Universidade Federal de Santa Maria
Centro de Tecnologia
Departamento de Linguagens e Sistemas de Computação
ELC117 - Paradigmas de Programação
Profª Andrea Schwertner Charão

Exercícios de Programação Funcional em Haskell

Entrega:
Coloque os exercícios num repositório versionado, num arquivo t5.hs.
Informe a URL do repositório em http://bit.do/entrega-paradigmas
Prazo: quarta-feira, 09/09, 23:55

Nos exercícios abaixo, é possível usar funções auxiliares para decompor o problema, mas não é permitido usar funções existentes que implementem diretamente o que é solicitado.

1. Escreva uma função `addSuffix :: String -> [String] -> [String]` usando *list comprehension*, para adicionar um dado sufixo às strings contidas numa lista. Exemplo:

```
> addSuffix "@inf.ufsm.br" ["fulano","beltrano"]
["fulano@inf.ufsm.br","beltrano@inf.ufsm.br
```

2. Reescreva a função do exercício acima, desta vez usando **recursão**.

3. Escreva uma função `countShorts :: [String] -> Int`, que receba uma lista de palavras e retorne a quantidade de palavras dessa lista que possuem menos de 5 caracteres. Use **recursão**.

4. Reescreva a função do exercício acima, desta vez usando *list comprehension*.

5. Escreva uma função `ciclo :: Int -> [Int] -> [Int]` que receba um número N e uma lista de inteiros, retornando uma nova lista com N repetições da lista original, conforme o exemplo abaixo:

```
> ciclo 4 [1,3]
[1,3,1,3,1,3,1,3]
```

   Obs.: Você deve usar **recursão** neste exercício.

6. Escreva uma função **recursiva** `combine :: [Int] -> [String] -> [(Int,String)]`, que receba duas listas e combine seus elementos em tuplas. Exemplo de uso:

```
> combine [10,11,12] ["dez","onze","doze"]
[(10,"dez"),(11,"onze"),(12,"doze")]
```

7. Escreva uma função numera :: [String] -> [(Int,String)], que receba uma lista de palavras e retorne outra lista contendo tuplas com as palavras numeradas a partir de 1. Use **recursão**. Exemplo de uso da função:

```
> numera ["abacaxi","mamao","banana"]
[(1,"abacaxi"),(2,"mamao"),(3,"banana")]
```

8. Explique, em forma de comentário, o resultado de cada expressão abaixo.
```
a) [ (x,y) | x <- [1..5], even x, y <- [(x + 1)..6], odd y ]
b) [ a ++ b | a <- ["lazy","big"], b <- ["frog", "dog"]]
c) concat [ [a,'-'] | a <- "paralelepipedo", a `elem` "aeiou"]
```

9. (G. Malcolm, Univ. Liverpool) Write a function `crossProduct :: [a] -> [b] -> [(a,b)]` that takes two lists xs and ys, and returns the list of all possible pairings:

```
[ (x,y) | x <- xs, y <- ys ]
```

without using the above list comprehension. (As an exercise in problem decomposition, try first defining a "helper" function

```
pairWithAll :: a -> [b] -> [(a,b)]
```

that pairs its first argument with each element in its second.)

10. Suponha que um retângulo seja representado por uma tupla (Float,Float,Float,Float), contendo respectivamente as coordenadas x e y do ponto no seu canto superior esquerdo, seguidas das suas medidas de largura e altura. Sabendo que o eixo x cresce de cima para baixo e o eixo y da esquerda para direita, crie uma função `genRects :: Int -> (Int,Int) -> [(Float,Float,Float,Float)]` que receba um número N e um ponto (x,y) e gere uma sequência de N retângulos não sobrepostos. Os retângulos devem ser alinhados pelos seus topos, a partir do ponto dado, com largura e altura constantes. Por exemplo, usando largura e altura iguais a 5.5:

```
> genRects 3 (0,0)
[(0.0,0.0,5.5,5.5),(5.5,0.0,5.5,5.5),(11.0,0.0,5.5,5.5),(16.5,0.0,5.5,5.5)]
```

Obs.: Use conversão explícita de tipos quando misturar Int e Float.

11. Escreva uma função **recursiva** que receba uma lista de tuplas e decomponha cada uma delas, gerando uma tupla de listas, conforme o exemplo abaixo:

```
> func  [(1,3),(2,4)]
([1,2], [3,4])
```

12. Refaça o exercício anterior usando *list comprehension*.

13. Refaça o exercício anterior usando uma função de alta ordem.