Enunciado

Una reconocida empresa de comercio electrónico nos pide desarrollar un sistema de stock de mercadería. El conjunto de mercaderías puede representarse con una secuencia de nombres de los productos donde puede haber productos repetidos. El stock puede representarse como una secuencia de tuplas de dos elementos, donde el primero es el nombre del producto y el segundo es la cantidad que hay en stock (en este caso no hay nombre de productos repetidos). También se cuenta con una lista de precios de productos representada como una secuencia de tuplas de dos elementos, donde el primero es el nombre del producto y el segundo es el precio.

Para implementar este sistema nos enviaron las siguientes especificaciones y nos pidieron que hagamos el desarrollo enteramente en Haskell, utilizando los tipos requeridos y solamente las funciones que se ven en la materia Introducción a la Programación / Algoritmos y Estructuras de Datos I (FCEyN-UBA).

Ejercicio 1 (2 puntos)

problema generarStock (productos: seq⟨seq⟨Char⟩⟩): seq⟨seq⟨Char⟩ x Z⟩ {
  requiere: {True}
  asegura: {La longitud de res es igual a la cantidad de productos distintos que hay en productos}
  asegura: {Para cada producto que pertenece a productos existe un i tal que 0 <= i < |res| y
  res[i]0=producto y res[i]1 es igual a la cantidad de veces que aparece producto en productos}
}

Ejercicio 2 (2 puntos)

problema stockDeProducto (stock:seq⟨seq⟨Char⟩ x Z⟩, producto: seq⟨Char⟩) : Z {
  requiere: {No hay productos repetidos en stock}
  requiere: {Todas las cantidades de los productos que hay en stock son mayores a cero}
  asegura: {(res = 0 y producto no se encuentra en el stock) o (existe un i tal que 0 <= i < |stock| y producto=stock[i]0
  y res = stock[i]1)}
}

Ejercicio 3 (2 puntos)

problema dineroEnStock (stock:seq⟨seq⟨Char⟩ x Z⟩, precios: seq⟨seq⟨Char⟩ x R⟩: R {
  requiere: {No hay productos repetidos en stock}
  requiere: {Todas las cantidades de los productos que hay en stock son mayores a cero}
  requiere: {No hay productos repetidos en precios}
  requiere: {Todos los precios de los productos son mayores a cero}
  requiere: {Todo producto en stock aparece en la lista de precios}
  asegura: {res es igual a la suma de los precios de todos los productos que están en stock multiplicado por la
  cantidad de cada producto que hay en stock}
}

Para resolver este ejercicio pueden utilizar la función del Preludio de Haskell fromIntegral que dado un valor de tipo Int devuelve su equivalente de tipo Float.

Ejercicio 4 (3 puntos)

```
problema aplicarOferta (stock:seq⟨seq⟨Char⟩ x Z⟩, precios: seq⟨seq⟨Char⟩ x R⟩: seq⟨seq⟨Char⟩ x R⟩ {
    requiere: {No hay productos repetidos en stock}
    requiere: {Todas las cantidades de los productos que hay en stock son mayores a cero}
    requiere: {No hay productos repetidos en precios}
    requiere: {Todos los precios de los productos son mayores a cero}
    requiere: {Todo producto en stock aparece en la lista de precios}
    asegura: {|res| = | precios|}
    asegura: {Para todo 0 <= i < |precios| si stockDeProducto(stock, precios[i]0) > 10 entonces res[i]0 = precios[i]0 y
    res[i]1 = precios[i]1 * 0,80}
      asegura: {Para todo 0 <= i < |precios| si stockDeProducto(stock, precios[i]0) <= 10 entonces res[i]0 = precios[i]0 y
    res[i]1 = precios[i]1}
}
```

Ejercicio 5 (1 punto)

Conteste marcando la opción correcta. En el contexto de la programación funcional, se llama polimorfismo:

★ **Cuando una función puede invocarse con distintos tipos de datos sin tener que redefinirla.**

☐ Cuando una función puede invocarse con distintos tipos de datos teniendo que definirla para cada tipo de dato particular.

☐ Cuando tengo un tipo de dato que puede ser usado para invocar a muchas funciones.

SOLUCIÓN

-- 1

```
generarStock :: [[Char]] -> [([Char],Int)]
generarStock [] = []
generarStock (p:ps) = añadirProducto p (generarStock ps)

añadirProducto :: [Char] -> [([Char],Int)] -> [([Char],Int)]
añadirProducto p [] = [(p,1)]
añadirProducto p ((n,c):ss) | p==n = ((n,c+1):ss)
 | otherwise = ((n,c):(añadirProducto p ss))
```

-- 2

```
stockDeProducto :: [([Char],Int)] -> [Char] -> Int
stockDeProducto [] _ = 0
stockDeProducto ((n,c):ss) p | p==n = c
 | otherwise = stockDeProducto ss p
```

-- 3

```
dineroEnStock :: [([Char],Int)] -> [([Char],Float)] -> Float
dineroEnStock [] _ = 0
dineroEnStock ((n,c):ss) ps = (precioProducto ps n)*(fromIntegral c)+(dineroEnStock ss ps)

precioProducto :: [([Char],Float)] -> [Char] -> Float
precioProducto ((n,v):ps) p | p==n = v
 | otherwise = precioProducto ps p
```

-- 4

```
aplicarOferta :: [([Char],Int)] -> [([Char],Float)] -> [([Char],Float)]
aplicarOferta _ [] = []
aplicarOferta ss ((n,v):ps) | (stockDeProducto ss n)>10 = ((n,v*0.80):(aplicarOferta ss ps))
 | otherwise = ((n,v):(aplicarOferta ss ps))
```