LTP - Ejercicios de Haskell resueltos

1. Implementar una función recursiva debe llamarse "primerosSuma", incluyendo la especificación de su tipo. La función debe tener dos parámetros:

- un número "n", entero positivo
- una lista de enteros positivos

Esta función "primerosSuma" debe devolver los primeros valores de la lista hasta que la suma de los mismos sea mayor o igual que "n" (o bien devolver la suma de todos los valores de la lista cuando dicha suma no alcance el valor de "n").

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main>  primerosSuma 4 [2,2,2,2,2,2,2,2]
[2,2]
*Main>  primerosSuma 5 [2,2,2,2,2,2,2,2]
[2,2,2]
*Main>  primerosSuma 5 [20,2,2,2,2,2,2,2]
[20]
```

Solución

Una implementación válida de la función es la siguiente:

```
primerosSuma :: Int -> [Int] -> [Int]
primerosSuma _ [] = []
primerosSuma n (x:xs)
   | n > x     = x : (primerosSuma (n-x) xs)
   | otherwise = [x]
```

2. Definir la función operador "==" para el tipo "Stack a" definido a continuación, de modo que funcione para tipos de "a" que sean de la clase de tipos "Eq".

```
data Stack a       = EmptyStack | Stk a (Stack a) deriving Show
push x s           = Stk x s
top (Stk x s)      = x
pop (Stk _ s)      = s
empty              = EmptyStack
isEmpty EmptyStack = True
isEmpty (Stk x s)  = False
```

Aunque se podría resolver fácilmente utilizando "deriving Eq", se tiene que resolver instanciando la clase de tipos "Eq" (usando "instance") y considerando que dos pilas son iguales únicamente si:

- Las dos pilas están vacías.
- Ninguna está vacía, sus topes son iguales y ambas tienen la misma cantidad de elementos.

En cualquier otro caso son distintas.

Ten en cuenta que para saber si dos pilas tienen la misma cantidad de elementos debes implementar una función auxiliar que cuente los elementos de una pila.

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main> let s0 = empty
*Main> let s1 = push 10 (push 5 empty)
*Main> let s2 = push 10 (push 7 empty)
*Main> let s3 = push 3 (push 10 (push 5 empty))
*Main> let s4 = push 10 (push 7 (push 5 empty))
*Main> s0 == s0
True
*Main> s1 == s2
True
*Main> s3 == s4
False
*Main> s1 == s3
False
```

Solución

Una implementación válida de las funciones es la siguiente:

```
length' :: Stack a -> Int
length' EmptyStack = 0
length' (Stk _ s)  = 1 + length' s

instance (Eq a) => Eq (Stack a) where
   EmptyStack == EmptyStack = True
   (Stk _ _) == EmptyStack  = False
   EmptyStack == (Stk _ _)  = False
   (Stk x s) == (Stk y t)   = (x == y) && (length' s == length' t)
```

3. Implementar una función recursiva, incluyendo la especificación de su tipo, llamada "exists", que reciba dos argumentos:

- un base de datos (un valor de tipo "Database")
- un libro (un valor de tipo "Book")

y devuelva un valor lógico que indique si el libro existe, o no, en la base de datos.

Los tipos "Database", "Book" y "Person" se definen así:

```
type Person = String
type Book = String
type Database = [(Person,Book)]
```

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main> let db = [("Alicia","El nombre de la rosa"), ("Pepe","El
Quijote"), ("Juan","El lobo estepario"), ("Juan","El nombre de la
```

```
rosa"), ("Pepe","Lazarillo de Tormes"), ("Jordi","El nombre de la
rosa"), ("Alicia","El Quijote")]
*Main> exists db "El nombre de la rosa"
True
*Main> exists db "El proceso"
False
```

Solución

Una implementación válida de la función es la siguiente:

```
exists :: Database -> Book -> Bool
exists [] _ = False
exists ((_,b):dbs) book = (b == book) || exists dbs book
```

4. Definir la función operador "==" para el tipo "Stack a" definido a continuación, de modo que funcione para tipos de "a" que sean de la clase de tipos "Eq".

```
data Stack a     = Stk [a] deriving Show
empty            = Stk []
push x (Stk xs)  = Stk (x:xs)
pop (Stk (x:xs)) = Stk xs
top (Stk (x:xs)) = x
isEmpty (Stk xs) = null xs
```

Aunque se podría resolver fácilmente utilizando "deriving Eq", se tiene que resolver instanciando la clase "Eq" (usando "instance") y considerando que dos pilas son iguales únicamente si:

- Las dos pilas están vacías.
- Ninguna está vacía, sus topes son iguales y ambas tienen la misma cantidad de elementos.

En cualquier otro caso son distintas.

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main> let s0 = empty
*Main> let s1 = push 10 (push 5 empty)
*Main> let s2 = push 10 (push 7 empty)
*Main> let s3 = push 3 (push 10 (push 5 empty))
*Main> let s4 = push 10 (push 7 (push 5 empty))
*Main> s0 == s0
True
*Main> s1 == s2
True
*Main> s3 == s4
False
*Main> s1 == s3
False
```

Solución

Una implementación válida de la función es la siguiente:

```
instance (Eq a) => Eq (Stack a) where
   Stk [] == Stk []         = True
   Stk (x:xs) == Stk[]      = False
   Stk [] == Stk (x:xs)     = False
   Stk (x:xs) == Stk (y:ys) = (x == y) && (length xs == length ys)
```

Otra implementación válida de la función es la siguiente:

```
instance (Eq a) => Eq (Stack a) where
   Stk a == Stk b = length a == length b &&
                    (length a == 0 || (head a == head b))
```

5. Dadas las siguientes definiciones de tipos y funciones:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)   deriving Show
type Forest a = [Tree a]

numLeaves :: Tree a -> Int
numLeaves (Leaf x)     = 1
numLeaves (Branch a b) = numLeaves a + numLeaves b
```

Se pide implementar 2 funciones recursivas, incluyendo las especificaciones de sus tipos:

(1) Implementar una función, llamada "countLeaves" que reciba dos argumentos:
- un valor "v" de un tipo genérico "a", que sea de la clase de tipos "Eq".
- un árbol del tipo "Tree a".

Y devuelva:
- la cantidad de hojas en el árbol que contiene el valor "v".

(2) Implementar una función, llamada "forestSize", que reciba:
- un "bosque" (un dato del tipo "Forest a").

Y devuelva:
- el número total de hojas de todos los árboles del bosque.

NOTA: En la implementación de "forestSize" se puede invocar la función "numLeaves", definida antes.

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main> let tree1 = (Branch (Leaf 2) (Leaf 3))
*Main> let tree2 = (Branch (Branch (Leaf 2) (Leaf 4)) (Branch (Leaf
4) (Leaf 5)))
*Main> countLeaves 3 tree1
1
*Main> countLeaves 4 tree2
```

```
2
*Main> let forest = [tree1, tree2, (Leaf 3)]
*Main> forestSize forest
7
```

Solución

A continuación, una implementación válida de las funciones:

```
countLeaves :: (Eq a) => a -> Tree a -> Int
countLeaves z (Leaf x)     = if (z == x) then 1 else 0
countLeaves z (Branch a b) = countLeaves z a + countLeaves z b

forestSize :: Forest a -> Int
forestSize []     = 0
forestSize (t:ts) = numLeaves t + forestSize ts
```

Otra implementación alternativa para la función "forestSize" sería:

```
forestSize :: Forest a -> Int
forestSize = sum . (map numLeaves)
```

6. Implementar una función recursiva, incluyendo la especificación de su tipo.
La función debe llamarse "howMany" y debe recibir dos argumentos:

- una base de datos (un valor de tipo "Database")
- un libro (un valor de tipo "Book")

y debe devolver cuántas veces está el libro en la base de datos.

Los tipos "Database", "Book" y "Person" se definen así:

```
type Person = String
type Book = String
type Database = [(Person,Book)]
```

Considerar los siguientes ejemplos de uso para comprobar la solución:

```
*Main> let db = [("Alicia","El nombre de la rosa"), ("Pepe","El
Quijote"), ("Juan","El lobo estepario"), ("Juan","El nombre de la
rosa"), ("Pepe","Lazarillo de Tormes"), ("Jordi","El nombre de la
rosa"), ("Alicia","El Quijote")]
*Main> howMany db "El nombre de la rosa"
3
*Main> howMany db "El proceso"
0
*Main> howMany db "El Quijote"
2
```

Solución

Una implementación válida de la función es la siguiente:

```
howMany :: Database -> Book -> Int
howMany [] _ = 0
howMany ((_,b):dbs) book
   | b == book = 1 + howMany dbs book
   | otherwise = howMany dbs book
```