

# Conjuntos y combinatoria

Taller de Álgebra I

Primer cuatrimestre 2019

# Repaso de clases anteriores

## Recursión básica

Repasemos algunos problemas que resolvimos definiendo funciones recursivas:

- ▶ Casos que se resuelven aplicando recursivamente la definición sobre el entero anterior

### Factorial

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

# Repaso de clases anteriores

## Recursión básica

Repasemos algunos problemas que resolvimos definiendo funciones recursivas:

- ▶ Casos que se resuelven aplicando recursivamente la definición sobre el entero anterior

### Factorial

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- ▶ Casos cuya definición recursiva no es respecto del entero inmediatamente anterior

### División

```
dividir :: Integer -> Integer -> Integer
dividir a b | a < b      = 0
            | otherwise = 1 + dividir (a-b)
```

# Repaso de clases anteriores

## Recursión sobre parámetros auxiliares

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

### Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de  $n$ .

# Repaso de clases anteriores

## Recursión sobre parámetros auxiliares

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

### Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de  $n$ .

```
sumaDivisores :: Integer -> Integer
sumaDivisores n = sumaDivisoresHasta n n

sumaDivisoresHasta :: Integer -> Integer -> Integer
sumaDivisoresHasta n 0 = 0
sumaDivisoresHasta n m
    | m `mod` n == 0 = m + sumaDivisoresHasta n (m-1)
    | otherwise      = sumaDivisoresHasta n (m-1)
```

# Repaso de clases anteriores

## Recursión múltiple

Casos donde la recursión se debe realizar sobre múltiples parámetros

### Sumas dobles

Definir una función que calcule  $f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i \times j$

# Repaso de clases anteriores

## Recursión múltiple

Casos donde la recursión se debe realizar sobre múltiples parámetros

### Sumas dobles

Definir una función que calcule  $f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i \times j$

```
sumaInterior :: Integer -> Integer -> Integer
sumaInterior m 0 = 0
sumaInterior m n = (m*n) + sumaInterior m (n-1)

sumaDoble :: Integer -> Integer -> Integer
sumaDoble 0 n = 0
sumaDoble m n = sumaInterior m n + sumaDoble (m-1) n
```

# Repaso de clases anteriores

## Listas

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.



# Repaso de clases anteriores

## Listas

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

A diferencia de las tuplas, las listas de distintas longitudes son del mismo tipo. Sin embargo, las tuplas pueden definirse con elementos de distintos tipos.

# Repaso de clases anteriores

## Listas

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

A diferencia de las tuplas, las listas de distintas longitudes son del mismo tipo. Sin embargo, las tuplas pueden definirse con elementos de distintos tipos.

Los patrones fundamentales (*constructores*) de las listas son:

- ▶ `[]`, la lista vacía.
- ▶ `(x:xs)`, una lista con *head* `x` y *tail* `xs`.

# Repaso de clases anteriores

## Recursión sobre listas

Las listas pueden ser tanto parte de la entrada como de la salida de una función:

- Funciones que toman una lista como entrada

### Longitud

```
longitud :: [Integer] -> Integer
longitud []          = 0
longitud (_:xs)      = 1 + longitud xs
```

# Repaso de clases anteriores

## Recursión sobre listas

Las listas pueden ser tanto parte de la entrada como de la salida de una función:

- Funciones que toman una lista como entrada

### Longitud

```
longitud :: [Integer] -> Integer
longitud []      = 0
longitud (_,xs) = 1 + longitud xs
```

- Funciones que devuelven una lista como salida

### Listar Impares

```
imparesHasta :: Integer -> [Integer]
imparesHasta 0 = []
imparesHasta n | n `mod` 2 == 1 = n : imparesHasta (n-1)
               | otherwise      = imparesHasta (n-1)
```

# Repaso de clases anteriores

## Recursión sobre listas

También podemos tener un caso donde tanto la entrada como salida son listas

- Por ejemplo: definir una función que dada una lista  $l$  de enteros, devuelva una lista  $l'$  con los valores de  $l$  que son múltiplo de 7.

### Filtrar listas

```
multiplosDe7 :: [Integer] -> [Integer]
multiplosDe7 []      = []
multiplosDe7 (x:xs) | x `mod` 7 == 0 = x:(multiplosDe7 xs)
                   | otherwise      = multiplosDe7 xs
```

# Repaso de clases anteriores

## Recursión sobre listas

También podemos tener un caso donde tanto la entrada como salida son listas

- ▶ Por ejemplo: definir una función que dada una lista  $l$  de enteros, devuelva una lista  $l'$  con los valores de  $l$  que son múltiplo de 7.

### Filtrar listas

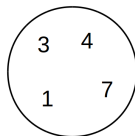
```
multiplosDe7 :: [Integer] -> [Integer]
multiplosDe7 []      = []
multiplosDe7 (x:xs) | x `mod` 7 == 0 = x:(multiplosDe7 xs)
                   | otherwise      = multiplosDe7 xs
```

Todos estos esquemas son importantes!

Repasen los ejercicios que fueron vistos en clase, resuelvan los que no pudieron hacer antes y ¡consulten!

# Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

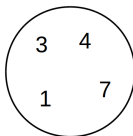


¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.

# Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.



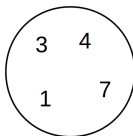
¿Es buena idea usar una lista `[Integer]`?

- ▶ Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - ▶ También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - ▶ Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - ▶ El **orden de los elementos** es relevante para las listas, pero no para conjuntos.



# Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

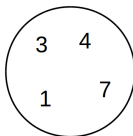


¿Es buena idea usar una lista `[Integer]`?

- ▶ Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - ▶ También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - ▶ Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - ▶ El **orden de los elementos** es relevante para las listas, pero no para conjuntos.
- ▶ ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?

# Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

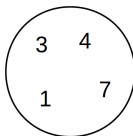


¿Es buena idea usar una lista `[Integer]`?

- ▶ Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - ▶ También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - ▶ Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - ▶ El **orden de los elementos** es relevante para las listas, pero no para conjuntos.
- ▶ ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?
  - ▶ Las listas pueden tener **elementos repetidos**, pero eso no tiene sentido con conjuntos.

# Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.



¿Es buena idea usar una lista `[Integer]`?

- ▶ Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - ▶ También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - ▶ Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - ▶ El **orden de los elementos** es relevante para las listas, pero no para conjuntos.
- ▶ ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?
  - ▶ Las listas pueden tener **elementos repetidos**, pero eso no tiene sentido con conjuntos.

Vamos a usar `[Integer]` para representar conjuntos, pero dejando claro que hablamos de conjuntos (sin orden ni repetidos). Para eso podemos hacer un **renombramiento de tipos**.

## Definición de tipo usando type

Definamos un renombre de tipos para conjuntos: `type Set a = [a]`

- ▶ Otra forma de escribir lo mismo, pero más descriptivo.
- ▶ `type` es la palabra reservada del lenguaje, `Set` es el nombre que le pusimos nosotros.
- ▶ Si bien internamente es una lista, la idea es tratar a `Set a` como si fuera conjunto (es un contrato entre programadores).
- ▶ Si nuestra función recibe un conjunto, **vamos a suponer** que no contiene elementos repetidos. (Haskell no hace nada para verificarlo.)
- ▶ Si nuestra función devuelve un conjunto, **debemos asegurar** que no contiene elementos repetidos. (Haskell tampoco hace nada automático.)
- ▶ Además, no hace falta preocuparse por el orden de los elementos. (Haskell no lo sabe.)

# Conjuntos

## Definición de tipo usando type

Definamos un renombre de tipos para conjuntos: `type Set a = [a]`

- ▶ Otra forma de escribir lo mismo, pero más descriptivo.
- ▶ `type` es la palabra reservada del lenguaje, `Set` es el nombre que le pusimos nosotros.
- ▶ Si bien internamente es una lista, la idea es tratar a `Set a` como si fuera conjunto (es un contrato entre programadores).
- ▶ Si nuestra función recibe un conjunto, **vamos a suponer** que no contiene elementos repetidos. (Haskell no hace nada para verificarlo.)
- ▶ Si nuestra función devuelve un conjunto, **debemos asegurar** que no contiene elementos repetidos. (Haskell tampoco hace nada automático.)
- ▶ Además, no hace falta preocuparse por el orden de los elementos. (Haskell no lo sabe.)

## Ejercicios entre todos

- ▶ Definir `vacío :: Set Integer` que represente el conjunto vacío
- ▶ Implementar entre todos la función  
`agregar :: Integer -> Set Integer -> Set Integer`  
que dado un número  $n$  y un conjunto  $c$  nos devuelva un conjunto (ayuda: La función “pertenece” en Haskell existe y se llama “elem”)

## Ejercicio

- 1 Implementar una función  
`incluido :: Set Integer -> Set Integer -> Bool` que determina si el primer conjunto está incluido en el segundo.
- 2 Implementar una función  
`iguales :: Set Integer -> Set Integer -> Bool` que determina si dos conjuntos son iguales.

```
Ejemplo> iguales [1,2,3,4,5] [2,3,1,4,5]  
True
```

## Más de conjuntos

- Implementar entre todos la función

`agregarATodos :: Integer -> Set (Set Integer) -> Set (Set Integer)` que dado un número  $n$  y un conjunto de conjuntos  $c/s$  agrega a  $n$  en cada conjunto de  $c/s$ .

## Más de conjuntos

- Implementar entre todos la función  
`agregarATodos :: Integer -> Set (Set Integer) -> Set (Set Integer)` que dado un número  $n$  y un conjunto de conjuntos  $c/s$  agrega a  $n$  en cada conjunto de  $c/s$ .

## Ejercicios

- 1 Implementar una función  
`partes :: Integer -> Set (Set Integer)` que genere todos los subconjuntos del conjunto  $\{1, 2, 3, \dots, n\}$ .

```
Ejemplo> partes 2  
[[], [1], [2], [1, 2]]
```



## Producto cartesiano

- Implementar una función

`productoCartesiano :: Set Integer -> Set Integer -> Set (Integer, Integer)`  
que dados dos conjuntos genere todos los pares posibles (como pares de dos elementos) tomando el primer elemento del primer conjunto y el segundo elemento del segundo conjunto.

```
Ejemplo> productoCartesiano [1, 2, 3] [3, 4]  
[(1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)]
```

- ¿Cómo podemos encarar este ejercicio?

## Producto cartesiano

- ▶ Implementar una función

`productoCartesiano :: Set Integer -> Set Integer -> Set (Integer, Integer)`  
que dados dos conjuntos genere todos los pares posibles (como pares de dos elementos) tomando el primer elemento del primer conjunto y el segundo elemento del segundo conjunto.

```
Ejemplo> productoCartesiano [1, 2, 3] [3, 4]  
[(1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)]
```

- ▶ ¿Cómo podemos encarar este ejercicio?
- ▶ Notar que tenemos dos parámetros sobre los que tenemos que hacer recursión para obtener todos los pares.

## Producto cartesiano

- ▶ Implementar una función

`productoCartesiano :: Set Integer -> Set Integer -> Set (Integer, Integer)`  
que dados dos conjuntos genere todos los pares posibles (como pares de dos elementos) tomando el primer elemento del primer conjunto y el segundo elemento del segundo conjunto.

```
Ejemplo> productoCartesiano [1, 2, 3] [3, 4]  
[(1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)]
```

- ▶ ¿Cómo podemos encarar este ejercicio?
- ▶ Notar que tenemos dos parámetros sobre los que tenemos que hacer recursión para obtener todos los pares.
- ▶ Podría servir alguna idea como la de la suma doble...

# Variaciones con repetición

## Variaciones con repetición

- Implementar una función

`variaciones :: Set Integer -> Integer -> Set [Integer]` que dado un conjunto `c` y una longitud `l` genere todas las posibles listas de longitud `l` a partir de elementos de `c`.

```
Ejemplo> variaciones [4, 7] 3
```

```
[[4, 4, 4], [4, 4, 7], [4, 7, 4], [4, 7, 7], [7, 4, 4], [7, 4, 7], [7, 7, 4], [7, 7, 7]]
```

- ¿Cómo podemos pensar este ejercicio recursivamente?

## Insertar un elemento en una lista

- Implementar una función `insertarEn :: [Integer] -> Integer -> Integer -> [Integer]` que dados una lista  $l$ , un número  $n$  y una posición  $i$  (contando desde 1) devuelva una lista en donde se insertó  $n$  en la posición  $i$  de  $l$  y los elementos siguientes corridos en una posición.

```
Ejemplo> insertarEn [1, 2, 3, 4, 5] 6 2  
[1, 6, 2, 3, 4, 5]
```

## Insertar un elemento en una lista

- Implementar una función `insertarEn :: [Integer] -> Integer -> Integer -> [Integer]` que dados una lista `l`, un número `n` y una posición `i` (contando desde 1) devuelva una lista en donde se insertó `n` en la posición `i` de `l` y los elementos siguientes corridos en una posición.

```
Ejemplo> insertarEn [1, 2, 3, 4, 5] 6 2  
[1, 6, 2, 3, 4, 5]
```

## Permutaciones (DIFÍCIL!)

- Implementar una función `permutaciones :: Integer -> [[Integer]]` que genere todas las posibles permutaciones de los números del 1 al `n`.

```
Ejemplo> permutaciones 3  
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

## Más ejercicios

Implementar funciones que devuelvan

- 1 Todas las formas de ubicar  $n$  bolitas numeradas en  $k$  cajas.
- 2 Todas las listas ordenadas de  $k$  números distintos tomados del conjunto  $\{1, \dots, n\}$ .
- 3 Todas las sucesiones de 0 y 1 de longitud 6 en las que hay tres 1's y tres 0's.
- 4 Todas las sucesiones de 0 y 1 de longitud 5 en las que hay mas 1's que 0's.
- 5 Implementar una función  
`subconjuntos :: Integer -> Integer -> Set (Set Integer)` que dados  $k$  y  $n$  enteros, genera todos los subconjuntos de  $k$  elementos del conjunto  $\{1, 2, 3, \dots, n\}$ .

```
Ejemplo> subjconjuntos 2 3  
[[1, 2], [2, 3], [1, 3]]
```

Recordar la demostración combinatoria de la igualdad

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$