Hoja de Ejercicios 2

A continuación se muestran diferentes ejercicios para resolver con el lenguaje Haskell. Algunos ejercicios necesitan algoritmos recursivos para resolverlos (donde se puede utilizar la recursividad final y otros son para practicar con las expresiones lambda.

Ejercicios:

a) Implementa una función en Haskell que elimine de una lista de enteros aquellos números múltiplo de x.

```
> cribar [0,5,8,9,-9,6,0,85,-12,15] 2
[5,9,-9,85,15]
```

Se piden diferentes versiones de la misma función:
- Con definición de listas por comprensión
- Con recursividad no final
- Con recursividad final o de cola

b) Dada la siguiente definición de función

```
doble :: Int -> Int
doble x = x + x
```

¿Cómo cambiaría la definición utilizando expresiones lambda?

c) Se pide una función en Haskell que dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos. Se piden diferentes versiones de la misma función:
- Con recursividad no final
- Con recursividad final o de cola
- Utilizando expresiones lambda u orden superior (se puede hacer uso de la función predefinida de Haskell `map`).

```
> sumaDobles [2,3,4]
18

> sumaDobles [1,2,3]
12
```

d) Implementa una función que sume los cuadrados de los números pares contenidos en una lista de números enteros. Se piden dos versiones:
   a. Una versión que haga uso de las funciones de orden superior de listas `map` y `filter` para definir la nueva función.
   b. Una versión que utilice la definición de listas por comprensión.

e) Dada una lista de enteros, implementar una función para devolver tuplas formadas por los elementos (sin repetir) de la lista, junto con la primera posición en la que aparecen.

```
> primeraAparicion [1,5,6,0,2,6,4,78,9,41,-9,8,-9,12,45,0]
[(1,1),(5,2),(6,3),(0,4),(2,5),(4,7),(78,8),(9,9),(41,10),
                                        (-9,11),(8,12),(12,14),(45,15)]
```

f) Implementar en Haskell una función que calcule el número de secuencias de ceros que hay en una lista de números.

```
> ceros [0]                    > ceros[0,0]
1                              1
> ceros [0,1,0]                > ceros [0,0,1,5,0,4,0,0,0,5]
2                              3
```

g) Implementar una función en Haskell que reciba una lista de números enteros y devuelva dos listas: una con los elementos sin repetir y otra con los elementos que están repetidos.

```
> repeticiones [0,6,0,8,-2,-5,4,-2,6,98,71,2,0,5]
([8,-5,4,98,71,2,5],[0,6,-2])
```

h) Dada una lista de números enteros implementar una función que devuelva una lista con los *n* elementos mayores de la lista original.

```
> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 4
[8,7,6,6]

> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 7
[8,4,6,6,7,0,2]

> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 11
[8,4,-5,6,-1,0,2,6,-10,7]
```

i) Implementa una función incluye en Haskell que reciba dos listas de números enteros y nos diga si la primera de las listas está contenida en la segunda. Se dice que una lista está contenida en otra si los elementos de la primera aparecen dentro de la segunda, en el mismo orden y de forma consecutiva.

```
> incluye [] [4,5]
True

> incluye [4,4,2] [5,4,4,5,4,4,2,9]
True

> incluye [4,4,2] [5,4,4,5,2,9]
False

> incluye [4,5] []
False
```

j) Dada una lista de enteros, se pide implementar una función que ordene dicha lista de menor a mayor utilizando un algoritmo de inserción. Dicho algoritmo de inserción consiste en recorrer la lista L, insertando cada elemento L[i] en el lugar correcto entre los elementos ya ordenados L[1] ,...,L[i-1].

```
> ordenar [2,3,1]
[1,2,3]

> ordenar [1,0,4,0,6,9]
[0,0,1,4,6,9]
```

k) Implementa una función polimórfica en Haskell que reciba 2 listas y vaya cogiendo un elemento de la primera y dos de la segunda, creando una lista final de ternas. En caso de que una de las dos listas se acabe, mostrará la lista de ternas construidas hasta ese momento.

```
> mezclarEnTernas [4,5,8,90] [0,5,6,-9,8,-1,9,52,22]
[(4,0,5),(5,6,-9),(8,8,-1),(90,9,52)]


> mezclarEnTernas [1,2,3] [5,6,7,8]
[(1,5,6),(2,7,8)]

> mezclarEnTernas [1,2,3,4,5] "atropellado"
[(1,'a','t'),(2,'r','o'),(3,'p','e'),(4,'l','l'),(5,'a','d')]

> mezclarEnTernas [True,False] [2.3,5.9,5.7]
[(True,2.3,5.9)]
```

l) Se pide una función polimórfica en Haskell que dado un elemento y una lista añada dicho elemento al final de la lista.

```
> alFinal 3 [1,2,6,7]
[1,2,6,7,3]
```

```
> alFinal True [False,False]
[False,False,True]

> alFinal 'k' "casita"
"casitak"
```

m) Mediante la programación de orden superior se pide implementar una de las funciones predefinidas en la librería estándar de Haskell: la función `zipWith`. Esta función recibe como parámetros una función y dos listas y une ambas listas aplicado la función entre los correspondientes parámetros.

```
> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]

> zipWith' (++) ["hola ", "ciao ", "hi "] ["pepe", "ciao", "peter"]
["hola pepe","ciao ciao","hi peter"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]

> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6]] [[3,2,2],[3,4,5]]
[[3,4,6],[9,20,30]]

> zipWith’ crearTupla [1,2,3] "casita"
[(1,'c'),(2,'a'),(3,'s')]
```

(Suponiendo que la función crearTupla tiene la siguiente definición:

```
crearTupla :: a-> b-> (a,b)
crearTupla x y = (x,y)
```

)

n) Define una función polimórfica que sea capaz de invertir los elementos de una lista. Se piden diferentes versiones:
   - Con recursividad no final
   - Con recursividad de cola o final
   - Utilizando la función de orden superior foldr

```
> reverse' [1,2,3]
[3,2,1]

> reverse' "casa"
"asac"
```

o) Define una función polimórfica que sea capaz de invertir los elementos de una lista de listas.

```
> reverse'' [[1,2,3],[3,4,5]]
[[5,4,3],[3,2,1]]
```

```
> reverse'' ["pepe", "casa", "patio"]
["oitap","asac","epep"]
```

p) Implementar la función predefinida de la librería estándar `flip`. Esta función lo que hace es recibir una función y devolver otra función que es idéntica a la función original, salvo que intercambia los dos primeros parámetros.

```
> flip' zip [1,2,3] "casa"
[('c',1),('a',2),('s',3)]

> flip' (+) 3 4
7

> flip' (++) "casa" "pollo"
"pollocasa"
```

q) Implementar la función polimórfica predefinida de la librería estándar `map`. Esta función lo que hace es recibir una función y una lista y devuelve la lista resultante de aplicar la función a cada elemento de la lista original.

```
> map (3*) [1,2,3]
[3,6,9]

> map doble [1,2,3]
[2,4,6]

> map not [True,False]
[False,True]
```