Programação Funcional

Teste 2022-01-12

Voltar

1. Apresente uma definição recursiva da função (pré-definida) `zip :: [a] -> [b] -> [(a,b)]` que constrói uma lista de pares a partir de duas listas. Por exemplo, `zip [1,2,3] [10,20,30,40]` corresponde a `[(1,10),(2,20),(3,30)]`.

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (h:t) (h':t') = (h,h') : zip t t'
```

2. Defina a função `preCrescente :: Ord a => [a] -> [a]` que calcula o maior prefixo crescente de uma lista. Por exemplo, `preCrescente [3,7,9,6,10,22]` corresponde a `[3,7,9]` e `preCrescente [1,2,7,9,9,1,8]` corresponde a `[1,2,7,9]`.

```
preCrescente :: Ord a => [a] -> [a]
preCrescente [] = []
preCrescente [x] = [x]
preCrescente (h:s:t)
    | s >= h = h : preCrescente (s:t)
    | otherwise = [h]
```

3. A amplitude de uma lista de inteiros define-se como a diferença entre o maior e o menor dos elementos da lista (a amplitude de uma lista vazia é 0). Defina a função `amplitude :: [Int] -> Int` que calcula a amplitude de uma lista (idealmente numa única passagem pela lista).

```
amplitude :: [Int] -> Int
amplitude l = uncurry (flip (-)) . foldr (\x (acc_min,acc_max) -
```

```
> (min x acc_min, max x acc_max)) (head l, head l) $ l
```

4. Considere o seguinte tipo `type Mat a = [[a]]` para representar matrizes. Defina a função `soma :: Num a => Mat a -> Mat a -> Mat a` que soma duas matrizes da mesma dimensão.

```
soma :: Num a => Mat a -> Mat a -> Mat a
soma = zipWith . zipWith $ (+)
```

5. Decidiu-se organizar uma agenda telefónica numa árvore binária de procura (ordenada por ordem alfabética de nomes). Para isso, declararam-se os seguintes tipos de dados:

```
type Nome = String
type Telefone = Integer
data Agenda = Vazia | Nodo (Nome, [Telefone]) Agenda Agenda
```

Defina Agenda como instância da classe Show de forma a que a visualização da árvore resulte numa listagem da informação ordenada por ordem alfabética (com um registo por linha) e em que os vários telefones associados a um nome se apresentem separados por /.

```
import Data.List (intercalate)

instance Show Agenda where
    show Vazia = ""
    show (Nodo (nome, tlfs) l r) =
        show l
        ++ nome ++ " " ++ intercalate "/" (map show tlfs) ++
"\n"
        ++ show r
```

6. Defina uma função `randomSel :: Int -> [a] -> IO [a]` que dado um inteiro `n` e uma lista `l`, produz uma lista com `n` elementos seleccionados aleatoriamente de `l`. Um elemento não pode aparecer na lista produzida mais

vezes do que aparece na lista argumento. Se `n` for maior do que o comprimento da lista a função deverá retornar uma permutação da lista argumento. Por exemplo, a invocação de `randomSel 3 [1,3,1,4,2,8,9,5]` poderia produzir qualquer uma das listas `[1,4,2]`, `[5,2,8]` ou `[1,9,1]`, mas nunca `[2,3,2]`.

```
randomSel :: Show a => Int -> [a] -> IO [a]
randomSel _ [] = return []
randomSel 0 _ = return []
randomSel n l =
    randomRIO (1, length l)
    >>= (\randomN ->
        fmap (l !! (randomN - 1) :) (randomSel (n - 1) (take
(randomN - 1) l ++ drop randomN l))
    )
```

7. Defina uma função `organiza :: Eq a => [a] -> [(a, [Int])]` que, dada uma lista constrói uma lista em que, para cada elemento da lista original se guarda a lista dos índices onde esse elemento ocorre. Por exemplo, `organiza "abracadabra"` corresponde a `[('a',[0,3,5,7,10]), ('b',[1,8]), ('r',[2,9]), ('c',[4]), ('d',[6])]`.

```
organiza :: Eq a => [a] -> [(a,[Int])]
organiza = foldr (\a -> insere a . map (\(c,is) -> (c,map (+1)
is))) []

insere :: Eq a => a -> [(a,[Int])] -> [(a,[Int])]
insere x [] = [(x,[0])]
insere x ((c,is):t)
    | x == c = (c, 0 : is) : t
    | otherwise = (c,is) : insere x t
```

8. Apresente uma definição alternativa da função `func`, usando recursividade explícita em vez de funções de ordem superior e fazendo uma única travessia da lista.

```
func :: [[Int]] -> [Int]
func 1 = concat (filter (x -> sum x >10) 1)
```

```
func :: [[Int]] -> [Int]
func [] = []
func (h:t)
    | sum h > 10 = h ++ func t
    | otherwise = func t
```

9. Considere a seguinte estrutura para manter um dicionário, onde as palavras estão organizadas de forma alfabética.

```
data RTree a = R a [RTree a]
type Dictionary = [ RTree (Char, Maybe String) ]

d1 = [R ('c',Nothing) [
        R ('a',Nothing) [
            R ('r', Nothing) [
                R ('a',Just "...") [
                    R ('s', Just "...") [] ],
                R ('o',Just "...") [],
                R ('r',Nothing) [
                    R ('o',Just "...") [] ]
    ]  ]   ]   ]
```

Cada árvore agrupa todas as palavras começadas numa dada letra. As palavras constroem-se descendo na árvore a partir da raiz. Quando uma palavra está completa, o valor associado à última letra é `Just s`, sendo `s` uma string com a descrição da palavra em causa (que corresponde ao caminho desde a raiz até aí). Caso contrário é `Nothing`.

Por exemplo, `d1` é um dicionário com as palavras: *cara, caras, caro* e *carro*.

Defina a função `insere :: String -> String -> Dictionary -> Dictionary` que, dadas uma palavra e a informação a ela associada, acrescenta essa entrada no dicionário. Se a palavra já existir no dicionário, atualiza a informação a ela associada.

```haskell
insere :: String -> String -> Dictionary -> Dictionary
insere [x] desc dict = insereFim x desc dict
insere (h:t) desc [] = [ R (h,Nothing) (insere t desc [])]
insere (h:t) desc (R (a,b) l:d)
    | h == a = R (a,b) (insere t desc l) : d
    | otherwise = R (a,b) l : insere (h:t) desc d

insereFim :: Char -> String -> Dictionary -> Dictionary
insereFim x desc [] = [ R (x,Just desc) [] ]
insereFim x desc (R (a,b) l:t)
    | x == a = R (a,Just desc) l : t
    | otherwise = R (a,b) l : insereFim x desc t
```