PLP - Práctica 0: Pre-Práctica de Programación Funcional

Zamboni, Gianfranco

17 de febrero de 2018

0.1. Ejercicio 1

`null :: Foldable t => t a -> Bool` indica si una estructura está vacía. El tipo a debe ser de la clase Foldable, esto es, son tipos a los que se les puede aplicar la función foldr. La notación "t aïndica que es un tipo parámetrico, es decir, un tipo t que usa a otro tipo a, por ejemplo, si le pasamos a la función una lista de enteros, entonces a = Int y t = [Int]

`head :: [a] -> a` devuelve el primer elemento de una lista.

`tail :: [a] -> [a]` devuelve los últimos elementos de una lista (todos los elementos, salvo el primero).

`init :: [a] -> [a]` devuelve los primeros elementos de una lista (todos los elementos salvo el último).

`last :: [a] -> a` devuelve el último elemento de una lista.

`take :: Int -> [a] -> [a]` devuelve los primeros n elementos de una lista

`drop :: Int -> [a] -> [a]` devuelve los últimos n elementos de una lista

`(++) :: [a] -> [a] -> [a]` concatena dos listas

`concat :: Foldable t => t [a] -> [a]` concatena todas las listas de un contenedor de listas que soporte la operación foldr.

`(!!) :: [a] -> Int -> a` devuelve el elemento de una lista `l` que se encuentra en la `n`-ésima posición. La numeración comienza desde 0.

`elem :: (Eq a, Foldable t) => a -> t a -> Bool`: Dada una estructura T que soporta la operación foldr y que almacene elementos del tipo a que puedan ser comparados por medio de la igualdad y dado un elemento A de ese tipo, indica si A aparecen en T.

0.2. Ejercicio 2

```haskell
-- a) La función abs de Prelude ya hace esto
valorAbsoluto :: Float -> Float
valorAbsoluto x | x < 0     = -x
                | otherwise =  x

-- b)
bisiesto :: Int -> Bool
bisiesto x = (x `mod` 4) == 0
```

```haskell
--c)
factorial :: Int -> Int
factorial 1 = 1
factorial x = x * factorial (x-1)

cantDivisoresPrimos :: Int -> Int
cantDivisoresPrimos x = length (filter esPrimo (divisores x))

-- Auxiliares

esPrimo :: Int -> Bool
esPrimo x = length (divisores x) == 2

divisores :: Int -> [Int]
divisores x = [ y | y <- [1..x], x `mod` y == 0 ];
```

0.3. Ejercicio 3

```haskell
--a)
inverso :: Float -> Maybe Float
inverso 0 = Nothing
inverso x = Just (1/x)

-- b)
aEntero :: Either Int Bool -> Int
aEntero (Left x) = x
aEntero (Right x) | x == True = 1
                  | otherwise = 0
```

0.4. Ejercicio 4

```haskell
--a)
limpiar :: String -> String -> String
limpiar xs ys = [ y | y <- ys, not(elem y xs) ]

-- b)
difPromedio :: [Float] -> [Float]
difPromedio xs = map (\y -> y - promedio xs) xs
    where promedio xs = (sum xs) / (genericLength xs)

-- c)
todosIguales :: [Int] -> Bool
todosIguales =
        foldr (\y  rec -> ((length xs == 1) || (y == (head xs)))
                && rec) True
```

0.5. Ejercicio 5

```haskell
data AB a = Nil | Bin (AB a) a (AB a)

-- a)
vacioAB:: AB a -> Bool
vacioAB Nil = True
vacioAB (Bin _ _ _) = False
```

```haskell
-- b)
negacionAB :: AB Bool -> AB Bool
negacionAB Nil = Nil
negacionAB (Bin l x r) =
    Bin (negacionAB l) (not x) (negacionAB r)

-- c)
productoAB :: AB Int -> Int
productoAB Nil = 1
productoAB (Bin l x r) = x * (productoAB l) * (productoAB r)
```