Exercícios para tutoria. Semana 2.

Programação Funcional

Prof. Rodrigo Ribeiro

Introdução

Setup inicial

Importando biblioteca com funções para manipulação de caracteres.

```
import Data.Char
```

A seguir, importamos a bibliotecas para construção de testes de programas Haskell. Utilizaremos funções destas bibliotecas para construção de testes para os exercícios deste material.

```
import           Test.Tasty
import           Test.Tasty.HUnit
import qualified Test.Tasty.QuickCheck  as QC
```

A seguinte função `main` é usada apenas para execução dos testes para as funções deste material.

```
main :: IO ()
main = defaultMain tests
```

Ao contrário da semana anterior em que você realizou testes usando apenas o interpretador, nestes exercícios você deverá conferir seus resultados utilizando a bateria de testes fornecida. Para execução dos testes, você deverá utilizar os seguintes comandos:

```
$> stack build
$> stack exec semana2-exe
```

O primeiro é responsável por compilar o projeto e o segundo de executá-lo.

Assim como no material da semana anterior, você deve substituir as chamadas para a função

```
tODO :: a
tODO = undefined
```

que interompe a execução do programa com uma mensagem de erro, por código que implementa as funcionalidades requeridas por cada exercício.

Descrição do material

Esse material consiste em exercícios sobre o conteúdo de recusividade e funções de ordem superior.

Antes de resolver os exercícios contidos nesse material, recomendo que você faça todos os exercícios presentes nos slides das aulas:

- Recursão sobre listas
- Tipos em Haskell
- Funções de ordem superior

Recursão sobre listas

1. Neste exercício você deverá definir uma função que retorna todos os elementos de uma lista de inteiros que estão dentro de um certo intervalo numérico.

a) Desenvolva a função

```
inRange :: Int -> Int -> [Int] -> [Int]
inRange = tODO
```

que retorna todos os números da lista de entrada que estão no intervalo especificado pelos primeiros dois parâmetros usando list comprehensions.

b) Desenvolva a função

```
inRangeRec :: Int -> Int -> [Int] -> [Int]
inRangeRec = tODO
```

que retorna todos os números da lista de entrada que estão no intervalo especificado pelos primeiros dois parâmetros usando recursão. O comportamento destas funções deve atender os seguintes testes unitários:

```
inRangeTest :: TestTree
inRangeTest
   = testCase "In range list comprehension" $
        inRange 5 10 [1..15] @?= [5..10]

inRangeRecTest :: TestTree
inRangeRecTest
   = testCase "In range recursion" $
        inRangeRec 5 10 [1..15] @?= [5..10]
```

c) Finalmente, suas duas implementações devem produzir o mesmo resultado para todas as entradas. Para isso, vamos utilizar a bibliotecs de testes baseados em propriedades, QuickCheck. Para isso, implemente a função

```
propInRange :: Int -> Int -> [Int] -> Bool
propInRange = tODO
```

que deve retornar verdadeiro somente quando os resultados de `inRange` e `inRangeRec` forem idênticos para os mesmos valores de entrada.

2. Neste exercício você deverá implementar funções para contar o número de inteiros positivos estritamente maiores que zero em uma lista.

a) Implemente a função

```
countPositives :: [Int] -> Int
countPositives = tODO
```

usando list comprehensions. Sua implementação deve atender o seguinte caso de teste.

```
countPositivesTest :: TestTree
countPositivesTest
    = testCase "countPositives list comprehension" $
        countPositives [0, 1, -3, -7, 8, -1, 6] @?= 3
```

b) Implemente a função

```
countPositivesRec :: [Int] -> Int
countPositivesRec = tODO
```

usando recursividade. Sua implementação deve atender o seguinte caso de teste.

```
countPositivesRecTest :: TestTree
countPositivesRecTest
    = testCase "countPositives recursion" $
        countPositivesRec [0, 1, -3, -7, 8, -1, 6] @?= 3
```

c) Escreva a função

```
propCountPositive :: [Int] -> Bool
propCountPositive = tODO
```

que retorna verdadeiro se o resultado de `countPositives` e `countPositivesRec` coincidem para a lista de entrada.

3. O objetivo deste exercício é a construção de uma função que converta uma determinada string de entrada para um formato de título. Dizemos que uma string está em formato de título se o seu primeiro caractere é uma letra maiúscula e as demais letras são minúsculas.

a) Implemente a função

```
toTitleString :: String -> String
toTitleString = tODO
```

que converte uma string de entrada para o formato de títulos. Sua implementação deve atender o seguinte teste unitário.

```
toTitleStringTest :: TestTree
toTitleStringTest
  = testCase "toTitle test case" $
        toTitleString "bERNardino 123" @?= "Bernardino 123"
```

b) Escreva uma função que caracterize a propriedade de correção da implementação de toTitle.

```
propToTitleStringCorrect :: String -> Bool
propToTitleStringCorrect = tODO
```

3. Considere a tarefa de implementar uma função que retorna a metada de cada número par presente em uma lista.

a) Implemente a função

```
halfEvens :: [Int] -> [Int]
halfEvens = tODO
```

que divide pela metade todos os números pares presentes em uma lista. Sua definição deve usar list comprehensions e não recursão. Sua função deve satisfazer o seguinte teste.

```
halfEvensTest :: TestTree
halfEvensTest
   = testCase "halfEvens Test" $
       halfEvens [0, 2, 1, 7, 8, 56, 17, 18] @?= [0, 1, 4, 28, 9]
```

b) Implemente a função

```
halfEvensRec :: [Int] -> [Int]
halfEvensRec = tODO
```

que divide pela metade todos os números pares presentes em uma lista usando recursividade. Sua função deve satisfazer o seguinte teste.

```
halfEvensRecTest :: TestTree
halfEvensRecTest
   = testCase "halfEvensRec Test" $
       halfEvensRec [0, 2, 1, 7, 8, 56, 17, 18] @?= [0, 1, 4, 28, 9]
```

c) Escreva a função

```
propHalfEvens :: [Int] -> Bool
propHalfEvens = tODO
```

que retorna verdadeiro sempre que o resultado de `halfEvens` e `halfEvensRec` for idêntico.

Funções de ordem superior

1. Implemente a função

```
uppers :: String -> String
uppers = tODO
```

que converte para maiúsculas todas as letras de uma string de entrada. Você deve implementar `uppers` utilizando a função `map`. Sua implementação deve satisfazer o seguinte caso de teste

```
uppersTest :: TestTree
uppersTest
   = testCase "uppers unit" $ uppers "aBCd12fG" @?= "ABCD12FG"
```

Além disso, apresente uma propriedade de correção para sua implementação de `uppers`.

```
propUppersCorrect :: String -> Bool
propUppersCorrect = tODO
```

2. Implemente a função

```
centsToReals :: [Int] -> Float
centsToReals = tODO
```

que converte cada preço em centavos para o equivalente em reais. Sua implementação deve satisfazer o seguinte teste unitário:

```
centsToRealsTest :: TestTree
centsToRealsTest
  = testCase "centsToReals unit" $ centsToReals [100, 200, 350] @?= [1, 2, 3.5]
```

3. Implemente a função

```
alphas :: String -> String
alphas = tODO
```

que remove todos os caracteres alfa-numéricos da string fornecida como entrada. Sua implementação deve utilizar a função `filter` e satisfazer o seguinte o seguinte teste unitário:

```
alphasTest :: TestTree
alphasTest
  = testCase "alphas unit" $ alphas "1abc2d67e8" @?= "abcde"
```

Adicionalmente, especifique uma propriedade que deve ser satisfeita pela implementação de `alphas`:

```
propAlphasCorrect :: String -> Bool
propAlphasCorrect = tODO
```

4. Implemente a função

```haskell
above :: Int -> [Int] -> [Int]
above = tODO
```

que remove todos os elementos menores que um certo valor de uma lista de inteiros. Sua implementação deve utilizar a função `filter` e satisfazer o seguinte teste unitário:

```haskell
aboveTest :: TestTree
aboveTest
  = testCase "above unit" $ above 5 [1,7,-2,3,8,10,15,9] @?= [7,8,10,15,9]
```

Adicionalmente, apresente uma propriedade de correção para sua implementação de `above`.

```haskell
propAboveCorrect :: Int -> [Int] -> Bool
propAboveCorrect = tODO
```

Funções auxiliares

```haskell
inRangeGroup :: TestTree
inRangeGroup
    = testGroup "In Range tests"
                [
                  inRangeTest,
                  inRangeRecTest,
                  QC.testProperty "In Range Equivalence" propInRange
                ]

countPositivesGroup :: TestTree
countPositivesGroup
    = testGroup "countPositives tests"
                [
                  countPositivesTest,
                  countPositivesRecTest,
                  QC.testProperty "countPositives Equivalence" propCountPositive
                ]

toTitleStringGroup :: TestTree
toTitleStringGroup
   = testGroup "toTitle tests"
               [
                 toTitleStringTest,
                 QC.testProperty "toTitle correct" propToTitleStringCorrect
               ]

halfEvensGroup :: TestTree
halfEvensGroup
  = testGroup "halfEvens tests"
```

```
                [
                  halfEvensTest,
                  halfEvensRecTest,
                  QC.testProperty "halfEvens Equivalence" propHalfEvens
                ]

uppersGroup :: TestTree
uppersGroup
  = testGroup "uppers tests"
                [
                  uppersTest,
                  QC.testProperty "uppers correct" propUppersCorrect
                ]

alphasGroup :: TestTree
alphasGroup
  = testGroup "alphas tests"
               [
                 alphasTest,
                 QC.testProperty "alphas correct" propAlphasCorrect
               ]

aboveGroup :: TestTree
aboveGroup
  = testGroup "above tests"
               [
                 aboveTest,
                 QC.testProperty "above correct" propAboveCorrect
               ]

tests :: TestTree
tests
   = testGroup "Semana 2 tests"
                [
                  inRangeGroup,
                  countPositivesGroup,
                  toTitleStringGroup,
                  halfEvensGroup,
                  centsToRealsTest
                ]
```