Ficha 9

Programaçao Funcional

2015/16

1. Considere o seguinte tipo para representar expressões inteiras.

```
data ExpInt = Const Int
            | Simetrico ExpInt
            | Mais ExpInt ExpInt
            | Menos ExpInt ExpInt
            | Mult ExpInt ExpInt
```

Os termos deste tipo `ExpInt` podem ser vistos como arvores cujas folhas são inteiros e cujos nodos (nao folhas) sao operadores.

(a) Defina uma funcão `calcula :: ExpInt -> Int` que, dada uma destas expressões calcula o seu valor.

```
calcula :: ExpInt -> Int
calcula (Const n) = n
calcula (Simetrico e) = (calcula e) * (-1)
calcula (Mais a b) = (calcula a) + (calcula b)
calcula (Menos a b) = (calcula a) - (calcula b)
calcula (Mult a b) = (calcula a) * (calcula b)
```

(b) Defina uma funcao `infixa :: ExpInt -> String` de forma a que `infixa (Mais (Const 3) (Menos (Const 2) (Const 5)))` dê como resultado `"(3 + (2 - 5))"`.

```
infixx :: ExpInt -> String
infixx (Const n) = show n
infixx (Simetrico x) = "(-" ++ (infixx x) ++ ")"
infixx (Mais x y) = "(" ++ (infixx x) ++ " + " ++ (infixx y) ++ ")"
infixx (Menos x y) = "(" ++ (infixx x) ++ " - " ++ (infixx y) ++ ")"
infixx (Mult x y) = "(" ++ (infixx x) ++ " x " ++ (infixx y) ++ ")"
```

(c) Defina uma outra funcao de conversão para strings `posfixa :: ExpInt -> String` de forma a que quando aplicada à expressão acima dê como resultado `"3 2 5 - +"`.

```
posfix :: ExpInt -> String
posfix (Const n) = show n
posfix (Simetrico e) = posfix e ++ " n"
posfix (Mais a b) = posfix a ++ " " ++ posfix b ++ "+"
posfix (Menos a b) = posfix a ++ " " ++ posfix b ++ "-"
posfix (Mult a b) = posfix a ++ " " ++ posfix b ++ "+"
```

2. Considere o seguinte tipo para representar arvores irregulares (rose trees).

```
data RTree a = R a [RTree a]
```

Defina as seguintes funçoes sobre estas arvores:

(a) `soma :: (Num a) => (RTree a) -> a` que soma os elementos da arvore.

soma :: (Num a) => (RTree a) -> a
soma (R valor []) = valor
soma (R valor subNodes) = valor + (sum (map soma subNodes))

(b) `altura :: (RTree a) -> Int` que calcula a altura da arvore.

altura :: (RTree a) -> Int
altura (R _ subNodes) = 1 + (foldl max 0 (map altura subNodes))

(c) `prune :: Int -> (RTree a) -> (RTree a)` que remove de uma arvore todos os elementos a partir de uma determinada profundidade.

prune :: Int -> (RTree a) -> (RTree a)
prune 1 (R v _) = R v []
prune x (R v subNodes) = R v (map (prune (x - 1)) subNodes)

(d) `mirror :: (RTree a) -> (RTree a)` que gera a arvore simétrica.

mirror :: (RTree a) -> (RTree a)
mirror (R v subNodes) = R v (map mirror (reverse subNodes))

(e) `postorder :: (RTree a) -> [a]` que corresponde à travessia postorder da arvore.

postorder :: (RTree a) -> [a]
postorder (R v subNodes) = foldr (++) [v] (map postorder subNodes)

3. Relembre a definiçao de arvores binarias apresentada na Ficha 8:

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

Nestas arvores a informaçao está nos nodos (as extermidades da arvore têm apenas uma marca – `Empty`).

(a) É também habitual definirem-se arvores em que a informação está apenas nas extermidades (leaf trees):

```
data LTree a = Tip a | Fork (LTree a) (LTree a)
```

Defina sobre este tipo as seguintes funções

i. `ltSum :: (Num a) => (LTree a) -> a` que soma as folhas de uma arvore.

ltSum :: (Num a) => (LTree a) -> a
ltSum (Tip a) = a
ltSum (Fork x y) = (ltSum x) + (ltSum y)

ii. `listaLT :: (LTree a) -> [a]` que lista as folhas de uma arvore (da esquerda para a direita).

listLT :: (LTree a) -> [a]
listLT (Tip a) = [a]
listLT (Fork x y) = (listLT x) ++ (listLT y)

iii. `ltHeight :: (LTree a) -> Int` que calcula a altura de uma arvore.

ltHeight :: (LTree a) -> Int
ltHeight (Tip _) = 1
ltHeight (Fork x y) = 1 + (max (ltHeight x) (ltHeight y))

(b) Estes dois conceitos podem ser agrupados num só, definindo o seguinte tipo:

```
data FTree a b = Leaf b | No a (FTree a b) (FTree a b)
```

Sao as chamadas full trees onde a informação está nao só nos nodos, como também nas folhas (note que o tipo da informacão nos nodos e nas folhas nao tem que ser o mesmo).

i. Defina a funçao `splitFTree :: (FTree a b) -> (BTree a, LTree b)` que separa uma arvore com informação nos nodos e nas folhas em duas arvores de tipos diferentes.

```
splitFTree :: (FTree a b) -> (BTree a, LTree b)
splitFTree (Leaf x) = (Empty,Tip x)
splitFTree (No x l r) = let (l1, l2) = splitFTree l
                            (r1, r2) = splitFTree r
                        in (Node x l1 r1, Fork l2 r2)
```

ii. Defina ainda a funcão inversa `joinTrees :: (BTree a) -> (LTree b) -> Maybe (FTree a b)` que sempre que as arvores sejam compatíveis as junta numa so.

```
joinTrees :: (BTree a) -> (LTree b) -> Maybe (FTree a b)
joinTrees Empty (Tip x) = Just (Leaf x)
joinTrees (Node x e d) (Fork e' d') = case (joinTrees e e') of
    Nothing -> Nothing
    Just e1 -> case (joinTrees d d') of
        Nothing -> Nothing
        Just d1 -> Just (No x e1 d1)
joinTrees _ _ = Nothing
```