Compra venta

Se necesita desarrollar un sistema en Haskell que permita administrar transacciones financieras y comerciales internacionales.

Existen personas que cuentan entre sus ahorros sumas de dinero en diferentes monedas. Además se conoce su nivel de satisfacción. Por ejemplo:

cacho = UnaPersona "pedro" [(100, "peso"), (400, "real"), (1, "dolar")] 4
tita = UnaPersona "juana" [(1000, "euro"), ((-5), "peso")] 2

Los productos que se comercializan, se representan con el siguiente tipo de dato:
data Producto = UnProducto {descripcion::String, precio::Float, moneda::String}

También se conocen las cotizaciones de las monedas

```
cotizaciones :: [(Float,String)]
cotizaciones = [(1, "peso"), (9, "dolar"), (4, "real"), (8,"euro")]
```

1) Para entrar en calor

Hacer las funciones que permitan realizar lo siguiente

a. Obtener el tipo de cambio de una moneda
   >tipoCambio "dolar"
   9

b. Convertir un cierto importe, de una moneda a otra
   >convertirA "real" (8, "dolar")
   18

c. Obtener la cantidad de dinero que tiene una persona en una moneda dada
   >cantidadDe "real" cacho
   400

d. Calcular el total de ahorro de una persona, expresado en pesos.
   >totalAhorro cacho
   1709
   *(100*1 + 400*4 + 1 * 9)*

Importante: Utilizar, al menos una vez, la función composición y una expresión lambda

2) Transacciones

Una persona puede realizar diferentes transacciones comerciales. En todos los casos se quiere saber cómo queda la persona luego de la transacción. Las transacciones siempre se pueden realizar, si no se tiene dinero suficiente en la moneda indicada, se registra como una deuda, con un importe negativo. Definir las funciones que permitan representar cada transacción.

- Comprar: La persona adquiere un producto, por lo que se descuenta de sus ahorros el precio del producto, en la moneda en que está expresado.
- Vender: La persona vende un producto y recibe el importe correspondiente a su precio, y lo registra prolijamente acumulándolo según la moneda.

Nota para todo el examen pero en particular para este punto: evitar repetir lógica

3) Estilos de consumo

Se quiere representar diferentes estilos que las personas despliegan cuando van a ciertos negocios.

El negocio cuenta con una dirección, una lista de productos y un estilo. Los estilos son:

- compulsivo: El cliente compra todos los productos.
- impulsivo: El cliente compra sólo el primer producto.
- selectivo: El cliente compra los que tienen precio en pesos.

a. Hacer una función por la que cuando un cliente va a un negocio, se comporte según el estilo correspondiente y su nivel de satisfacción aumente en una unidad.
b. Incorporar un nuevo estilo, denominado exclusivo, por el que el cliente compra el más

caro. Realizarlo utilizando la función maximum y definiendo adecuadamente el tipo de dato del producto a comprar.

c. Que pasa si la lista de productos es infinita. Mostrar ejemplos donde sucedan diferentes cosas.

SOLUCION

```
type Moneda = String
type Importe = Float
type Dinero = (Importe, Moneda)
--No define nuevos tipos, sino simplemente crea sinónimos de tipo, alias, para ganar expresividad y escribir menos al definir los tipos de datos de las funciones

importe = fst
descMoneda = snd

--importe::Dinero->Importe
--importe (x,_) = x

--descMoneda::Dinero->Moneda
--descMoneda (_,x) = x

data Persona = UnaPersona {nombre::String, ahorros::[Dinero],satisfaccion::Int}  deriving Show

cacho = UnaPersona "pedro" [(100, "peso"), (400, "real"), (1, "dolar")] 4
tita = UnaPersona "juana" [(1000, "euro"), ((-5), "peso")] 2

--Los productos que se comercializan, se representan con el siguiente tipo de dato:
data Producto = UnProducto {descripcion::String, precio::Importe, moneda::Moneda} deriving (Eq , Show)

bici = UnProducto "bicicleta" 500 "peso"
libro = UnProducto "origami" 10 "yen"

--También se conocen las cotizaciones de las monedas
cotizaciones::[Dinero]
cotizaciones = [(1, "peso"), (9, "dolar"), (4, "real"), (8,"euro"),(20,"yen")]

--1) Para entrar en calor

-- A
tipoCambio:: Moneda -> Importe
tipoCambio moneda = buscarImporteDe moneda cotizaciones

buscarImporteDe:: Moneda -> [Dinero] -> Importe
buscarImporteDe unaMoneda dineros = (importe.head.filter ((unaMoneda ==).descMoneda)) dineros
```

```
-- B
convertirA:: Moneda -> Dinero -> Importe
convertirA desde (importe,hasta) = importe * tipoCambio desde / tipoCambio hasta

-- C
cantidadDe:: Moneda -> Persona -> Importe
--cantidadDe moneda persona = buscarImporteDe moneda (ahorros persona)

cantidadDe moneda persona
        | tiene moneda (ahorros persona) = buscarImporteDe moneda (ahorros persona)
        | otherwise = 0

tiene moneda ahorros = any ((moneda==).descMoneda) ahorros

-- D
calculaAhorro:: Persona -> Importe
calculaAhorro persona = (sum.map (convertirA "peso").ahorros) persona


--2) Transacciones

--Comprar: La persona adquiere un producto, por lo que se descuenta de sus ahorros el precio del producto, en la moneda en que está expresado.
--Vender: La persona vende un producto y recibe el importe correspondiente  a su precio, y lo registra prolijamente acumulándolo según la moneda.

vender (UnaPersona nombre ahorros satisfaccion) producto = UnaPersona nombre (acumular (precio producto, moneda producto) ahorros) satisfaccion

comprar persona (UnProducto nombre precio moneda)   = vender persona (UnProducto nombre (-precio) moneda)
--comprar (UnaPersona nombre ahorros satisfaccion) (UnProducto _  precio moneda) = UnaPersona nombre ( acumular ((-precio),moneda) ahorros) satisfaccion


acumular:: Dinero -> [Dinero]-> [Dinero]
{-
acumular dinero [] = [dinero]
acumular (importe,moneda) ((importeAhorro,monedaAhorro):ahorros)
        |moneda == monedaTotal = (importe+importeAhorro,moneda):ahorros
   |otherwise = (importeAhorro,monedaAhorro): acumular (importe,moneda) ahorros
-}
```

```
acumular dinero ahorros
        | tiene (descMoneda dinero) ahorros = map (incrementarMismaMoneda dinero) ahorros
        | otherwise = dinero:ahorros

incrementarMismaMoneda (importe,moneda) (importeAhorro,monedaAhorro)
        |moneda == monedaAhorro = (importe+importeAhorro,moneda)
   |otherwise = (importeAhorro,monedaAhorro)

--3) Estilos de consumo

type Estilo = [Producto]->[Producto]
data Negocio = UnNegocio {direccion::String, productos::[Producto], estilo::Estilo}

banco = UnNegocio "Medrano 900" [bici,libro] compulsivo
joya = UnNegocio "Rivadavia 4000" [bici,libro] exclusivo

-- A)
compulsivo,impulsivo,selectivo:: Estilo
compulsivo productos = productos
--compulsivo = id
impulsivo productos = [head productos]
--impulsivo (x:_) = [x]
selectivo productos = filter (("peso"==).moneda) productos

irA negocio persona = (aumentarSatisfaccion.comprarSegunEstilo negocio) persona

comprarSegunEstilo (UnNegocio _ productos estilo) persona = foldl comprar persona (estilo
productos)

aumentarSatisfaccion (UnaPersona n a satisfaccion) = UnaPersona n a (satisfaccion + 1)

-- B)
instance Ord Producto where
        (<=) (UnProducto _ precio1 moneda1) (UnProducto _ precio2 moneda2) =
precio1*tipoCambio moneda1 <= precio2*tipoCambio moneda2

exclusivo::Estilo
exclusivo productos = [maximum productos]
```