Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

3 de abril de 2018

Generación Infinita

Ejercicio: Definir

- `pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).

Generación Infinita

Ejercicio: Definir

- `pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).
- `triplas :: [(Int, Int, Int)]`, una lista (infinita) que contenga todas las triplas de números naturales (sin repetir).

Generación Infinita

Ejercicio: Definir

- `pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).
- `triplas :: [(Int, Int, Int)]`, una lista (infinita) que contenga todas las triplas de números naturales (sin repetir).
- `cuadruplas :: [(Int, Int, Int, Int)]`, (bla bla) cuádruplas (bla bla).

Generación Infinita

Ejercicio: Definir

- `pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).
- `triplas :: [(Int, Int, Int)]`, una lista (infinita) que contenga todas las triplas de números naturales (sin repetir).
- `cuadruplas :: [(Int, Int, Int, Int)]`, (bla bla) cuádruplas (bla bla).

De tarea

```
listasQueSuman :: Int -> [[Int]]
```

que, dado un número natural *n*, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea *n*

```
listasPositivas :: [[Int]]
```

que contenga todas las listas finitas de enteros mayores o iguales que 1.

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

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

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

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

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - unión
  - intersección
  - complemento
  - singleton

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

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

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - unión
  - intersección
  - complemento
  - singleton
- ¿Puede definirse un map para esta estructura?
  Para utilizar, por ejemplo, de esta manera: `(mapC (+1) conj1) e`
- ¿Puede definirse la función `esVacio :: Conj a -> Bool`?

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

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

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - unión
  - intersección
  - complemento
  - singleton
- ¿Puede definirse un map para esta estructura?
  Para utilizar, por ejemplo, de esta manera: `(mapC (+1) conj1) e`
- ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
  ¿Y `esVacio :: Conj Bool -> Bool`?
- *Si $A \subseteq \mathbb{N}$ es infinito y computable, entonces existe una enumeración computable estrictamente creciente de los elementos de A.*[a]
  Demostrar esta afirmación programando la susodicha enumeración.

[a] Existe una $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $A = \{f(0), f(1), f(2), \dots\}$

Volvemos a las listas

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’’]` $\rightsquigarrow$ `[‘‘inutil’’, ‘‘increible’’]`
- `sinVacias :: [[a]] -> [[a]]`, que, dada una lista de listas, devuelve las que no son vacías (en el mismo orden).

Volvemos a las listas

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'']` $\rightsquigarrow$ `[''inutil'', ''increible'']`
- `sinVacias :: [[a]] -> [[a]]`, que, dada una lista de listas, devuelve las que no son vacías (en el mismo orden).

Ahora sí

Definir las siguientes funciones:

- `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.

¿Qué esquema de recursión podemos usar en estos casos?

Esquemas de recursión sobre listas: FoldR

FoldR

```
foldr ::
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

Esquemas de recursión sobre listas: FoldR

FoldR

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

Esquemas de recursión sobre listas: 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.

Esquemas de recursión sobre listas: 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 paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs =
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> 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))
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> 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)
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> 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
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> 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
---> 6
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> 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
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el último elemento de la lista y el caso base del `foldr`. Por esta razón decimos que el `foldr` *acumula* el resultado desde la **derecha**.

Esquemas de recursión sobre listas: FoldR

FoldR

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

Definir utilizando foldr

- `longitud :: [a] -> Int`
- `producto :: [Int] -> Int`
- `concat :: [[a]] -> [a]`
- `all :: (a -> Bool) -> [a] -> Bool`
- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) ->[a] -> [a]`

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> 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)
```

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> 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)
```

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> 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)
```

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> 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
```

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> 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
```

Notar que el primer `(+)` que se puede resolver es entre el primer elemento de la lista y el caso base del `foldl`.

Esquemas de recursión sobre listas: FoldL

FoldL

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

Definir utilizando foldl

- `producto :: [Int] -> Int`
- `reverso :: [a] -> [a]`

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
```

Usando foldl

```
suma [1..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
---> 1 + (2 + (3 + (foldr (+) 0 [4..])))
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
---> foldl (+) (((0 + 1) + 2) + 3) [4..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
```

Usando foldl

```
all even [0..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

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

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

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

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

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

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

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

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
---> False && (foldr (\x r -> even x && r) True [2..])
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
---> False && (foldr (\x r -> even x && r) True [2..])
---> False
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
---> False && (foldr (\x r -> even x && r) True [2..])
---> False
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
-->* foldl (...) (even 2 && (even 1 && (even 0 && True))) [3..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
---> False && (foldr (\x r -> even x && r) True [2..])
---> False
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
-->* foldl (...) (even 2 && (even 1 && (even 0 && True))) [3..]
-->* foldl (...) (even 3 && (even 2 && (...))) [4..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

¿Qué computan estas funciones?

- ```
  f1 :: [Bool] -> Bool
  f1 = foldr (&&) True
  ```
- ```
  f2 :: [a] -> [a]
  f2 = foldr (:) []
  ```
- ```
  f3 :: [a] -> [a] -> [a]
  f3 xs ys = foldr (:) ys xs
  ```
- ```
  f4 :: [a] -> [a]
  f4 = foldl (flip (:)) []
  ```

¡Las difíciles!

Sin usar recursión explícita:

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

¡Las difíciles!

Sin usar recursión explícita:

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

Definir la función take, ¿cuál es la diferencia?

```
take :: Int -> [a] -> [a]
take n = foldr ...
```

Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)              – determina si es o no el caso trivial
-> (a -> b)                – resuelve el caso trivial
-> (a -> [a])              – parte el problema en sub-problemas
-> ([b] -> b)              – combina resultados
-> a                       – input
-> b                       – resultado
```

Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)          – determina si es o no el caso trivial
-> (a -> b)            – resuelve el caso trivial
-> (a -> [a])          – parte el problema en sub-problemas
-> ([b] -> b)          – combina resultados
-> a                   – input
-> b                   – resultado
```

Definir las siguientes funciones

- ```
  dc :: DivideConquer a b
  dc esTrivial resolver repartir combinar x = ...
  ```
- ```
  mergeSort :: Ord a => [a] -> [a]
  mergeSort = dc ...
  ```

Tipos algebraicos y su definición en Haskell

Tipos algebraicos

- definidos como **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*
- se definen mediante la cláusula `data`

Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Tipos algebraicos y su definición en Haskell

Tipos algebraicos

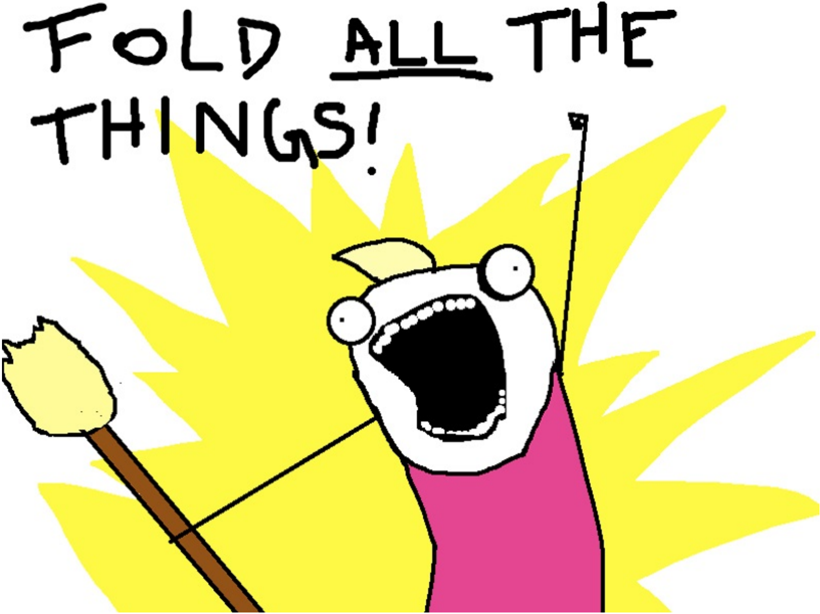
- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

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

Folds sobre estructuras nuevas



¿Cómo hacemos?

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

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

¿Por qué tiene ese tipo?

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

¿Cómo hacemos?

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

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

¿Por qué tiene ese tipo?
(Pista: pensar en cuáles son los constructores del tipo `[a]`).

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.

Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```haskell
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

Ejercicio

Usando el esquema definido, escribir las funciones:

- `proposiciones :: Formula -> [String]`
- `quitarImplicaciones :: Formula -> Formula` que convierte todas las formulas de la pinta $(p \implies q)$ a $(\neg p \vee q)$
- `evaluar :: [(Proposicion, Bool)] -> Formula -> Bool` que dada una formula y los valores de verdad asignados a cada una de sus proposiciones, nos devuelve el resultado de evaluar la fórmula lógica.