Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Sistema de stock

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. Implementar la función `productos :: [String] ->[(String, Int)]`

`problema` `generarStock` (productos: $seq\langle String\rangle$) : $seq\langle String \times \mathbb{Z}\rangle$ {
 `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 $0 \leq i < |res|$ y $res[i]_0$=*producto* y $res[i]_1$ es igual a la cantidad de veces que aparece *producto* en *productos*}
}

Ejercicio 2. Implementar la función `stockDeProducto :: [(String, Int))] ->String`

`problema` `stockDeProducto` (stock: $seq\langle String \times \mathbb{Z}\rangle$, producto: String) : $\mathbb{Z}$ {
 `requiere:` {No hay productos repetidos en *stock*}
 `requiere:` {Todas las cantidades (segundas componentes) de *stock* son mayores a cero}
 `asegura:` {($res = 0$ y *producto* no se encuentra en el stock) o (existe un i tal que $0 \leq i < |stock|$ y $producto = stock[i]_0$ y $res = stock[i]_1$}
}

Ejercicio 3. Implementar la función `dineroEnStock :: [(String, Int))] ->[(String, Float)] ->Float`

`problema` `dineroEnStock` (stock: $seq\langle String \times \mathbb{Z}\rangle$, precios: $seq\langle String \times \mathbb{R}\rangle$) : $\mathbb{R}$ {
 `requiere:` {No hay productos repetidos en *stock*}
 `requiere:` {No hay productos repetidos en *precios*}
 `requiere:` {Todas las cantidades (segundas componentes) de *stock* son mayores a cero}
 `requiere:` {Todas las precios (segundas componentes) de *precios* son mayores a cero}
 `requiere:` {Todo producto de *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. Implementar la función `aplicarOferta :: [(String, Int)] ->[(String, Float)] ->[(String,Float)]`

`problema` `aplicarOferta` (stock: $seq\langle String \times \mathbb{Z}\rangle$, precios: $seq\langle String \times \mathbb{R}\rangle$) : $seq\langle String \times \mathbb{R}\rangle$ {
 `requiere:` {No hay productos repetidos en *stock*}
 `requiere:` {No hay productos repetidos en *precios*}
 `requiere:` {Todas las cantidades (segundas componentes) de *stock* son mayores a cero}
 `requiere:` {Todas las precios (segundas componentes) de *precios* son mayores a cero}
 `requiere:` {Todo producto de *stock* aparece en la lista de precios}
 `asegura:` {$|res| = |precios|$}
 `asegura:` {Para todo $0 \leq i < |precios|$, 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 $0 \leq i < |precios|$, si stockDeProducto($stock$, $precios[i]_0$) ≤ 10, entonces $res[i]_0 = precios[i]_0$ y $res[i]_1 = precios[i]_1$ }

}

Sopa de números

Una **sopa de números** es un juego que consiste en descubrir propiedades de un tablero de dimensiones $n \times m$ con n y $m > 0$, en los que en cada posición hay un número entero positivo. Cada posición se identifica con una dupla (i, j) en el cual la primera componente corresponde a una fila y la segunda a una columna. A modo de ejemplo, la siguiente figura muestra un tablero de 5×4 en el que el número 13 aparece en la posición $(1, 1)$ y el número 5 aparece en la posición $(4, 3)$. Notar que tanto la numeración de las filas como la de las columnas comienzan en 1.

13	12	6	4
1	1	32	25
9	2	14	7
7	3	5	16
27	2	8	18

Un camino en un tablero está dado por una secuencia de posiciones adyacentes en la que solo es posible desplazarse desde una posición dada hacia la posición de su derecha o hacia la que se encuentra debajo. En otras palabras, un camino de longitud l en un tablero se define como una secuencia con l posiciones, ordenadas de manera tal que el elemento i-ésimo es la posición resultante de haberse movido hacia la derecha o hacia abajo desde la posición (i-1)-ésima. Siguiendo con el ejemplo, a continuación puede observarse un camino de longitud 5 que representa la sucesión Fibonacci y que empieza en la posición (2,1) y termina en (4,3) del tablero.

13	12	6	4
1	1	32	25
9	2	14	7
7	3	5	16
27	2	8	18

Para manipular las sopas de números en Haskell vamos a representar el tablero como una lista de filas de igual longitud. A su vez, cada fila vamos a representarla como una lista de enteros positivos. Las posiciones vamos a representarlas con tuplas de dos números enteros positivos y un camino va a estar dado por una lista de posiciones.

Para implementar esta sopa de números 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). Asumimos los siguientes renombres de tipos de datos en las especificaciones de los ejercicios:

- Fila = $seq\langle\mathbb{Z}\rangle$
- Tablero = $seq\langle Fila\rangle$
- Posicion = $\mathbb{Z} \times \mathbb{Z}$ – Observación: las posiciones son: (fila, columna)
- Camino = $seq\langle Posicion\rangle$

Ejercicio 5. Implementar la función `maximo :: Tablero ->Int`

`problema maximo` (t: Tablero) : $\mathbb{Z}$ {

requiere: {El tablero t es un tablero bien formado, es decir, la longitud de todas las filas es la misma, y tienen al menos un elemento}

requiere: {Existe al menos una columna en el tablero t }

requiere: {El tablero t no es vacío, todos los números del tablero son positivos, mayor estricto a 0}

asegura: {res es igual al número más grande del tablero t}

}

Ejercicio 6. Implementar la función `masRepetido :: Tablero ->Int`

`problema` `masRepetido` (t: Tablero) : $\mathbb{Z}$ {
  `requiere:` {El tablero t es un tablero bien formado, es decir, la longitud de todas las filas es la misma, y tienen al menos un elemento}
  `requiere:` {Existe al menos una columna en el tablero t }
  `requiere:` {El tablero t no es vacío, todos los números del tablero son positivos, mayor estricto a 0}
  `asegura:` {res es igual al número que más veces aparece en un tablero t. Si hay empate devuelve cualquiera de ellos}
}

Ejercicio 7. Implementar la función `valoresDeCamino :: Tablero ->Camino ->[Int]`

`problema` `valoresDeCamino` (t: Tablero, c: Camino) : $seq\langle\mathbb{Z}\rangle$ {
  `requiere:` {El tablero t es un tablero bien formado, es decir, la longitud de todas las filas es la misma, y tienen al menos un elemento}
  `requiere:` {Existe al menos una columna en el tablero t }
  `requiere:` {El tablero t no es vacío, todos los números del tablero son positivos, mayor estricto a 0}
  `requiere:` {El camino c es un camino válido, es decir, secuencia de posiciones adyacentes en la que solo es posible desplazarse hacia la posición de la derecha o hacia abajo y todas las posiciones están dentro de los limites del tablero t}
  `asegura:` {res es igual a la secuencia de números que están en el camino c, ordenados de la misma forma que aparecen las posiciones correspondientes en el camino.}
}

Ejercicio 8. Implementar la función `esCaminoFibo :: [Int] ->Int ->Bool`

`problema` `esCaminoFibo` (s:$seq\langle\mathbb{Z}\rangle$, i : $\mathbb{Z}$) : Bool {
  `requiere:` {La secuencia de números s es no vacía y está compuesta por números positivos (mayor estricto que 0) que representan los valores de un camino en un tablero}
  `requiere:` {$i \geq 0$}
  `asegura:` {res = true $\Leftrightarrow$ los valores de s son la sucesión de Fibonacci inicializada con el número pasado como parámetro i}
}

Notas: En este ejercicio se pasa una secuencia de valores en lugar de un tablero y un camino para no generar dependencia con el ejercicio anterior. Recordemos que la sucesión de Fibonacci está definida con la siguiente función recursiva:

```
f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2) con n>1
```

En el ejemplo del tablero y del camino (verde claro) que figuran más arriba tenemos que `esCaminoFibo [1,1,2,3,5] 1` reduce a True.

Perfectos amigos

El Departamento de Matemática (DM) de la FCEyN-UBA nos ha encargado que desarrollemos un sistema para el tratamiento de números naturales. Específicamente les interesa conocer cuándo un número es perfecto y cuándo dos números son amigos. Aunque por ahí no lo sabías, estos conceptos existen y se definen como:

- Un número natural es perfecto cuando la suma de sus divisores propios (números que lo dividen menores a él) es igual al mismo número. Por ejemplo, 6 es un número perfecto porque la suma de sus divisores propios (1,2 y 3) es igual a 6.
- Dos números naturales distintos son amigos si cada uno de ellos se obtiene sumando los divisores propios del otro. Por ejemplo, 220 y 284 son amigos porque los divisores propios de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110 que sumados dan 284 y los divisores propios de 284 son 1, 2 , 4, 71, 142 que sumados dan 220.

Para implementar este sistema nos enviaron las siguientes especificaciones en lenguaje semiformal 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 9. Implementar la función `divisoresPropios :: Int ->[Int]`

`problema` `divisoresPropios` (n: $\mathbb{Z}$) : $seq\langle\mathbb{Z}\rangle$ {
`requiere:` $\{n > 0\}$
`asegura:` $\{res$ es la lista de divisores propios de n, ordenada de menor a mayor$\}$
}

Ejercicio 10. Implementar la función `sonAmigos :: Int ->Int ->Bool`

`problema` `sonAmigos` (n,m: $\mathbb{Z}$) : Bool {
`requiere:` $\{n > 0\}$
`requiere:` $\{m > 0\}$
`requiere:` $\{m \neq n\}$
`asegura:` $\{res = \text{True} \Leftrightarrow n$ y m son números amigos$\}$
}

Ejercicio 11. Implementar la función `losPrimerosNPerfectos :: Int ->[Int]`

`problema` `losPrimerosNPerfectos` (n: $\mathbb{Z}$) : $seq\langle\mathbb{Z}\rangle$ {
`requiere:` $\{n > 0\}$
`asegura:` $\{res$ es la lista de los primeros n números perfectos, de menor a mayor$\}$
}

Por cuestiones de tiempos de ejecución, no les recomendamos que prueben este ejercicio con un $n > 4$.

Ejercicio 12. Implementar la función `listaDeAmigos :: [Int] ->[(Int,Int)]`

`problema` `sonAmigos` (lista: $seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z} \times \mathbb{Z}\rangle$ {
`requiere:` $\{$Todos los números de $lista$ son mayores a 0$\}$
`requiere:` $\{$Todos los números de $lista$ son distintos$\}$
`asegura:` $\{res$ es una lista de tuplas sin repetidos, que contienetuplas de dos números donde esos dos números pertenecen a $lista$ y son amigos$\}$
`asegura:` $\{|res|$ es la cantidad de tuplas de dos números amigos que hay en lista. Consideraremos que la tupla (a, b) (con a y b pertenecientes a $\mathbb{Z}$) es igual a la tupla (b, a) para contar la cantidad de tuplas.$\}$
}