Práctica de ejercicios #4 - Ejercicios Integradores

Estructuras de Datos, Universidad Nacional de Quilmes

16 de septiembre de 2020

1. Pizzas

Tenemos los siguientes tipos de datos:

```
data Pizza = Prepizza
           | Capa Ingrediente Pizza

data Ingrediente = Salsa
                 | Queso
                 | Jamon
                 | Aceitunas Int
```

Definir las siguientes funciones:

- `cantidadDeCapas :: Pizza -> Int`
  Dada una pizza devuelve la lista de capas de ingredientes
- `armarPizza :: [Ingrediente] -> Pizza`
  Dada una lista de ingredientes construye una pizza
- `sacarJamon :: Pizza -> Pizza`
  Le saca los ingredientes que sean jamón a la pizza
- `tieneSoloSalsaYQueso :: Pizza -> Bool`
  Dice si una pizza tiene salsa y queso
- `duplicarAceitunas :: Pizza -> Pizza`
  Recorre cada ingrediente y si es aceitunas duplica su cantidad
- `cantCapasPorPizza :: [Pizza] -> [(Int, Pizza)]`
  Dada una lista de pizzas devuelve un par donde la primera componente es la cantidad de ingredientes de la pizza, y la respectiva pizza como segunda componente.

2. Mapa de tesoros (con bifurcaciones)

Un mapa de tesoros es un árbol con bifurcaciones que terminan en cofres. Cada bifurcación y cada cofre tiene un objeto, que puede ser chatarra o un tesoro.

```
data Dir = Izq | Der

data Objeto = Tesoro | Chatarra

data Cofre = Cofre [Objeto]

data Mapa = Fin Cofre
          | Bifurcacion Cofre Mapa Mapa
```

Definir las siguientes operaciones:

1. `hayTesoro :: Mapa -> Bool`
   Indica si hay un tesoro en alguna parte del mapa.
2. `hayTesoroEn :: [Dir] -> Mapa -> Bool`
   Indica si al final del camino hay un tesoro. Nota: el final de un camino se representa con una lista vacía de direcciones.
3. `caminoAlTesoro :: Mapa -> [Dir]`
   Indica el camino al tesoro. Precondición: existe un tesoro y es único.
4. `caminoDeLaRamaMasLarga :: Mapa -> [Dir]`
   Indica el camino de la rama más larga.
5. `tesorosPorNivel :: Mapa -> [[Objeto]]`
   Devuelve los tesoros separados por nivel en el árbol.
6. `todosLosCaminos :: Mapa -> [[Dir]]`
   Devuelve todos lo caminos en el mapa.

3. Nave Espacial

modelaremos una `Nave` como un tipo algebraico, el cual nos permite construir una nave espacial, dividida en sectores, a los cuales podemos asignar tripulantes y componentes. La representación es la siguiente:

```
data Componente = LanzaTorpedos | Motor Int | Almacen [Barril]
data Barril = Comida | Oxigeno | Torpedo | Combustible

data Sector = S SectorId [Componente] [Tripulante]
type SectorId = String
type Tripulante = String

data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
data Nave = N (Tree Sector)
```

Implementar las siguientes funciones utilizando recursión estructural:

1. `sectores :: Nave -> [SectorId]`
   Propósito: Devuelve todos los sectores de la nave.
2. `poderDePropulsion :: Nave -> Int`
   Propósito: Devuelve la suma de poder de propulsión de todos los motores de la nave. Nota: el poder de propulsión es el número que acompaña al constructor de motores.
3. `barriles :: Nave -> [Barril]`
   Propósito: Devuelve todos los barriles de la nave.
4. `agregarASector :: [Componente] -> SectorId -> Nave -> Nave`
   Propósito: Añade una lista de componentes a un sector de la nave.
   Nota: ese sector puede no existir, en cuyo caso no añade componentes.
5. `asignarTripulanteA :: Tripulante -> [SectorId] -> Nave ->\ Nave`
   Propósito: Incorpora un tripulante a una lista de sectores de la nave.
   Precondición: Todos los id de la lista existen en la nave.
6. `sectoresAsignados :: Tripulante -> Nave -> [SectorId]`
   Propósito: Devuelve los sectores en donde aparece un tripulante dado.

7. `tripulantes :: Nave ->\ [Tripulante]`
   Propósito: Devuelve la lista de tripulantes, sin elementos repetidos.

4. Manada de lobos

Modelaremos una manada de lobos, como un tipo `Manada`, que es un simple registro compuesto de una estructura llamada `Lobo`, que representa una jerarquía entre estos animales.
Los diferentes casos de lobos que forman la jerarquía son los siguientes:

- Los cazadores poseen nombre, una lista de especies de presas cazadas y 3 lobos a cargo.
- Los exploradores poseen nombre, una lista de nombres de territorio explorado (nombres de bosques, ríos, etc.), y poseen 2 lobos a cargo.
- Las crías poseen sólo un nombre y no poseen lobos a cargo.

La estructura es la siguiente:

```
type Presa = String -- nombre de presa
type Territorio = String -- nombre de territorio
type Nombre = String -- nombre de lobo
data Lobo = Cazador Nombre [Presa] Lobo Lobo Lobo
          | Explorador Nombre [Territorio] Lobo Lobo
          | Cría Nombre
data Manada = M Lobo
```

1. Construir un valor de tipo `Manada` que posea 1 cazador, 2 exploradores y que el resto sean crías. Resolver las siguientes funciones utilizando *recursión estructural* sobre la estructura que corresponda en cada caso:

2. `buenaCaza :: Manada -> Bool`
   Propósito: dada una manada, indica si la cantidad de alimento cazado es mayor a la cantidad de crías.

3. `elAlfa :: Manada -> (Nombre, Int)`
   Propósito: dada una manada, devuelve el nombre del lobo con más presas cazadas, junto con su cantidad de presas. Nota: se considera que los exploradores y crías tienen cero presas cazadas, y que podrían formar parte del resultado si es que no existen cazadores con más de cero presas.

4. `losQueExploraron :: Territorio -> Manada -> [Nombre]`
   Propósito: dado un territorio y una manada, devuelve los nombres de los exploradores que pasaron por dicho territorio.

5. `exploradoresPorTerritorio :: Manada -> [(Territorio, [Nombre])]`
   Propósito: dada una manada, denota la lista de los pares cuyo primer elemento es un territorio y cuyo segundo elemento es la lista de los nombres de los exploradores que exploraron dicho territorio. Los territorios no deben repetirse.

6. `superioresDelCazador :: Nombre -> Manada -> [Nombre]`
   Propósito: dado un nombre de cazador y una manada, indica el nombre de todos los cazadores que tienen como subordinado al cazador dado (directa o indirectamente).
   Precondición: hay un cazador con dicho nombre y es único.