Trabajo Práctico 4 **Prog. Avanzada**

Ejercicio 1

Generar una lista infinita de unos.

Respuesta

```haskell
list1 :: [Int]
list1 = 1 : list1
```

Ejercicio 2

Generar una lista infinita de naturales comenzando desde un número dado.

Respuesta

```haskell
list2 :: Int -> [Int]
list2 n = n : list2 (n + 1)
```

Ejercicio 3

Generar una lista con los primeros **n** naturales.

Respuesta

```haskell
primerosN :: Int -> [Int]
primerosN n = take n (list2 0)
```

Ejercicio 4

Retornar los primeros 5 elementos de una lista infinita de enteros positivos.

Respuesta

```haskell
primeros5 :: [Int]
primeros5 = take 5 (list2 0)
```

Ejercicio 5

Dada una lista de enteros, retornar sus cuadrados, es decir, dado [x0, x1, …, xn] deberia retornar [(x0)^2, (x1)^2, …, (xn)^2]

Respuesta

```haskell
cuadrados :: [Int] -> [Int]
cuadrados = map (^2)
```

Ejercicio 6

Dado un entero positivo, retornar la lista de sus divisores.

Respuesta

-metodo función lambda

```
divisoresPrueba :: Int -> [Int]
divisoresPrueba n = filter (\x -> n `mod` x == 0) [1..n]
```

-metodo función auxiliar

```
esDivisor :: Int -> Int -> Bool
esDivisor divisor numero = (divisor `mod` numero) == 0  --

divisores :: Int -> [Int]
divisores n = filter (esDivisor n) [1..n]
```

-metodo where

```
divisoress :: Int -> [Int]
divisoress n = filter p [1..n] where p x = mod n x == 0
```

Ejercicio 7

Dada una lista de naturales , obtener la lista que contenga solo los números primos de la lista original.

Respuesta

```
numPrim :: Int -> Bool
numPrim n = (n > 1) && null [x | x <- [2..n - 1], n `mod` x == 0]

listaprimos :: [Int] -> [Int]
listaprimos = filter numPrim
```

Ejercicio 8

Dada una lista de naturales, retornar la suma de los cuadrados de la lista.

Respuesta

```
listaNaturales :: [Int] -> Int
listaNaturales xs= sum  (map (^2) xs )
```

Ejercicio 9

Dada una lista de naturales, retornar la lista con sus sucesores

Respuesta

```haskell
listaNaturales2 :: [Int] -> [Int]
listaNaturales2 xs = map (succ) xs
```

Ejercicio 10

Dada una lista de enteros, sumar todos sus elementos

Respuesta

```haskell
sumEnteros :: [Int] -> Int
sumEnteros = foldl (+) 0
```

Ejercicio 11

Definir el factorial usando **fold**

Respuesta

```haskell
factorial :: Int -> Int
factorial n = foldr (*) 1 [1..n]
```

Ejercicio 12

Redefinir la función **and** tal que **and xs** tal que **and xs** se verifica si todos los elementos de xs son verdaderos, Por ejemplo: **and [1<2. 2<3, 1/=0] = True, and [1<2, 2<3, 1==0] = False**.

Respuesta

```haskell
ands :: [Bool] -> Bool
ands = foldr (&&) True
```

Ejercicio 13

Usando **foldl** o **foldr** definir una funcion **tam :: [a] -> Int** que devuelve la cantidad de elementos de una lista dada. Dar un ejemplo en los cuales **foldr** y **foldl** evaluén diferente con los mismos parámetros.

Respuesta

```haskell
tam :: [a] -> Int
tam xs = foldl (\acumulador i -> acumulador + 1) 0 xs

    --otra manera
incrementar :: Int -> a -> Int
incrementar acumulador _ = acumulador + 1

tam1 :: [a] -> Int
tam1 xs = foldl incrementar 0 xs
```

Utilizando listas por comprensión resolver

Ejercicio 14

Dada una lista de enteros, retornar sus sucesores.

Respuesta

```
listaSuccesores :: [Int] -> [Int]
listaSuccesores xs = [x + 1| x <- xs]
```

Ejercicio 15

Dada una lista de naturales, retornar sus cuadrados.

Respuesta

```
listaCuadrados :: [Int] -> [Int]
listaCuadrados xs = [x^2 | x <-xs]
```

Ejercicio 16

Dada una lista de enteros, retornar los elementos pares que sean mayores a 10.

Respuesta

```
listaPares :: [Int] -> [Int]
listaPares xs = [x | x <- xs, x > 10 && even x]
```

Ejercicio 17

Dado un entero, retornar sus divisores.

Respuesta

```
listDiv :: Int -> [Int]
listDiv num = [x | x <- [1..num], (num `mod` x) == 0  ]
```

Ejercicio 18

Definir la función **todoOcurrenEn :: Eq a => [a] -> [a] -> Bool** tal que **todosOcurrenEn xs ys** se verifica si todos los elementos de xs son elementos de ys. Por ejemplo: **todosOcurrenEn [1,5,2,5] [5,1,2,4] = True, todosOcurrenEn [1,5,2,5] [5,2,4] = False**.

Respuesta

```
todosOcurrenEn :: (Eq a) => [a] -> [a] -> Bool
todosOcurrenEn xs ys = null [x | x<- xs, x `notElem` ys]
```

Ejercicio 19

Dado un natural **n**, retornar los números primos comprendidos entre 2 y **n.**

Respuesta

```haskell
esPrimo :: Int -> Bool
esPrimo n | n < 2 = False
          | otherwise = all (\x -> n `mod` x /= 0) [2..intSqrt n]
          where intSqrt = floor . sqrt . fromIntegral
--intSqrt = toma un numero entero y devuelve la parte entera de su
raiz cuadrada

    --otra manera
numPrimitos :: Int -> [Int]
numPrimitos n = [x | x <- [2..n], esPrimo x ]
```

Ejercicio 20

Dadas dos listas de naturales, retornar su producto cartesiano.

Respuesta

```haskell
cartesiano :: [Int] -> [Int] -> [(Int,Int)]
cartesiano xs ys = [(x,y)  | x <- xs, y <- ys]
```

Ejercicio 21

Dadas una lista y un elemento retornar el número de ocurrencias del elemento x en la lista ys.

Respuesta

```haskell
numOcurrencia :: (Eq a) => [a] -> a -> Int
numOcurrencia ys x = length [ y | y <- ys, y == x ]
```

Ejercicio 22

Escribir la funcion **split2 :: [a] -> [([a],[a])],** que dada una lista xs, devuelve la lista con todas las formas de partir **xs** en dos. Por ejemplo: **slipt2 [1,2,3] = [([ ], [1,2,3]), ([1], [2,3]), ([1,2], [3]), ([1,2,3], [ ])].**

Respuesta

```haskell
    --modo 1
split2 :: [a] -> [([a], [a])]
```

```haskell
split2 xs = [ (x,y) |n <- [0 ..length(xs)],  x <- [take n (xs)], y
<- [drop n (xs)]]

    --modo 2
Split3 :: [a] -> [([a], [a])]
split3 xs = [ (take i xs, drop i xs) | i <- [0..length xs] ]

 --funcionamiento:
    --dado [1,2,3]
    --xs = [1,2,3]
    --n <- [0..length(xs)] = n <- [0..length([1,2,3])] = n <-
[0..3] {itera con n = 0}

    -- x <- take 0 ([1,2,3]), y <- drop 0 ([1,2,3])
    --      x <- []         ,    y <- [1,2,3]

    -- (x,y) = ([], [1,2,3])

    --y luego con n = 1, n = 2, y n = 3

    -- total: [([],[1,2,3]), ([1],[2,3]), ([1,2],[3]), ([1,2,3],
[])]
    --              n = 0  ,   n = 1    ,   n = 2    ,   n = 3
```

Ejercicio 23

Definir una función que, dada una lista de enteros devuelva la suma de la suma de todos los segmentos iniciales.
Por ejemplo: **sumaSeg [1,2,3] = 0 + 1 + 3 + 6 = 10**

Respuesta

```haskell
sumaSeg :: [Int] -> Int
sumaSeg xs = sum [sum (take n xs) | n <- [1..length xs]]
```

Ejercicio 24

Definir la lista infinita de los números pares.

Respuesta

```haskell
pares :: [Int]
pares = [2,4..]
```