Departamento de Computação
Universidade Federal de Ouro Preto
Programação Funcional
Prof. Lucília Figueiredo

Lista de Exercícios 03 - Tipos de Dados Algébricos

1. O tipo de dados dos números naturais pode ser definido do seguinte modo:

```
data Nat = Zero | Succ Nat
```

   Defina as seguintes funções sobre esse tipo de dado:

   (a) `pred :: Nat -> Nat` que retorna o predecessor de um número natural.

   (b) `isZero :: Nat -> Bool` que determina se um número natural é igual a `Zero`.

   (c) `plus :: Nat -> Nat` que soma dois número naturais.

   (d) `mult :: Nat -> Nat` que multiplica dois número naturais.

   (e) compare :: Nat -¿ Nat -¿ Ordering que compara dois números naturais. O tipo de dado `Ordering` é definido como

```
data Ordering = LT | EQ | GT
```

2. Uma árvore binária pode ser definida do seguinte modo:

```
data BTree a = Nil | Node (BTree a) a (BTree a)
```

   Defina as funções?

```
mapT :: (a->b) -> BTree a -> BTree b
filterT :: (a -> Bool) -> BTree a -> BTree a
elemT :: Eq a => a -> BTree a -> Bool
```

   que operam sobre árvores binárias da mesma forma que `map`, `filter` e `elem` operam sobre listas.

3. Defina a função `heightBT :: BTree a -> BTree (a,Int)` que, dada uma árvore binária, retorna a árvore binária obtida incluindo em cada nodo a altura desse nodo na árvore.

4. Considerando que a árvore binária definida anteriormente é uma árvore binária de pesquisa:

   (a) Defina a função `insertT Ord a => :: a -> BTree a -> BTree a` tal que `insert x t` insere o elemento `x` na árvore binária `t`, de maneira que a árvore resultante seja uma árvore binária de pesquisa.

   (b) Defina a função `flatten :: Btree a => [a]` tal que `flatten t` retorna uma lista ordenada dos valores contidos nos nodos da árvore `t`.

5. O tipo árvore pode ser definido do seguinte modo?

```
data Tree a = Node a a (Tree a)
```

Defina a função `minTree :: Ord a => Tree a -> Tree a` que substitui os valores em todos os nodos da árvore pelo valor mínimo de todos os nodos. A função deve operar em um único passo sobre a árvore.

6. Um *móbile* é constituído de pendentes, fios e barras. Em cada uma das extremidades de uma barra de um mobile é preso um fio, no qual pode estar pendurado um pendente ou uma nova barra. Barras e fios são considerados elementos sem peso e pendentes possuem um peso (representado por um valor inteiro). O seguinte tipo de dado pode ser usado para representar um móbile:

```
data Mobile = Pendente Int | Barra Mobile Mobile
```

O *peso de um móbile* é igual à soma dos pesos de todos os seus pendentes. Um móbile é *balanceado* se ele consiste de um único pendente ou se os pesos dos móbiles pendurados nas duas extremidades da sua barra são iguais e esses móbiles são balanceados. Defina as seguintes funções sobre móbiles:

   (a) `peso :: Mobile -> Int`, que retorna o peso do móbile dado como argumento.

   (b) `balanceado :: Mobile -> Bool`, que determina se o móbile dado como argumento é ou não balanceado.

7. Os itens a seguir sugerem os passo para que você defina a função `makeBMobile :: [Int] -> Maybe Mobile` que constrói um móbile balanceado, caso isso seja possível, com pendentes cujos pesos são os valores da lista dada como argumento. Em outras palavras, `makeBMobile ns` retorna `Just mob`, se for possível construir um móbile balanceado `mob` com os pesos da lista `ns`, ou retorna `Nothing`, caso não seja possível construir um móbile balanceado.

   Lembre-se que que o tipo de dado `Maybe` é definido como:

```
data Maybe a = Nothing | Just a
```

   (a) Defina a função `splits :: [a] -> [([a],[a])]`, que retorna a lista de todas as possíveis partições, em duas listas, da lista dada como argumento . Por exemplo:

```
splits [1,2,3] ▷ [([1,2,3],[]),([2,3],[1]),
                  ([1,3],[2]),([3],[1,2]),
                  ([1,2],[3]),([2],[1,3]),
                  ([1],[2,3]),([],[1,2,3])]
```

   (b) Use as funções `filter` e `splits` para definir a função `eqsplits :: [a] -> [([a],[a])]`, que retorna a lista de todas as possíveis partições balanceadas da lista dada como argumento, ou seja, lista dos pares nos quais a soma dos valores, para as duas listas, é a mesma. Por exemplo:

```
eqsplits [1,2,3] ▷ [([3],[1,2]),([1,2],[3])]
```

   (c) Finalmente, defina a função `makeBMobile`, usando `eqsplits`, completando a definição a seguir:

```
makeBMobile ::  [Int] -> Mobile
makeBMobile []  = Nothing
makeBMObile [n] = Just (Pendente n)
makeBMobile ns  = ...
```