Práctica extra

Ejercicio 1. Racional

Consideremos el tipo datos

```
data Racional = R Integer Integer
```

para representar números racionales y la instancia `Show` para visualizar los datos de forma correcta

```
instance Show Racional where
    show (R x y) = show x' ++ "/" ++ show y'
        where R x' y' = normaliza (R x y)
```

Define la función

```
normaliza :: Racional -> Racional
```

que dado un racional devuelve un racional irreducible con el signo en el numerador. Por ejemplo `show (R 3 (-6))` devolverá `-1/2`.

Nota la función predefinida `gcd` calcula el máximo común divisor de dos enteros.

Define la función

```
sumaRac :: Racional -> Racional -> Racional
restaRac :: Racional -> Racional -> Racional
```

que suma o resta dos racionales.

Completa las instancias `Eq Racional` y `Ord Racional`

```
instance Eq Racional where
    R x y == R x' y' = undefined
instance Ord Racional where
    R x y <= R x' y' = undefined
```

Ejercicio 2. Dinero

Considera el siguiente tipo para definir cantidades de dinero dadas en euros o dolares.

```
data Dinero = Euros Double | Dolares Double deriving Show
```

y la función que determina el cambio de dólares a euros

```
dolaresPorEuro ::Double
dolaresPorEuro = 1.17
```

Define las funciones

```
aEuros:: Dinero -> Dinero
aDolares:: Dinero -> Dinero
```

que toman una cantidad de dinero y la convierten a la unidad indicada.

Define la función

```
sumaDinero:: Dinero -> Dinero -> Dinero
```

que suma dos cantidades de dinero y devuelve el resultado en euros.

Define la función

`sumaListaDinero ::[Dinero] -> Dinero`

que suma todas las cantidades de dinero que aparecen en una lista y devuelve el resultado en euros.

Ejercicio 3. Complejos

Considera el tipo para representar complejos en forma cartesiana y en forma polar.

```
data Complejo = C Double Double | P Double Double
```

La forma polar contiene el módulo y el argumento.

```
modulo    = sqrt(x*x+y*y)
argumento = atan2 y x
```

siendo `x` e `y` las coordenadas cartesianas.

Sea la instancias de `Show Complejo`

```
instance Show Complejo where
    show (C x y)
        | y == 0 = show x
        | y < 0  = show x ++ show (y) ++ "i"
        | otherwise = show x ++ "+" ++ show y ++ "i"
    show (P m r) = show (aCartesiana (P m r))
```

Define las función

```
aCartesiana:: Complejo -> Complejo
aPolar :: Complejo -> Complejo
```

que transforma un complejo a la forma indicada.

Define la funciones

```
sumaComp :: Complejo -> Complejo -> Complejo
prodComp :: Complejo -> Complejo -> Complejo
```

que suma o multiplica dos complejos.

Ejercicio 4: Listas

Vamos a definir nuestro propio tipo para manejar listas.

```
data List a = Nil | Cons a (List a) deriving Show
```

Aquí aparece un ejemplo de lista usando este tipo

```
ejemplo = Cons 3 (Cons 5 (Cons 1 (Cons 9 Nil)))
```

Define el operador

```
infixr 5 ++>
```

que permite concatenar dos listas.

Define la función

```
mapList:: (a -> b) -> List a -> List b
```

que dada una lista, la transforma en otra aplicando la función dada cada elemento.

Define la función

```
filterList:: (a -> Bool) -> List a -> List a
```

que filtra los elementos de la lista que verifican el predicado.

Define las funciones

```
aHaskell :: List a -> [a]
aList :: [a] -> List a
```

que transforma una lista nuestra en una lista de Haskell y viceversa.

Ejercicio 5. Arbol

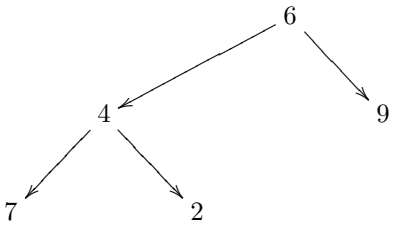
Considera el tipo

```
data Arbol a = Vacio | Nodo (Arbol a) a (Arbol a) deriving Show
```

que define árboles con información en los nodos. Aquí aparece un ejemplo de dato de este tipo

```
ejemploArbol = Nodo (Nodo (Nodo Vacio 7 Vacio) 4 (Nodo Vacio 2 Vacio)) 6 (Nodo Vacio 9 Vacio)
```

que representa al árbol



Define la función

```
nodos:: Arbol a -> Int
prof:: Arbol a -> Int
```

que cuenta el número de nodos y la profundidad de un árbol.

Define la función

```
espejo:: Arbol a -> Arbol a
```

que crea la imagen especular de un espejo (Las ramas izquierdas pasan a ser la derechas en cada nodo).

Define la función

```
mapArbol:: (a -> b) -> Arbol a -> Arbol b
```

que transforma un árbol en otro en el que a cada elemento se le ha apliacando una función.

Ejercicio 6. ArbolH

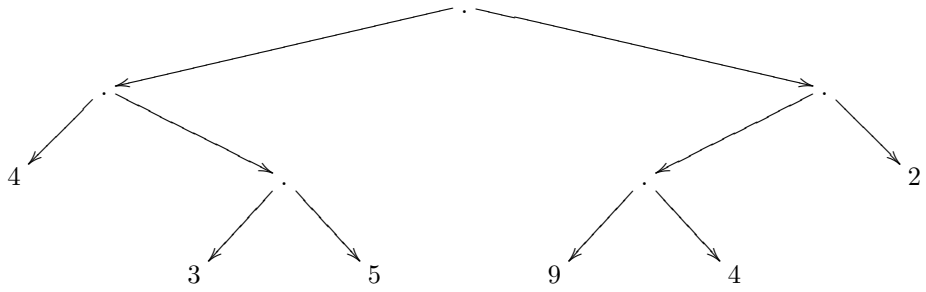
Considera el tipo

```
data ArbolH a = Hoja a | NodoH (ArbolH a) (ArbolH a) deriving Show
```

que define árboles con información en las hojas. Aquí aparece un ejemplo de dato de este tipo

```
ejemploArbolH = NodoH (NodoH (Hoja 4) (NodoH (Hoja 3) (Hoja 5))) (NodoH (NodoH (Hoja 9) (Hoja 4)) (Hoja 2))
```

que representa al árbol



Define la función

```
sumaArbolH :: Num a => Arbol a -> a
```

que suma todos los elementos del árbol.

define la función

```
mapArbolH :: (a -> b) -> ArbolH a -> ArbolH b
```

que transforma un árbol en otro aplicando a cada elemento una función.

Define la función

```
arbolHAList:: ArbolH a -> [a]
```

que transforma un árbol en una lista.