Renombres de tipos + Más recursión sobre listas

Taller de Álgebra I

Segundo cuatrimestre de 2016

Renombrar tipos

Hay veces que es útil renombrar tipos, usar un sinónimo. Por ejemplo, el `String` es un sinónimo de `[Char]`. En algún lugar del `Prelude` dice,

```
type String = [Char]
```

Es indistinto usar `[Char]` o `String`.

- A las expresiones que tengan tipo `String` se les puede aplicar cualquier función que reciba el tipo `[Char]`.
- A las expresiones que tengan tipo `[Char]` se les puede aplicar cualquier función que reciba el tipo `String`.

Más ejemplos

Hasta ahora, venimos usando tuplas siempre que necesitamos trabajar con pares de elementos. Los renombres de tipos nos permiten expresar mejor **qué** queremos representar con estas tuplas.
Por ejemplo:

Puntos en el plano

```
type Punto2D = (Float, Float)
```

Números racionales

```
type Racional = (Integer, Integer)
```

Más ejemplos

Ahora podemos definir funciones más expresivas.
Por ejemplo:

```
normaVectorial :: Punto2D -> Float
normaVectorial punto = sqrt ((fst punto)^2 + (snd punto)^2)
```

```
productoRacionales :: Racional -> Racional -> Racional
productoRacionales q1 q2 = (fst q1 * fst q2, snd q1 * snd q2)
```

```
igualdadRacionales :: Racional -> Racional -> Bool
igualdadRacionales q1 q2 = fst q1 * snd q2 == fst q2 * snd q1
```

Problema para hoy

El problema que vamos a resolver es utilizar listas para representar conjuntos de números enteros, y programar las operaciones entre conjuntos: unión, intersección, etc.

Para lograrlo, vamos a:

- definir un tipo `Conjunto` reutiizando el tipo `[Integer]`, asumiendo que:
  a) las listas con las que vamos a trabajar no tienen repetidos, y
  b) no importa el orden en que se encuentran sus elementos.

Con lo que tenemos lo podemos hacer.

Problema para hoy

Vamos a reutilizar las funciones que programamos en la clase anterior:

```
pertenece :: Integer -> [Integer] -> Bool
hayRepetidos :: [Integer] -> Bool
quitar :: Integer -> [Integer] -> [Integer]
eliminarRepetidos :: [Integer] -> [Integer]
maximo :: [Integer] -> Integer
ordenar :: [Integer] -> [Integer]
```

por lo que es importante seguir trabajando **en el mismo archivo** donde están definidas estas funciones.

Definición del tipo `Conjunto`

Definición

- `type Conjunto = [Integer]`

Lo que estamos haciendo es definir el tipo `Conjunto` en base al tipo de datos `[Integer]`. Solo es un **sinónimo**.

Observación: ¡como es un sinónimo todas la funciones definidas para `[Integer]` nos sirven!

Nota: también podríamos definirlo de forma genérica

```
type Conjunto a = [a]
```

Eliminar y agregar elementos

Ejercicios

1. `eliminarElemento :: Integer -> Conjunto -> Conjunto`
   Pueden usar la función de la clase pasada para listas.
2. `agregarElemento :: Integer -> Conjunto -> Conjunto`
   Si el elemento ya está en el conjunto, no tiene que aparecer repetido.

Soluciones

```
eliminarElemento :: Integer -> Conjunto -> Conjunto
eliminarElemento num conj = quitar num conj
```

```
agregarElemento :: Integer -> Conjunto -> Conjunto
agregarElemento num conj
    | pertenece num conj = conj
    | otherwise          = num : conj
```

Algunas operaciones con conjuntos

Ejercicios

1. `union :: Conjunto -> Conjunto -> Conjunto`
2. `interseccion :: Conjunto -> Conjunto -> Conjunto`
3. `inclusion :: Conjunto -> Conjunto -> Bool`
4. `igualdadDeConjunto :: Conjunto -> Conjunto -> Bool`

Algunas operaciones con conjuntos: Soluciones

```
union :: Conjunto -> Conjunto -> Conjunto
union a b
    | length a == 0          = b
    | pertenece (head a) b = union (tail a) b
    | otherwise              = a : union (tail a) b

-- o bien, reutilizando lo que ya hicimos:
union' :: Conjunto -> Conjunto -> Conjunto
union' a b = eliminarRepetidos (a ++ b)
```

```
interseccion :: Conjunto -> Conjunto -> Conjunto
interseccion a b
    | length a == 0          = []
    | pertenece (head a) b = (head a) : (interseccion (tail a) b)
    | otherwise              = interseccion (tail a) b
```

```
inclusion :: Conjunto -> Conjunto -> Bool
inclusion a b
    | length a == 0 = True
    | otherwise = pertenece (head a) b && inclusion (tail a) b
```

```
igualdadConjuntos :: Conjunto -> Conjunto -> Bool
igualdadConjuntos a b = inclusion a b && inclusion b a
```

Ejercicios adicionales

Implementar las siguientes funciones:

1. `diferenciaConjuntos :: Conjunto -> Conjunto -> Conjunto`
   que recibe por parámetro dos conjuntos A y B, y debe devolver el conjunto $A - B$.
2. `diferenciaSimetrica :: Conjunto -> Conjunto -> Conjunto`
   que recibe por parámetro dos conjuntos A y B, y debe devolver el conjunto $A \triangle B$.
3. `interseccionMultiple :: [Conjunto] -> Conjunto`
   que recibe una lista de conjuntos y devuelve la intersección de todos ellos.
4. `unionMultiple :: [Conjunto] -> Conjunto`
   que recibe una lista de conjuntos y devuelve la unión de todos ellos.
5. `mayoresQue :: Integer -> Conjunto -> Conjunto`
   que recibe un conjunto y un número entero, y devuelve los elementos del conjunto que son mayores que el número.
6. `separarParesEImpares :: Conjunto -> (Conjunto, Conjunto)`
   que recibe un conjunto de enteros y lo particiona en dos: uno con sus elementos pares y otro con sus elementos impares.
7. `valoresAbsolutos :: Conjunto -> Conjunto`
   que, dado un conjunto A, devuelve el conjunto $\{|x| : x \in A\}$. (¡Cuidado, pueden aparecer elementos repetidos!).