> [a]) -> (Int -> [a]) -> [a] -> Int -> [a]`

recr

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
```

Otra pista (la última): el caso base sería una función que siempre devuelve la lista vacía.

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

Ejercicios

Definir las siguientes funciones usando `recr` o `foldr`:

1. `insertarOrdenado :: Ord a => a -> [a] -> [a]`
2. `pertenece :: Eq a => a -> [a] -> Bool`
3. `take :: Int -> [a] -> [a]`

Soluciones

```haskell
insertarOrdenado e = recr (\x xs r -> if e<x then e:x:xs else x:r) [e]
pertenece e = foldr (\x r -> x==e || r) False
take n xs = foldr (\x r -> \m -> if m==0 then [] else x:(r (m-1)))
(const []) xs n
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.

Algunos ejemplos

```haskell
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.
- Pueden involucrar una **combinación de otros tipos**.

Algunos ejemplos

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.
- Pueden involucrar una combinación de otros tipos.
- Están formados por uno o más **constructores**.

Algunos ejemplos

```haskell
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.
- Pueden involucrar una combinación de otros tipos.
- Están formados por uno o más constructores.
- Cada constructor puede o no tener **argumentos**.

Algunos ejemplos

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.
- Pueden involucrar una combinación de otros tipos.
- Están formados por uno o más constructores.
- Cada constructor puede o no tener argumentos.
- Los argumentos de los constructores pueden ser **recursivos**.

Algunos ejemplos

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

- Se definen mediante la cláusula `data`.
- Pueden involucrar una combinación de otros tipos.
- Están formados por uno o más constructores.
- Cada constructor puede o no tener argumentos.
- Los argumentos de los constructores pueden ser recursivos.
- Se inspeccionan usando **pattern matching**.

Algunos ejemplos

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Tipos algebraicos

¿Qué son los constructores?

Tipos algebraicos

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Tipos algebraicos

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
```

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
Nothing :: Maybe a
```

Tipos algebraicos

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
Nothing :: Maybe a

> :t Just
```

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
Nothing :: Maybe a

> :t Just
Just :: a -> Maybe a
```

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
Nothing :: Maybe a

> :t Just
Just :: a -> Maybe a

> :t Just True
```

Tipos algebraicos

¿Qué son los constructores?

Podemos pensar a los constructores como **constantes y funciones** que permiten construir elementos de un tipo.

Veamos su tipo...

```
> :t Nothing
Nothing :: Maybe a

> :t Just
Just :: a -> Maybe a

> :t Just True
Just True :: Maybe Bool
```

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con proposiciones lógicas que sólo usan $\neg$ y $\wedge$:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con proposiciones lógicas que sólo usan $\neg$ y $\wedge$:

```
data Prop = Var String | Not Prop | And Prop Prop
```

`Var v` representa una variable proposicional de nombre `v`, `Not p` representa la negación de una proposición `p`, y `And p q` representa la conjunción de las proposiciones `p` y `q`.

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con proposiciones lógicas que sólo usan $\neg$ y $\wedge$:

```
data Prop = Var String | Not Prop | And Prop Prop
```

`Var v` representa una variable proposicional de nombre `v`, `Not p` representa la negación de una proposición `p`, y `And p q` representa la conjunción de las proposiciones `p` y `q`.

Objetivo

Queremos definir el esquema de recursión estructural (fold) para `Prop`.

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con proposiciones lógicas que sólo usan $\neg$ y $\wedge$:

```
data Prop = Var String | Not Prop | And Prop Prop
```

`Var v` representa una variable proposicional de nombre `v`, `Not p` representa la negación de una proposición `p`, y `And p q` representa la conjunción de las proposiciones `p` y `q`.

Objetivo

Queremos definir el esquema de recursión estructural (fold) para `Prop`.

¿Cómo hacemos?

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con proposiciones lógicas que sólo usan $\neg$ y $\wedge$:

```
data Prop = Var String | Not Prop | And Prop Prop
```

`Var v` representa una variable proposicional de nombre `v`, `Not p` representa la negación de una proposición `p`, y `And p q` representa la conjunción de las proposiciones `p` y `q`.

Objetivo

Queremos definir el esquema de recursión estructural (fold) para `Prop`.

¿Cómo hacemos? Empecemos por recordar y analizar en profundidad al fold de listas...

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

Pista: pensar cuáles son los constructores del tipo `[a]`.

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?
Los constructores de las listas son `[]` y `(:)` (aunque no tengan la apariencia típica de un constructor).

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?
Los constructores de las listas son `[]` y `(:)` (aunque no tengan la apariencia típica de un constructor).
Entonces tenemos el parámetro `z` para el constructor `[]`, y `f` para `(:)`.

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

Los constructores de las listas son `[]` y `(:)` (aunque no tengan la apariencia típica de un constructor).

Entonces tenemos el parámetro `z` para el constructor `[]`, y `f` para `(:)`.

Como `[]` no tiene argumentos, `z` tiene tipo `b` (lo que construye el fold).

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?
Los constructores de las listas son `[]` y `(:)` (aunque no tengan la apariencia típica de un constructor).
Entonces tenemos el parámetro `z` para el constructor `[]`, y `f` para `(:)`.
Como `[]` no tiene argumentos, `z` tiene tipo `b` (lo que construye el fold).
Mientras que `(:)` sí tiene argumentos: la cabeza de la lista (de tipo `a`) y la cola de la lista (de tipo `[a]`). Y el segundo argumento es recursivo!

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

Los constructores de las listas son `[]` y `(:)` (aunque no tengan la apariencia típica de un constructor).

Entonces tenemos el parámetro `z` para el constructor `[]`, y `f` para `(:)`.

Como `[]` no tiene argumentos, `z` tiene tipo `b` (lo que construye el fold).

Mientras que `(:)` sí tiene argumentos: la cabeza de la lista (de tipo `a`) y la cola de la lista (de tipo `[a]`). Y el segundo argumento es recursivo!

Por lo tanto, `f` tiene tipo `a -> b -> b` porque recibe el argumento no recursivo de `(:)` (la cabeza de la lista, de tipo `a`) y el llamado recursivo (que es de tipo `b`) sobre el argumento recursivo, y devuelve lo que construye el fold (algo de tipo `b`).

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

En general

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además la estructura que va a recorrer.

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

En general

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además la estructura que va a recorrer.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. ¡Y todos van a devolver lo mismo!

Fold sobre estructuras nuevas

Recordemos `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

En general

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además la estructura que va a recorrer.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. ¡Y todos van a devolver lo mismo!

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: ?
```

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: ?
```

Empecemos por lo fácil: necesitamos tres argumentos que se correspondan con los tres constructores de `Prop`, más uno por la proposición a la cual aplicaremos el fold, y no olvidemos lo que devuelve (que puede ser cualquier cosa).

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: ? -> ? -> ? -> Prop -> a
```

Empecemos por lo fácil: necesitamos tres argumentos que se correspondan con los tres constructores de `Prop`, más uno por la proposición a la cual aplicaremos el fold, y no olvidemos lo que devuelve (que puede ser cualquier cosa).

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: ? -> ? -> ? -> Prop -> a
```

Sigamos con el constructor `Var`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: ? -> ? -> ? -> Prop -> a
```

Sigamos con el constructor `Var`.
Necesitamos una función que dado un string (el argumento no recursivo de `Var`) nos permita construir un objeto de tipo `a` (porque es lo que devuelve el fold).

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> ? -> ? -> Prop -> a
```

Sigamos con el constructor `Var`.
Necesitamos una función que dado un string (el argumento no recursivo de `Var`) nos permita construir un objeto de tipo `a` (porque es lo que devuelve el fold).

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> ? -> ? -> Prop -> a
```

Pasemos a `Not`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> ? -> ? -> Prop -> a
```

Pasemos a `Not`.
Este constructor tiene un solo argumento, el cual es recursivo. Por lo tanto, necesitamos una función que, dado el resultado de la llamada recursiva sobre el argumento de `Not`, nos devuelva un nuevo resultado. Es decir, el tipo de la función es `a -> a`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> ? -> Prop -> a
```

Pasemos a `Not`.
Este constructor tiene un solo argumento, el cual es recursivo. Por lo tanto, necesitamos una función que, dado el resultado de la llamada recursiva sobre el argumento de `Not`, nos devuelva un nuevo resultado. Es decir, el tipo de la función es `a -> a`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> ? -> Prop -> a
```

Pasemos a `And`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> ? -> Prop -> a
```

Pasemos a `And`.
Este constructor tiene dos argumentos recursivos. Por lo tanto, necesitamos una función que, dados los resultados de las llamadas recursivas sobre ambos argumentos de `And`, nos devuelva algo de tipo `a`. Es decir, el tipo de la función es `a -> a -> a`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

Pasemos a `And`.
Este constructor tiene dos argumentos recursivos. Por lo tanto, necesitamos una función que, dados los resultados de las llamadas recursivas sobre ambos argumentos de `And`, nos devuelva algo de tipo `a`. Es decir, el tipo de la función es `a -> a -> a`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd ? = ?
```

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd ? = ?
```

Hacemos pattern matching.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = ?
foldProp casoVar casoNot casoAnd (Not p) = ?
foldProp casoVar casoNot casoAnd (And p q) = ?
```

Hacemos pattern matching.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```haskell
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```haskell
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```haskell
foldProp casoVar casoNot casoAnd (Var v) = ?
foldProp casoVar casoNot casoAnd (Not p) = ?
foldProp casoVar casoNot casoAnd (And p q) = ?
```

¿Qué hay que hacer en el caso del constructor `Var`?

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = ?
foldProp casoVar casoNot casoAnd (And p q) = ?
```

Simplemente aplicar `casoVar` al nombre de variable `v`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```haskell
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```haskell
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```haskell
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = ?
foldProp casoVar casoNot casoAnd (And p q) = ?
```

¿Qué hay que hacer en el caso del constructor `Not`?

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = casoNot (foldProp casoVar
casoNot casoAnd p)
foldProp casoVar casoNot casoAnd (And p q) = ?
```

Aplicamos `casoNot` a la llamada recursiva sobre `p`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = casoNot (foldProp casoVar
casoNot casoAnd p)
foldProp casoVar casoNot casoAnd (And p q) = ?
```

¿Qué hay que hacer en el caso del constructor `And`?

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = casoNot (foldProp casoVar
casoNot casoAnd p)
foldProp casoVar casoNot casoAnd (And p q) = casoAnd (foldProp casoVar
casoNot casoAnd p) (foldProp casoVar casoNot casoAnd q)
```

Aplicamos `casoAnd` a las llamadas recursivas sobre `p` y `q`.

Fold sobre estructuras nuevas

Ahora analicemos el nuevo tipo de datos:

```
data Prop = Var String | Not Prop | And Prop Prop
```

Tenemos dos constructores recursivos (`Not` y `And`) y otro no (`Var`).

Debemos preguntarnos...

1) ¿Cuál va a ser el tipo de nuestro fold?

```
foldProp :: (String -> a) -> (a -> a) -> (a -> a -> a) -> Prop -> a
```

2) ¿Y la implementación?

```
foldProp casoVar casoNot casoAnd (Var v) = casoVar v
foldProp casoVar casoNot casoAnd (Not p) = casoNot (foldProp casoVar
casoNot casoAnd p)
foldProp casoVar casoNot casoAnd (And p q) = casoAnd (foldProp casoVar
casoNot casoAnd p) (foldProp casoVar casoNot casoAnd q)
```

(Para no repetir tantas veces "`foldProp casoVar casoNot casoAnd`" podemos poner `where rec = foldProp casoVar casoNot casoAnd`.)

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Hay resolución paso por paso!

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp ?
```

¿Cómo empezamos a pensar esta solución?

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp ?
```

¿Cómo empezamos a pensar esta solución?

Bueno, en principio sabemos que vamos a usar `foldProp` (ya sea porque simplemente lo pide el enunciado o porque llegamos a la conclusión de que los esquemas de recursión está buenísimos y los queremos aplicar a todo lo que se pueda).

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = ?
              casoNot = ?
              casoAnd = ?
```

¿Cómo empezamos a pensar esta solución?
Bueno, en principio sabemos que vamos a usar `foldProp` (ya sea porque simplemente lo pide el enunciado o porque llegamos a la conclusión de que los esquemas de recursión está buenísimos y los queremos aplicar a todo lo que se pueda).
Así que podemos arrancar de esta forma.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = ?
              casoNot = ?
              casoAnd = ?
```

¿Qué es lo que queremos hacer cuando nos encontramos con una expresión de la pinta `Var v`?

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = ?
              casoNot = ?
              casoAnd = ?
```

¿Qué es lo que queremos hacer cuando nos encontramos con una expresión de la pinta `Var v`?

En este caso vamos a querer conocer el valor de verdad de esa variable, y esto nos lo dice la función `asig`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = ?
              casoAnd = ?
```

¿Qué es lo que queremos hacer cuando nos encontramos con una expresión de la pinta `Var v`?

En este caso vamos a querer conocer el valor de verdad de esa variable, y esto nos lo dice la función `asig`.

(Podríamos haber puesto `\v -> asig v`, pero noten que esto es exactamente lo mismo que `asig`.)

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = ?
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = ?
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?
Bueno, vamos a querer negar el valor de la proposición `p`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = ?
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?
Bueno, vamos a querer negar el valor de la proposición `p`.
Este valor lo conocemos gracias al llamado recursivo que hace `foldProp` y que se lo pasa a la función `casoNot`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = \r -> ?
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?
Bueno, vamos a querer negar el valor de la proposición `p`.
Este valor lo conocemos gracias al llamado recursivo que hace `foldProp` y que se lo pasa a la función `casoNot`. En otras palabras, `casoNot` es una función que siempre recibe el llamado recursivo sobre `p`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = \r -> not r
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?
Bueno, vamos a querer negar el valor de la proposición `p`.
Este valor lo conocemos gracias al llamado recursivo que hace `foldProp` y que se lo pasa a la función `casoNot`. En otras palabras, `casoNot` es una función que siempre recibe el llamado recursivo sobre `p`. Y lo que queremos hacer es negar (lógicamente) este resultado.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

¿Qué queremos hacer cuando nos encontramos con `Not p`?
Bueno, vamos a querer negar el valor de la proposición `p`.
Este valor lo conocemos gracias al llamado recursivo que hace `foldProp` y que se lo pasa a la función `casoNot`. En otras palabras, `casoNot` es una función que siempre recibe el llamado recursivo sobre `p`. Y lo que queremos hacer es negar (lógicamente) este resultado.
Esto es equivalente a la función `not`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`?

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`? Razonemos de forma parecida a las anteriores.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`?
Razonemos de forma parecida a las anteriores.
Sabemos que `casoAnd` es una función que recibe los resultados de evaluar `p` y `q`.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

`evalProp :: (String -> Bool) -> Prop -> Bool`

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`?
Razonemos de forma parecida a las anteriores.
Sabemos que `casoAnd` es una función que recibe los resultados de evaluar `p` y `q`.
¿Y con estos resultados qué queremos hacer?

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = ?
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`?
Razonemos de forma parecida a las anteriores.
Sabemos que `casoAnd` es una función que recibe los resultados de evaluar `p` y `q`.
¿Y con estos resultados qué queremos hacer?
Nada más y nada menos que su conjunción.

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = (&&)
```

Sigamos... ¿Qué queremos hacer cuando nos encontramos con `And p q`?
Razonemos de forma parecida a las anteriores.
Sabemos que `casoAnd` es una función que recibe los resultados de evaluar `p` y `q`.
¿Y con estos resultados qué queremos hacer?
Nada más y nada menos que su conjunción.

(De nuevo, acá podríamos haber puesto `\rp rq -> rp && rq`, pero esto es exactamente lo mismo que simplemente `(&&)`.)

Fold sobre estructuras nuevas

Ejercicio

Definir usando `foldProp` la función

```
evalProp :: (String -> Bool) -> Prop -> Bool
```

de modo que `evalProp asig p` sea el resultado de evaluar la proposición `p` con la asignación `asig` de valores de verdad a variables.

Solución

```
evalProp asig = foldProp casoVar casoNot casoAnd
        where casoVar = asig
              casoNot = not
              casoAnd = (&&)
```

Cuando ya tenemos un poco más de práctica podemos escribirlo directamente como

```
evalProp asig = foldProp asig not (&&)
```

Pero les recomiendo resolverlo con el where en sus primeros ejercicios, para que se entienda mejor qué se hace en cada caso.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Ejercicios

1. Definir el esquema de recursión estructural para el tipo anterior.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Ejercicios

1. Definir el esquema de recursión estructural para el tipo anterior.
2. Usando dicho esquema, definir la función

   ```
   evaluar :: Num a => a -> Polinomio a -> a
   ```

   que evalúa el polinomio cuando la variable x tiene el valor que indicamos.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Hay pistas!

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

1) Es altamente recomendable empezar por el tipo.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: ? -> ? -> ? -> ? -> Polinomio a -> b
```

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: ? -> ? -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: ? -> ? -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> ? -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> ? -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> ? -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> ? -> ? -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

4) ¿Qué tipo tienen los argumentos correspondientes a los constructores recursivos?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

`foldPoli :: b -> (a -> b) -> ? -> ? -> Polinomio a -> b`

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

4) ¿Qué tipo tienen los argumentos correspondientes a los constructores recursivos?
Son similares al de `And`, porque son constructores con argumentos recursivos.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

`foldPoli :: b -> (a -> b) -> (b -> b -> b) -> ? -> Polinomio a -> b`

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

4) ¿Qué tipo tienen los argumentos correspondientes a los constructores recursivos?
Son similares al de `And`, porque son constructores con argumentos recursivos.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
```

2) ¿Qué tipo tiene el argumento correspondiente a `X`?
Es similar al caso base de listas, porque es un constructor sin argumentos.

3) ¿Qué tipo tiene el argumento correspondiente a `Cte`?
Es similar al de `Var`, porque es un constructor con un argumento no recursivo.

4) ¿Qué tipo tienen los argumentos correspondientes a los constructores recursivos?
Son similares al de `And`, porque son constructores con argumentos recursivos.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
    case p of
      X -> casoX
      Cte c -> casoCte c
      Suma p1 p2 -> casoSuma (rec p1) (rec p2)
      Prod p1 p2 -> casoProd (rec p1) (rec p2)
    where rec = foldPoli casoX casoCte casoSuma casoProd
```

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = ?
```

Pensemos como lo hicimos con `Prop`.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = ?
```

Vayamos paso a paso.

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = foldPoli casoX casoCte casoSuma casoProd
```

¿Qué hacemos en el caso del constructor `X`?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd

evaluar v = foldPoli casoX casoCte casoSuma casoProd
```

Queremos evaluar con $X = v$. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = foldPoli v casoCte casoSuma casoProd
```

Queremos evaluar con $X = v$. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = foldPoli v casoCte casoSuma casoProd
```

¿Qué hacemos en el caso del constructor `Cte`?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = foldPoli v casoCte casoSuma casoProd
```

Queremos que valga esa constante. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
```

```
evaluar v = foldPoli v id casoSuma casoProd
```

Queremos que valga esa constante. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id casoSuma casoProd
```

¿Qué hacemos en el caso del constructor Suma?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id casoSuma casoProd
```

Queremos que represente la suma. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id (+) casoProd
```

Queremos que represente la suma. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id (+) casoProd
```

¿Qué hacemos en el caso del constructor `Prod`?

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id (+) casoProd
```

Queremos que represente el producto. Entonces...

Fold sobre estructuras nuevas

Sigamos haciendo folds. Consideremos el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Soluciones

```
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli casoX casoCte casoSuma casoProd p =
     case p of
       X -> casoX
       Cte c -> casoCte c
       Suma p1 p2 -> casoSuma (rec p1) (rec p2)
       Prod p1 p2 -> casoProd (rec p1) (rec p2)
     where rec = foldPoli casoX casoCte casoSuma casoProd
evaluar v = foldPoli v id (+) (*)
```

Notemos que podemos usar `(+)` y `(*)` porque le estamos pidiendo `Num a`.

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con árboles estrictamente binarios:

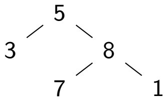
```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con árboles estrictamente binarios:

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

Ejemplo: `miArbol = Nodo 5 (Hoja 3) (Nodo 8 (Hoja 7) (Hoja 1))` representa al árbol

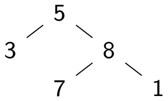


Fold sobre estructuras nuevas

Definimos el siguiente tipo para trabajar con árboles estrictamente binarios:

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

Ejemplo: `miArbol = Nodo 5 (Hoja 3) (Nodo 8 (Hoja 7) (Hoja 1))` representa al árbol



Ya vimos en la teórica que el fold para esta estructura queda así:

```
foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
foldA casoHoja casoNodo (Hoja x) = casoHoja x
foldA casoHoja casoNodo (Nodo x l r) = casoNodo x (rec l) (rec r)
        where rec = foldA casoHoja casoNodo
```

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.
Hay pistas.

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

Nuevamente la idea es ir analizando caso por caso qué queremos hacer.

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

Nuevamente la idea es ir analizando caso por caso qué queremos hacer. En este caso puede ser muy útil dibujar algún árbol en papel y ver cómo se debería comportar cada una de las funciones en ese caso.

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

Nuevamente la idea es ir analizando caso por caso qué queremos hacer.
En este caso puede ser muy útil dibujar algún árbol en papel y ver cómo se debería comportar cada una de las funciones en ese caso.
Una vez que ya sepamos bien qué hace cada función, pasemos a las definiciones.

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

1) `altura`:

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

1) `altura`:
¿Cuál es la altura de una hoja?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

1) `altura`:
¿Cuál es la altura de una hoja?
¿Cuál es la altura de un árbol que tiene dos subárboles de alturas a_ℓ y a_r?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

2) `cantHojas`:

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

2) `cantHojas`:

¿Cuántas hojas tiene una hoja? (Aunque estas preguntas puedan sonar un poco tontas, son las que nos tenemos que hacer para poder resolver cada caso.)

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

2) `cantHojas`:
¿Cuántas hojas tiene una hoja? (Aunque estas preguntas puedan sonar un poco tontas, son las que nos tenemos que hacer para poder resolver cada caso.)
¿Cuántas hojas tiene un árbol que tiene dos subárboles de h_ℓ y h_r hojas?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

3) `cantNodos`:

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

3) `cantNodos`:
¿Cuántos nodos tiene una hoja?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

3) `cantNodos`:
¿Cuántos nodos tiene una hoja?
¿Cuántos nodos tiene un árbol que tiene dos subárboles de n_ℓ y n_r nodos?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

4) `espejo`: el más interesante.

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

4) `espejo`: el más interesante.
¿Cuál es el árbol espejo de una hoja?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

4) `espejo`: el más interesante.
¿Cuál es el árbol espejo de una hoja?
¿Cómo construimos el espejo de un árbol (que no es una hoja) sabiendo los espejos de sus subárboles?

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

```
altura = foldA (const 1) (\_ al ar -> 1 + max al ar)
```

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

```
altura = foldA (const 1) (\_ al ar -> 1 + max al ar)

cantHojas = foldA (const 1) (\_ hl hr -> hl + hr)
```

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

```
altura = foldA (const 1) (\_ al ar -> 1 + max al ar)

cantHojas = foldA (const 1) (\_ hl hr -> hl + hr)

cantNodos = foldA (const 0) (\_ nl nr -> 1 + nl + nr)
```

Fold sobre estructuras nuevas

Ejercicios

Definir las siguientes funciones usando `foldA`:

- `altura :: Arbol a -> Int` (longitud de la rama más larga)
- `cantHojas :: Arbol a -> Int` (cantidad de hojas)
- `cantNodos :: Arbol a -> Int` (cantidad de nodos)
- `espejo :: Arbol a -> Arbol a` (espeja el árbol)

Soluciones

```
altura = foldA (const 1) (\_ al ar -> 1 + max al ar)

cantHojas = foldA (const 1) (\_ hl hr -> hl + hr)

cantNodos = foldA (const 0) (\_ nl nr -> 1 + nl + nr)

espejo = foldA Hoja (\x el er -> Bin x er el)
```

Fold sobre estructuras nuevas

Uno más.

Fold sobre estructuras nuevas

Uno más.

En este caso consideramos árboles que en cada nodo tienen una lista de hijos. Las hojas son los nodos con lista de hijos vacía.

```
data RoseTree a = Rose a [RoseTree a]
```

Fold sobre estructuras nuevas

Uno más.

En este caso consideramos árboles que en cada nodo tienen una lista de hijos. Las hojas son los nodos con lista de hijos vacía.

```
data RoseTree a = Rose a [RoseTree a]
```

Este también lo vimos en la teórica (aunque con otro nombre: `AG`), y el fold quedaba así:

```
foldRT :: (a -> [b] -> b) -> RoseTree a -> b
foldRT f (Rose x hijos) = f x (map (foldRT f) hijos)
```

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Hay pistas!

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si bien la idea sigue siendo hacer lo que veníamos haciendo (esto es, analizando cada caso), ahora tenemos un solo caso! Pero esto no le quita complejidad al asunto...

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si bien la idea sigue siendo hacer lo que veníamos haciendo (esto es, analizando cada caso), ahora tenemos un solo caso! Pero esto no le quita complejidad al asunto...
En cada ejercicio tenemos que pensar qué hace la función `f :: a -> [b] -> b` de `foldRT`.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si bien la idea sigue siendo hacer lo que veníamos haciendo (esto es, analizando cada caso), ahora tenemos un solo caso! Pero esto no le quita complejidad al asunto...
En cada ejercicio tenemos que pensar qué hace la función `f :: a -> [b] -> b` de `foldRT`.
Y ahora esta función puede tener que encargarse de hacer distintas cosas según la lista de hijos sea vacía o no.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Detalle importante: también tenemos que dar los tipos en este ejercicio, pero eso sí se los dejo completamente a ustedes.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Empecemos con `hojas`.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Empecemos con `hojas`.
Tenemos el elemento que guarda cada nodo y también una lista de listas de hojas de los hijos (una lista por cada hijo). ¿Cuáles son las hojas de nuestro árbol?

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Empecemos con `hojas`.
Tenemos el elemento que guarda cada nodo y también una lista de listas de hojas de los hijos (una lista por cada hijo). ¿Cuáles son las hojas de nuestro árbol?
En principio tenemos que separar en dos casos: si estamos en una hoja (acá la lista de hijos sería vacía) o no (en este caso esa lista no sería vacía), porque de eso depende si vamos a devolver o no su elemento.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si es una hoja, tenemos que devolver la lista que sólo contiene el valor almacenado en ella.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si es una hoja, tenemos que devolver la lista que sólo contiene el valor almacenado en ella.
Si no, sólo tenemos que devolver las hojas de los hijos.
¿Cómo se convierte una lista de listas en una sola lista?

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Si es una hoja, tenemos que devolver la lista que sólo contiene el valor almacenado en ella.
Si no, sólo tenemos que devolver las hojas de los hijos.
¿Cómo se convierte una lista de listas en una sola lista?

```
hojas :: RoseTree a -> [a]
hojas = foldRT (\x hs -> if null hs then [x] else concat hs)
```

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Pasemos a `altura`.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Pasemos a `altura`.
De forma similar, acá también tenemos que separar en el caso hoja (altura igual a 1) o no hoja (altura que depende de las alturas de sus hijos, ¿de qué forma depende?).

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Pasemos a `altura`.
De forma similar, acá también tenemos que separar en el caso hoja (altura igual a 1) o no hoja (altura que depende de las alturas de sus hijos, ¿de qué forma depende?).

```
altura :: RoseTree a -> Int
altura = foldRT (\_ hs -> if null hs then 1 else 1 + maximum hs)
```

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Por último, `distancias`.

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Por último, `distancias`.
Me pregunto cuál será la idea...

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Por último, `distancias`.
Me pregunto cuál será la idea...
Sí, es parecido a los demás. Pero hay un detalle a tener en cuenta: ¿cuál es la lista de distancias en el caso de una hoja? Para esto notemos que la distancia de la raíz a una hoja en un árbol que es solamente una hoja, es 0 (porque es el mismo nodo).

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Por último, `distancias`.
Me pregunto cuál será la idea...
Sí, es parecido a los demás. Pero hay un detalle a tener en cuenta: ¿cuál es la lista de distancias en el caso de una hoja? Para esto notemos que la distancia de la raíz a una hoja en un árbol que es solamente una hoja, es 0 (porque es el mismo nodo).

```
distancias :: RoseTree a -> [Int]
distancias=foldRT (\_ hs->if null hs then [0] else map (+1) (concat hs))
```

Fold sobre estructuras nuevas

Ejercicios

Usando `foldRT`, definir las siguientes funciones:

1. `hojas`, que dado un `RoseTree` devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
2. `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.
3. `distancias`, que dado un `RoseTree` devuelve las distancias de su raíz a cada una de sus hojas.

Soluciones

Resumiendo:

```
hojas :: RoseTree a -> [a]
hojas = foldRT (\x hs -> if null hs then [x] else concat hs)

altura :: RoseTree a -> Int
altura = foldRT (\_ hs -> if null hs then 1 else 1 + maximum hs)

distancias :: RoseTree a -> [Int]
distancias=foldRT (\_ hs->if null hs then [0] else map (+1) (concat hs))
```

Ahora cambiemos completamente de tema (o no).

Funciones como estructuras de datos

Ahora cambiemos completamente de tema (o no).

Con el espíritu de seguir estudiando nuevos tipos de datos, en lo que resta de la clase vamos a ver cómo podemos usar funciones como estructuras de datos. En particular, lo vamos a ver ejemplificando con los conjuntos.

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

```
type Conj a = (a->Bool)
```

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

```
type Conj a = (a->Bool)
```

De esta manera, si tenemos un conjunto `s :: Conj a` y un elemento `e::a`, entonces la expresión `s e` da `True` si `e` pertenece a `s`, y `False` en caso contrario.

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

```
type Conj a = (a->Bool)
```

De esta manera, si tenemos un conjunto `s :: Conj a` y un elemento `e::a`, entonces la expresión `s e` da `True` si `e` pertenece a `s`, y `False` en caso contrario.

Por ejemplo:

- `(>3)` es el conjunto $\{4, 5, 6, 7, \ldots\}$

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

```
type Conj a = (a->Bool)
```

De esta manera, si tenemos un conjunto `s :: Conj a` y un elemento `e::a`, entonces la expresión `s e` da `True` si `e` pertenece a `s`, y `False` en caso contrario.

Por ejemplo:

- `(>3)` es el conjunto $\{4, 5, 6, 7, \ldots\}$
- `(=='c')` es el conjunto $\{c\}$

Funciones como estructuras de datos

Hay muchas formas de representar a los conjuntos, y una de ellas es mediante su función de pertenencia:

```
type Conj a = (a->Bool)
```

De esta manera, si tenemos un conjunto `s :: Conj a` y un elemento `e::a`, entonces la expresión `s e` da `True` si `e` pertenece a `s`, y `False` en caso contrario.

Por ejemplo:

- `(>3)` es el conjunto $\{4, 5, 6, 7, \ldots\}$
- `(=='c')` es el conjunto $\{c\}$
- `\x -> x>15 && x<20` es el conjunto $\{16, 17, 18, 19\}$

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.
Hay pistas.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: ?
conjVacio = ?
```

Como siempre, empecemos por su tipo.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: ?
conjVacio = ?
```

Como siempre, empecemos por su tipo.

Un conjunto vacío es un conjunto, de cualquier tipo de elementos. Así que...

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: Conj a
conjVacio = ?
```

Como siempre, empecemos por su tipo.

Un conjunto vacío es un conjunto, de cualquier tipo de elementos. Así que...

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: Conj a
conjVacio = ?
```

Dado que estamos representando a los conjuntos mediante su función de pertenencia, ¿cuál es esta función para el conjunto vacío?

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: Conj a
conjVacio = ?
```

Dado que estamos representando a los conjuntos mediante su función de pertenencia, ¿cuál es esta función para el conjunto vacío?
Tiene que ser una función que dado cualquier elemento de siempre `False`.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
conjVacio :: Conj a
conjVacio = const False
```

Dado que estamos representando a los conjuntos mediante su función de pertenencia, ¿cuál es esta función para el conjunto vacío?
Tiene que ser una función que dado cualquier elemento de siempre `False`.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
union :: ?
union s1 s2 = ?
```

¿Tipo?

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
union :: ?
union s1 s2 = ?
```

¿Tipo?

Claramente es una función que dados dos conjuntos devuelve otro. Con un detallecito: todos tienen que ser del mismo tipo de elementos.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
union :: Conj a -> Conj a -> Conj a
union s1 s2 = ?
```

¿Tipo?

Claramente es una función que dados dos conjuntos devuelve otro. Con un detallecito: todos tienen que ser del mismo tipo de elementos.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
union :: Conj a -> Conj a -> Conj a
union s1 s2 = ?
```

Tenemos que definir su función de pertenencia. Al tratarse de la unión, tenemos que verificar si el objeto que le pasamos como parámetro pertenece a `s1` o a `s2`.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
union :: Conj a -> Conj a -> Conj a
union s1 s2 = \x -> s1 x || s2 x
```

Tenemos que definir su función de pertenencia. Al tratarse de la unión, tenemos que verificar si el objeto que le pasamos como parámetro pertenece a `s1` o a `s2`.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
interseccion :: ?
interseccion s1 s2 = ?
```

Es muy similar a la anterior.

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
interseccion :: Conj a -> Conj a -> Conj a
interseccion s1 s2 = \x -> s1 x && s2 x
```

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
complemento :: ?
complemento s = ?
```

¿Tipo?

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
complemento :: Conj a -> Conj a
complemento s = ?
```

¿Tipo?

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
complemento :: Conj a -> Conj a
complemento s = ?
```

Todos los elementos que pertenecen a `s` no pertenecen a su complemento, y viceversa. Entonces...

Funciones como estructuras de datos

```
type Conj a = (a->Bool)
```

Ejercicios

Definir y dar el tipo de:

- `conjVacio`: el conjunto vacío.
- `union`: dados dos conjuntos, devuelve su unión.
- `interseccion`: dados dos conjuntos, devuelve su intersección.
- `complemento`: dado un conjunto, devuelve su complemento.

Soluciones

```
complemento :: Conj a -> Conj a
complemento s = not . s
```

Todos los elementos que pertenecen a `s` no pertenecen a su complemento, y viceversa. Entonces...

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

SPOILER ALERT! Pensar el ejercicio antes de seguir avanzando.

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.
Hasta acá todo bien. Pensemos la implementación.

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.
Hasta acá todo bien. Pensemos la implementación.

```
map f s = \x -> ??
```

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.
Hasta acá todo bien. Pensemos la implementación.

```
map f s = \x -> ??
```

Queremos que `map f s` nos de un conjunto `s2` tal que `s e = True` si y sólo si `s2 (f e) = True`.

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.
Hasta acá todo bien. Pensemos la implementación.
`map f s = \x -> ??`
Queremos que `map f s` nos de un conjunto `s2` tal que `s e = True` si y sólo si `s2 (f e) = True`.
El problema es que nosotros recibimos "`f e`" y queremos obtener `e` para poder aplicarle `s`, y para esto necesitamos la inversa de `f`, ¡si es que existe!

Funciones como estructuras de datos

Ejercicio

¿Puede definirse un map para esta estructura?
La idea es que, por ejemplo, $map\ (+1)\ \{0, 2, 4\} = \{1, 3, 5\}$.

Solución

Supongamos que se puede.
El tipo debería ser `map :: (a -> b) -> Conj a -> Conj b`.
Hasta acá todo bien. Pensemos la implementación.

```
map f s = \x -> ??
```

Queremos que `map f s` nos de un conjunto `s2` tal que `s e = True` si y sólo si `s2 (f e) = True`.
El problema es que nosotros recibimos "`f e`" y queremos obtener `e` para poder aplicarle `s`, y para esto necesitamos la inversa de `f`, ¡si es que existe!
Por lo tanto, no podemos definir un map para esta estructura.

Funciones como estructuras de datos

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?

Funciones como estructuras de datos

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
2. ¿Y `esVacio :: Conj Bool -> Bool`?

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
2. ¿Y `esVacio :: Conj Bool -> Bool`?

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
2. ¿Y `esVacio :: Conj Bool -> Bool`?

Soluciones

No podemos definir `esVacio :: Conj a -> Bool` para cualquier tipo a porque para esto deberíamos recorrer todo el dominio de la función de pertenencia del conjunto.

Funciones como estructuras de datos

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
2. ¿Y `esVacio :: Conj Bool -> Bool`?

Soluciones

No podemos definir `esVacio :: Conj a -> Bool` para cualquier tipo `a` porque para esto deberíamos recorrer todo el dominio de la función de pertenencia del conjunto.

En cambio, para el caso concreto de `Bool` sí se puede porque es finito.

Funciones como estructuras de datos

Ejercicios

1. ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
2. ¿Y `esVacio :: Conj Bool -> Bool`?

Soluciones

No podemos definir `esVacio :: Conj a -> Bool` para cualquier tipo `a` porque para esto deberíamos recorrer todo el dominio de la función de pertenencia del conjunto.

En cambio, para el caso concreto de `Bool` sí se puede porque es finito. Quedaría así:

```
esVacio :: Conj Bool -> Bool
esVacio s = not (s True || s False)
```

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

SPOILER ALERT! Pensar los ejercicios antes de seguir avanzando.
Hay resolución paso a paso.

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

Definamos una función `enumeracion` que dado un conjunto infinito y computable $S \subseteq \mathbb{N}$ devuelve la función $f : \mathbb{N} \to \mathbb{N}$ que queremos.

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

Definamos una función `enumeracion` que dado un conjunto infinito y computable $S \subseteq \mathbb{N}$ devuelve la función $f : \mathbb{N} \to \mathbb{N}$ que queremos.

¿Qué tipo tiene?

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

Definamos una función `enumeracion` que dado un conjunto infinito y computable $S \subseteq \mathbb{N}$ devuelve la función $f : \mathbb{N} \to \mathbb{N}$ que queremos.

¿Qué tipo tiene?

```
enumeracion :: Conj Int -> Int -> Int
```

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> ?
```

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> ?
```

¿Cómo hacemos para encontrar el n-ésimo elemento (en orden) de S?

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> ?
```

¿Cómo hacemos para encontrar el n-ésimo elemento (en orden) de S?
Podemos construir una lista ordenada de los naturales que pertenecen a S.

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> ?
```

¿Cómo hacemos para encontrar el n-ésimo elemento (en orden) de S?
Podemos construir una lista ordenada de los naturales que pertenecen a S.
Esta lista es `filter s [0..]`.

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> ?
```

¿Cómo hacemos para encontrar el n-ésimo elemento (en orden) de S?
Podemos construir una lista ordenada de los naturales que pertenecen a S.
Esta lista es `filter s [0..]`.
Sólo resta acceder a la posición n.

Funciones como estructuras de datos

Ejercicios

"Si $S \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de S (es decir, existe $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $S = \{f(0), f(1), f(2), \ldots\}$)."

Demostrar esta afirmación programando dicha enumeración.

Soluciones

```
enumeracion :: Conj Int -> Int -> Int
enumeracion s = \n -> (filter s [0..]) !! n
```

¿Cómo hacemos para encontrar el n-ésimo elemento (en orden) de S?
Podemos construir una lista ordenada de los naturales que pertenecen a S.
Esta lista es `filter s [0..]`.
Sólo resta acceder a la posición n.

¿Preguntas?

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

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