INSTITUTO TECNOLÓGICO SUPERIOR DEL SUR DE GUANAJUATO

Ejercicios en haskell

Materia:
Programación Lógica y Funcional

Elaborado por:
Luis Manuel Cárdenas Ibarra

Docente:
Gustavo Ivan Vega

Uriangato, Gto. 28 de Febrero del 2025

En el ejercicio 1 de funciones de haskell, nos pide realizar 3 funciones, la primera de ellas es para calcular el descuento de un producto, para ello, primero definí la función como totalDescuento, para hacer referencia a que daría el total del producto con el descuento pedido, esta función recibe 2 valores de tipo Float y devuelve otro valor de tipo Float, definimos los primeros 2 valores Float como precio y descuento, con ellos se aplica una regla de 3 para obtener el total con el descuento solicitado y se aplica al tercer valor Float de la función.

```
module Compras where
    -- Función para aplicar descuento
    totalDescuento :: Float -> Float -> Float
    totalDescuento precio descuento = (100-descuento)*precio/100
```

La función para calcular el iba, es bastante similar a descuento, la diferencia es que el iva es fijo por lo que solo recibe 2 valores, uno de entrada, el cual es el precio, y uno de salida, para el total con el IVA.

```
-- Funcion para aplicar IVA
totalIva :: Float -> Float
totalIva precio = precio+(16*precio/100)
```

Para el caso de la función total, vemos que recibe un arreglo de duplas (“[(Float,Float)]”) el cual serán los valores de entrada, es decir, precio y el porcentaje de descuento, después solicita 3 valores Float, los cuales se utilizarán para hacer referencia a las funciones totalDescuento y totalIva. Sin embargo, totalIva solo permite 2 valores, no 3, como totalDescuento, pero esta parte la arreglaremos al momento de llamar la función total en la función main. Despues de tener los valores de precio, porcentaje y la función a realizar, está otro valor Float el cual guardará el resultado de la función, que será la suma de las tuplas en el arreglo con el descuento o con el IVA.

```
total :: [(Float,Float)] -> (Float -> Float -> Float) -> Float
total cesta funcion = sum [funcion precio porcentaje | (precio, porcentaje) <- cesta]
```

En el main, declaré el arreglo de tuplas como cesta, donde ingresé 3 tuplas, las cuales simulas ser 3 productos distintos, con distintos descuentos, después se imprimen 2 lineas, la primera imprime el precio total de la cesta con el respectivo descuento de cada producto y la segunda con el iva para cada producto, en la linea de descuento no hay mucho que decir, se llama la función total y se le envía los valores de la cesta y la función que se quiere realizar (totalDescuento), sin embargo la linea para imprimir el IVA es diferente, ya que esta solo tiene 1 valor de entrada y otro de salida, por lo que, para que solo tomara el primer valor de las tuplas se pone (\precio _ → totalIva precio), de esta manera solucionamos que solo tome el valor necesario para calcular el total con el IVA.

```
main :: IO ()
main = do
    let cesta = [(100, 10), (200, 5), (50, 20)] -- Lista de (precio, porcentaje)
    putStrLn $ "Precio con descuento: " ++ show (total cesta totalDescuento)
    putStrLn $ "Precio con IVA: " ++ show (total cesta (\precio _ -> totalIva precio))
```

El la siguiente imagen se muestra el resultado del primer ejercicio:

```
ghci> :l Ej1.hs
[1 of 1] Compiling Compras          ( Ej1.hs, interpreted )
Ok, one module loaded.
ghci> main
Precio con descuento: 320.0
Precio con IVA: 406.0
```

Para el segundo ejercicio, pide realizar una función que reciba otra función y una lista para devolver otra lista con los resultados de aplicar otra función a cada valor de la lista, utilizamos la función map de haskell para recorrer todos los valores de lista y aplicarles la función.

```haskell
module DobleFuncion where
    -- Función de orden superior
    funcionSuperior :: (a -> b) -> [a] -> [b]
    funcionSuperior f lista = map f lista
```

La función cubo simplemente recibe un entero y devuelve otro, con el valor ingresado, se multiplica 3 veces así mismo y listo, en la función main, se declara la lista de números y se imprimen 2 lineas, una muestra la lista de prueba y la segunda la lista después aplicar la función cubo a cada elemento de la lista.

```haskell
-- Función para elevar al cubo un número
cubo :: Int -> Int
cubo x = x * x * x

main :: IO ()
main = do
    let numeros = [1, 2, 3, 4, 5]
    putStrLn $ "Lista original: " ++ show numeros
    putStrLn $ "Lista duplicada: " ++ show (funcionSuperior cubo numeros)
```

Este es el resultado al ejecutar el código:

```
ghci> :l Ej2.hs
[1 of 1] Compiling DobleFuncion     ( Ej2.hs, interpreted )
Ok, one module loaded.
ghci> main
Lista original: [1,2,3,4,5]
Lista duplicada: [1,8,27,64,125]
```

El import nos sirve para poder recorrer cada carácter de cada frase de una oración con la función fromListWith, de otra manera, para contar la longitud de cada frase, se deberá hacer una función recursiva, sin embargo al utilizar map, es más sencillo este proceso, fromListWith, vemos que parte la oración (words) en palabras y las manda a “palabra”, después de tener la palabra, usamos length para saber la longitud de cada palabra y las almacena en Map String Int el cual es una lista de un String con un entero para imprimir la palabra y la cantidad de caracteres.

```haskell
import Data.Map (Map, fromListWith)

-- Función que recibe una frase y devuelve un diccionario (Map) con las palabras y su longitud
contarLongitudes :: String -> Map String Int
contarLongitudes frase = fromListWith (+) [(palabra, length palabra) | palabra <- words frase]

-- Ejemplo de uso
main :: IO ()
main = do
    let frase = "Inserta una frase aqui"
    print $ contarLongitudes frase
```

La salida se muestra de la siguiente manera, aunque en desorden, las palabras y la longitud de cada una de ellas es correcta.

```
ghci> :l Ej3.hs
[1 of 2] Compiling Main             ( Ej3.hs, interpreted )
Ok, one module loaded.
ghci> main
fromList [("Inserta",7),("aqui",4),("frase",5),("una",3)]
```

En el cuarto ejercicio importamos toUpper para pasar las asignaturas a mayúsculas, y en la función asignaturas, recibe 1 lista de tuplas y devuelve otra, map se utiliza para transformar cada elemento de la lista, (asignatura, nota) representa cada asignatura de la lista y map toUpper asignatura, es para pasar a mayúsculas cada signatura, por ultimo, se llama la función calificación y se le manda la nota, con esto obtendremos una lista de duplas con las asignatura en mayúsculas y el rendimiento en cada asignatura.

```haskell
import Data.Char (toUpper)  -- Importamos la función toUpper

-- Función que recibe un diccionario con asignaturas y notas y devuelve otro diccionario
--con las asignaturas en mayúsculas y las calificaciones correspondientes
asignaturas :: [(String, Int)] -> [(String, String)]
asignaturas = map (\(asignatura, nota) -> (map toUpper asignatura, calificacion nota))
```

La función calificación recibe un entero que será la nota de la asignatura y devuelve un string dependiendo de la nota que se haya obtenido

```haskell
-- Función que determina la calificación según la nota
calificacion :: Int -> String
calificacion nota
  | nota >= 95 = "Excelente"
  | nota >= 85 = "Notable"
  | nota >= 75 = "Bueno"
  | nota >= 70 = "Suficiente"
  | otherwise  = "Desempenio insuficiente"
```

En la función main se define el diccionario de notas, que es la lista de materias con las calificaciones de cada materia, y se ingresa en la función de asignaturas.

```haskell
-- Ejemplo de uso
main :: IO ()
main = do
    let diccionarioNotas = [("matematica", 92), ("historia", 68), ("fisica", 76), ("biologia", 83)]
    print $ asignaturas diccionarioNotas
```

Este código imprime lo siguiente:

```
ghci> :l Ej4.hs
[1 of 2] Compiling Main             ( Ej4.hs, interpreted )
Ok, one module loaded.
ghci> main
[("MATEMATICA","Notable"),("HISTORIA","Desempenio insuficiente"),("FISICA","Bueno"),("BIOLOGIA","Bueno")]
```

Para el ejercicio 5, nos pide calcular el módulo de un vector, esto se calcula obteniendo la raiz cuadrada de la suma de los cuadrados de cada elemento en la lista, por lo que al inicio, map recorre la lista y eleva al cuadrado cada elemento, con sum, suma todos los elementos y obtiene la raiz cuadrada, el main simplemente define la lista de valores del vector y la ingresa a la función moduloVector.

```haskell
module ModuloVector where
-- Función que calcula el módulo de un vector
moduloVector :: [Double] -> Double
moduloVector vector = sqrt (sum (map (^2) vector))

-- Ejemplo de uso
main :: IO ()
main = do
    let vector = [5.0, 8.0]  -- Un vector en 2 dimensiones
    print $ moduloVector vector  -- Imprime el módulo del vector
```

Nos devuelve lo siguiente:

```
ghci> :l Ej5.hs
[1 of 1] Compiling ModuloVector     ( Ej5.hs, interpreted )
Ok, one module loaded.
ghci> main
9.433981132056603
```