EJERCICIOS HASKELL

Ejercicio 3.3.1. *El doble factorial de un número* `n` *se define por*

```
0!! = 1
1!! = 1
n!! = n*(n-2)* ... * 3 * 1, si n es impar
n!! = n*(n-2)* ... * 4 * 2, si n es par
```

Por ejemplo,

```
8!! = 8*6*4*2   = 384
9!! = 9*7*5*3*1 = 945
```

Definir, por recursión, la función

```
dobleFactorial :: Integer -> Integer
```

tal que `(dobleFactorial n)` *es el doble factorial de* `n`*. Por ejemplo,*

```
dobleFactorial 8  ==  384
dobleFactorial 9  ==  945
```

Solución:

```
dobleFactorial :: Integer -> Integer
dobleFactorial 0 = 1
dobleFactorial 1 = 1
dobleFactorial n = n * dobleFactorial (n-2)
```

Ejercicio 3.1.1. *Definir por recursión la función*

```
potencia :: Integer -> Integer -> Integer
```

tal que `(potencia x n)` *es* `x` *elevado al número natural* `n`*. Por ejemplo,*

```
potencia 2 3 == 8
```

Solución:

```
potencia :: Integer -> Integer -> Integer
potencia m 0 = 1
potencia m n = m*(potencia m (n-1))
```

Ejercicio 3.5.1. *Definir por recursión la función*

```
menorDivisible :: Integer -> Integer -> Integer
```

tal que `(menorDivisible a b)` *es el menor número divisible por los números del* `a` *al* `b`. *Por ejemplo,*

```
menorDivisible 2 5 == 60
```

Indicación: *Usar la función* `lcm` *tal que* `(lcm x y)` *es el mínimo común múltiplo de* `x` *e* `y`.

Solución:

```
menorDivisible :: Integer -> Integer -> Integer
menorDivisible  a b
    | a == b    = a
    | otherwise = lcm a (menorDivisible (a+1) b)
```

Ejercicio 3.14.7. *Definir por recursión la función*

```
esPermutacion :: Eq a => [a] -> [a] -> Bool
```

tal que `(esPermutacion xs ys)` *se verifica si* `xs` *es una permutación de* `ys`. *Por ejemplo,*

```
esPermutacion [1,2,1] [2,1,1]  ==  True
esPermutacion [1,2,1] [1,2,2]  ==  False
```

Solución:

```
esPermutacion :: Eq a => [a] -> [a] -> Bool
esPermutacion []     []     = True
esPermutacion []     (y:ys) = False
esPermutacion (x:xs) ys     = elem x ys && esPermutacion xs (borra x ys)
```

Ejercicio 4.1.3. *Definir, por comprensión, la función*

```
sumaCuadradosC :: Integer -> Integer
```

tal que `(sumaCuadradosC n)` *es la suma de los cuadrados de los números de 1 a* `n`. *Por ejemplo,*

```
sumaCuadradosC 4 == 30
```

Solución:

```
sumaCuadradosC :: Integer -> Integer
sumaCuadradosC n = sum [x^2 | x <- [1..n]]
```

Ejercicio 4.2.1. *Se quiere formar una escalera con bloques cuadrados, de forma que tenga un número determinado de escalones. Por ejemplo, una escalera con tres escalones tendría la siguiente forma:*

```
  XX
 XXXX
XXXXXX
```

Definir, por recursión, la función

```
numeroBloquesR :: Integer -> Integer
```

tal que `(numeroBloquesR n)` *es el número de bloques necesarios para construir una escalera con* `n` *escalones. Por ejemplo,*

```
numeroBloquesR 1   == 2
numeroBloquesR 3   == 12
numeroBloquesR 10  == 110
```

Solución:

```
numeroBloquesR :: Integer -> Integer
numeroBloquesR 0 = 0
numeroBloquesR n = 2*n + numeroBloquesR (n-1)
```

Ejercicio 4.2.2. *Definir, por comprensión, la función*

```
numeroBloquesC :: Integer -> Integer
```

tal que `(numeroBloquesC n)` *es el número de bloques necesarios para construir una escalera con* `n` *escalones. Por ejemplo,*

```
numeroBloquesC 1   == 2
numeroBloquesC 3   == 12
numeroBloquesC 10  == 110
```

Solución:

```
numeroBloquesC :: Integer -> Integer
numeroBloquesC n = sum [2*x | x <- [1..n]]
```

Ejercicio 4.4.9. *Definir, por recursión, la función*

```
listaNumeroR :: [Integer] -> Integer
```

tal que `(listaNumeroR xs)` *es el número formado por los dígitos de la lista* `xs`*. Por ejemplo,*

```
listaNumeroR [5]        == 5
listaNumeroR [1,3,4,7]  == 1347
listaNumeroR [0,0,1]    == 1
```

Solución:

```
listaNumeroR :: [Integer] -> Integer
listaNumeroR xs = listaNumeroR' (reverse xs)

listaNumeroR' :: [Integer] -> Integer
listaNumeroR' [x]    = x
listaNumeroR' (x:xs) = x + 10 * (listaNumeroR' xs)
```

Ejercicio 4.4.10. *Definir, por comprensión, la función*

```
listaNumeroC :: [Integer] -> Integer
```

tal que `(listaNumeroC xs)` *es el número formado por los dígitos de la lista* `xs`*. Por ejemplo,*

```
listaNumeroC [5]        == 5
listaNumeroC [1,3,4,7]  == 1347
listaNumeroC [0,0,1]    == 1
```

Solución:

```
listaNumeroC :: [Integer] -> Integer
listaNumeroC xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]
```