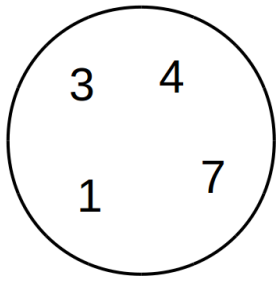
Conjuntos y combinatoria

Taller de Álgebra I

Verano 2020

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 renombre 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.)

Vamos a trabajar con conjuntos de tipos que respeten la igualdad (==). Por ejemplo:

- Set Integer
- Set Float
- Set Bool
- Set (Integer,Integer)
- Set [Integer]
- Set [(Float,Bool)]

Conjunto

Ejercicios entre todos: Implementar las funciones

- `vacio :: Set a` que represente el conjunto vacío
- `pertenece :: Eq a => a -> Set a -> Bool`
  que dado un elemento y un conjunto retorna verdadero si el elemento pertenece al conjunto
- `agregar :: Eq a => a -> Set a -> Set a`
  que dado un elemento y un conjunto agrega el elemento al conjunto

Ejercicios: Implemetar las funciones

1. `union :: Eq a => Set a -> Set a -> Set a` que retorna la unión de los dos conjuntos
2. `interseccion :: Eq a => Set a -> Set a -> Set a` que retorna la intersección de los dos conjuntos
3. `incluido :: Eq a => Set a -> Set a -> Bool` que determina si el primer conjunto está incluido en el segundo
4. `iguales :: Eq a => Set a -> Set a -> Bool` que determina si dos conjuntos son iguales

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

Permutaciones

¿De cuántas maneras puedo ordenar los elementos de A (con $|A| = n$)? De $n!$ maneras.

Pensemos cómo construír estas maneras recursivamente:

- Un conjunto con 1 elemento tiene un ordenamiento posible.
- Dados todos los ordenamientos para un conjunto con $(n-1)$ elementos. ¿Cómo obtengo los de el conjunto que tiene uno más?
- Para cada uno de esos ordenamientos tengo que insertar el nuevo elemento *en cada una de las posiciones posibles*.

Permutaciones

Implementar las siguientes funciones

- `insertarEnPos :: Integer -> Integer -> [Integer] -> [Integer]` que dados un número *n*, una posición *i* y una lista *l*, devuelva una lista en donde se insertó n en la posición i de l y los elementos siguientes corridos en una posición.

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

- `insertarEnTodaPos :: Integer -> [Integer] -> Set [Integer]` que dados un número *n* y una lista *l*, devuelve un conjunto de lista en donde se insertó n en cada posible posición de l.

```
Ejemplo> insertarEnTodaPos 6 [1, 2, 3, 4]
[[6, 1, 2, 3, 4],[1, 6, 2, 3, 4],[1, 2, 6, 3, 4],[1, 2, 3, 6, 4],[1, 2,
3, 4, 6]]
```

- `insertarEnTodaListaEnTodaPos :: Integer -> Set [Integer] -> Set [Integer]` que dados un número *n* y un conjunto de listas, devuelve el conjunto de listas que tiene todas las listas obtenidas de insertar *n* en todas las listas del conjunto en todas las posiciones posibles.
- `permutaciones :: Integer -> Set [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]]
```

Producto Cartesiano

Supongamos que tenemos dos conjuntos A, B (con $|A| = n$,$|B| = m$). Queremos obtener el conjunto

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

En primer lugar, ¿cuántos elementos tiene $A \times B$?
Por cada uno de los n elementos de A, hay un par con cada uno de los m elementos de B, por lo tanto,

$$|A \times B| = n \cdot m$$

¿Pero qué pasa si queremos *listar* todos estos pares? ¿Qué función deberíamos definir?

Producto Cartesiano

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

Ahora consideremos el siguiente problema: ¿De cuántas maneras puedo tomar k elementos de A, considerando el orden y con reposición (es decir, pudiendo sacar varias veces el mismo elemento)?
Pues, para cada una de las k veces podría tomar cualquiera de los n elementos, por lo tanto tenemos

$$n^k$$

posibilidades.
Pero una vez más, nos gustaría poder listarlas.

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?
- Notemos que en este caso, hay una relación entre `variaciones conj n` y `variaciones conj (n-1)`.
- Puede sernos útil pensar una función que dado un conjunto *C* y un conjunto de listas *L*, genere todas las listas producto de agregar *cada elemento de C* a *cada elemento de L*. (Que, de por sí, ¡es una recursión doble!)