Exercícios para tutoria. Semana 3.

Programação Funcional

Prof. Rodrigo Ribeiro

Introdução

Setup inicial

Inicialmente, vamos importar bibliotecas para construção de testes de programas Haskell. Utilizaremos funções destas bibliotecas para construção de testes para os exercícios deste material.

```
import           Control.Monad
import           Data.List                    (transpose)
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 primeira semana, 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 semana3-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 listas 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

Manipulação de matrizes

O objetivo dos exercícios a seguir é a implementação de funções para soma e multiplicação de matrizes. Para isso, representaremos matrizes como listas de listas de números:

```haskell
type Matrix = [[Float]]
```

A construção `type` de Haskell permite a definição de um *sinônimo de tipo*, isto é, um novo nome para um tipo existente. Como um exemplo dessa representação, considere a seguinte matriz:

$$\begin{bmatrix} 1 & 4 & 9 \\ 2 & 5 & 7 \end{bmatrix}$$

esta pode ser representada pelo seguinte valor:

```haskell
example :: Matrix
example = [ [1, 4, 9]
          , [2, 5, 7]]
```

Com base no apresentado, faça o que se pede:

1. O objetivo deste exercício é validar se uma lista constitui uma matriz válida. Para ser considerada válida, uma matriz deve: 1) possuir pelo menos uma linha e coluna e

2) todas as linhas possuem o mesmo número de elementos. Para isso, desenvolva as funções a seguir:
   a) Escreva a função `uniform` que testa se uma lista é uniforme. Dizemos que uma lista é uniforme se todos os seus elementos são iguais.

```haskell
uniform :: [Float] -> Bool
uniform = tODO
```

Os seguintes casos de teste devem ser satisfeitos por sua implementação de `uniform`.

```haskell
uniformTests :: TestTree
uniformTests
   = testGroup "Tests for uniform"
               [
                 testCase "uniform empty"  $ uniform []        @?= True
               , testCase "uniform single" $ uniform [1]       @?= True
               , testCase "uniform many"   $ uniform [1,1,1,1] @?= True
               , testCase "uniform fail"   $ uniform [1..5]    @?= False
               ]
```

b) Usando a função `uniform`, escreva a função

```haskell
valid :: Matrix -> Bool
valid = tODO
```

que testa de uma matriz é válida ou não. Sua função deve passar nos seguintes casos de test

```haskell
validTests :: TestTree
validTests
  = testGroup "Tests for valid"
              [
                testCase "valid empty"   $ valid []          @?= False
              , testCase "valid single"  $ valid [[1,2,3]]   @?= True
              , testCase "valid example" $ valid example     @?= True
              , testCase "valid fail"    $ valid [[1],[1,2]] @?= False
              ]
```

2. Desenvolva a função

```haskell
dimension :: Matrix -> (Int,Int)
dimension = tODO
```

que retorna as dimensões (número de linhas e colunas) de uma dada matriz. Sua definição deve considerar que a dimensão de matrizes inválidas é definida como `(0,0)`.

```haskell
dimensionTests :: TestTree
dimensionTests
  = testGroup "Tests for dimension"
              [
                testCase "dimension empty"   $ dimension []        @?= (0,0)
              , testCase "dimesnion single"  $ dimension [[1,2,3]] @?= (1,3)
              , testCase "dimension example" $ dimension example   @?= (2,3)
              ]
```

Usando sua definição de `dimension`, implemente uma função que testa se uma matriz é quadrada. *Lembre-se*: Matrizes inválidas não são quadradas!

```
square :: Matrix -> Bool
square = tODO

squareTests :: TestTree
squareTests
  = testGroup "Tests for square"
              [
                testCase "square empty"   $ square []            @?= False
              , testCase "square single"  $ square [[1,2,3]]     @?= False
              , testCase "square example" $ square example       @?= False
              , testCase "square success" $ square [[1,2],[3,4]] @?= True
              ]

question2Tests :: TestTree
question2Tests
  = testGroup "Tests for exercice 2"
              [
                dimensionTests
              , squareTests
              ]
```

3. Implemente a função

```
idMatrix :: Int -> Matrix
idMatrix = tODO
```

que a partir de um inteiro positivo $n \geq 1$ gera a matriz identidade de dimensão $n \times n$.

```
question3Tests :: TestTree
question3Tests
  = testGroup "Tests for idMatrix"
              [
                testCase "idMatrix 0" $ idMatrix 0 @?= []
              , testCase "idMatrix 1" $ idMatrix 1 @?= [[1]]
              , testCase "idMatrix 2" $ idMatrix 2 @?= [[1,0],[0,1]]
              , QC.testProperty "idMatrix square" idMatrixSquareProp
              ]

idMatrixSquareProp :: QC.Property
idMatrixSquareProp
  = QC.forAll (QC.suchThat (QC.arbitrary :: QC.Gen Int) (> 0))
              (\ n -> square (idMatrix n) == True)
```

4. Uma função muito útil para lidar com listas é `zipWith`, cuja definição é apresentada a seguir:

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _              = []
zipWith f _ []              = []
zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
```

Apresente uma definição de `zipWith` em termos das funções `map`, `zip` e `uncurry`.

```haskell
zipWith' ::  (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' = tODO
```

5. Somar duas matrizes de mesma dimensão consiste em adicionar os elementos que ocorrem na mesma posição (i.e., mesma linha e mesma coluna) nas duas matrizes, para obter o elemento nessa posição na matriz resultante. Por exemplo:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{bmatrix}$$

a) Desenvolva a função

```haskell
nullMatrix :: (Int,Int) -> Matrix
nullMatrix = tODO
```

que produz uma matriz nula (com zeros em todas as posições) de dimensões às fornecidas como parâmetros.

b) Defina a função

```haskell
addRow :: [Float] -> [Float] -> [Float]
addRow = tODO
```

que soma duas linhas de uma matriz.

```haskell
addRowTest :: TestTree
addRowTest
  = testGroup "addRow tests"
              [
                testCase "addRow test 1" $ addRow [1,2,3] [10,20,30] @?= [11, 22, 33]
              , testCase "addRow test 2" $ addRow [4,5,6] [40,50,60] @?= [44, 55, 66]
              ]
```

b) Usando as funções `addRow` e `zipWith`, defina a função para soma de duas matrizes.

```haskell
(.+.) :: Matrix -> Matrix -> Matrix
_ .+. _ = tODO
```

Sua implementação deve satisfazer os seguintes testes.

```haskell
sumMatrixTest :: TestTree
sumMatrixTest
  = testGroup "sumMatrix tests"
```

```
              [
                testCase "sumMatrix single" $ [[1]] .+. [[2]] @?= [[3]]
              , testCase "sumMatrix zero"   $ [[1,2],[3,4]] .+. [[0,0],[0,0]] @?= [[1,2],[3,
              , QC.testProperty "sumMatrix null" $ nullMatrixProp
              ]

nullMatrixProp :: QC.Property
nullMatrixProp
  = QC.forAll matrixGen
              (\ m -> m .+. (nullMatrix (dimension m)) == m)
```

6. Para implementação do produto de duas matrizes, vamos precisar da noção de produto interno, ou produto de dois vetores:

$$(a_1, a_2, ..., a_n).(b_1, b_2, ..., b_n) = a_1b_1 + a_2b_2 + ... + a_nb_n$$

A multiplicação de matrizes é definida do seguinte modo: duas matrizes de dimensões $n \times m$ e $m \times p$ são multiplicadas de modo a formar uma matriz de dimensões $n \times p$, na qual o elemento da linha i e coluna j é o resultado do produto interno da linha i da primeira matriz pela coluna j da segunda. Por exemplo:

$$\begin{bmatrix} 1 & 10 \\ 100 & 10 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 31 & 42 \\ 130 & 240 \end{bmatrix}$$

Com base no apresentado, faça o que se pede.

a) Implemente a função

```
innerProduct :: [Float] -> [Float] -> Float
innerProduct = tODO
```

que calcula o produto interno de dois vetores fornecidos como argumento. Sua função deve atender o seguinte caso de teste.

```
innerProductTest :: TestTree
innerProductTest
  = testGroup "Inner product tests"
              [
                testCase "inner product a11" $ innerProduct [1,10] [1,3] @?= 31
              , testCase "inner product a12" $ innerProduct [1,10] [2,4] @?= 42
              , testCase "inner product a21" $ innerProduct [100, 10] [1,3] @?= 130
              , testCase "inner product a22" $ innerProduct [100, 10] [2,4] @?= 240
              ]
```

b) Usando sua implementação de produto interno, implemente uma função para multiplicar duas matrizes. Observe que ao tentarmos multiplicar matrizes cujas dimensões não sejam apropriadas, você deverá retornar uma lista vazia.

```haskell
(.*.) :: Matrix -> Matrix -> Matrix
_ .*. _ = tODO

prodMatrixTest :: TestTree
prodMatrixTest
  = testGroup "Matrix multiplication tests"
              [
                testCase "matrix multiplication fail 1" $ [] .*. [] @?= []
              , testCase "matrix multiplication fail 2" $ [[2,1],[2,2]] .*. [[1]] @?= []
              , testCase "matrix multiplication success" $ [[1,10],[100,10]] .*. [[1,2],[3,4
              ]
```

Funções auxiliares

```haskell
question1Tests :: TestTree
question1Tests
   = testGroup "Tests for exercice 1"
               [
                 uniformTests
               , validTests
               ]

zipWithProperty :: QC.Property
zipWithProperty
  = QC.forAll (QC.arbitrary :: QC.Gen ([Int],[Int]))
           (\ (xs,ys) -> zipWith' (+) xs ys == zipWith (+) xs ys)

question4Tests :: TestTree
question4Tests
  = testGroup "Tests for exercice 4"
              [
                QC.testProperty "zipWith' test" zipWithProperty
              ]

buildFromDimensions :: Int -> Int -> QC.Gen Matrix
buildFromDimensions n m
  = replicateM n (replicateM m genEntry)
    where
      genEntry = QC.choose (1,10) :: QC.Gen Float

matrixGen :: QC.Gen Matrix
matrixGen
  = do
       n <- QC.choose (1,5) :: QC.Gen Int
       m <- QC.choose (1,5) :: QC.Gen Int
       buildFromDimensions n m
```

```haskell
question5Tests :: TestTree
question5Tests
  = testGroup "Tests for exercice 5"
              [
                sumMatrixTest
              ]

question6Tests :: TestTree
question6Tests
  = testGroup "Tests for exercice 6"
              [
                innerProductTest
              , prodMatrixTest
              ]

tests :: TestTree
tests
   = testGroup "Semana 3 tests"
               [
                 question1Tests,
                 question2Tests,
                 question3Tests,
                 question4Tests,
                 question5Tests,
                 question6Tests
               ]
```