Exercícios para tutoria. Semana 1.

Programação Funcional

Prof. Rodrigo Ribeiro

Introdução

Esse material consiste em exercícios sobre o conteúdo introdutório da linguagem Haskell. Todos os exercícios envolvem problemas de cunho matemático que servem para familiarizar o aluno com a sintaxe da linguagem, o ambiente de programação (editor de texto favorito e interpretador GHCi) e funções recursivas simples.

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

- Primeiros passos em Haskell.
- Definindo funções simples.

Nos exercícios seguintes, 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 requerida por cada exercício.

A seguinte função `main` é usada apenas para omitir mensagens de erro do compilador de Haskell. Você pode ignorá-la.

```
main :: IO ()
main = return ()
```

Recursão sobre inteiros

Exercício 1

Considere a seguinte função:

$$f(n) = \begin{cases} \frac{n}{2} & \text{se } n \text{é par.} \\ 3n+1 & \text{caso contrário.} \end{cases}$$

Esta simples função é um componente de um problema em aberto na matemática: Aplicações sucessivas dessa função eventualmente atingem o valor 1.

Com base no apresentado, faça o que se pede:

a) Codifique a função

```
next :: Int -> Int
next = tODO
```

que implementação da função f(n) apresentada anteriormente.

b) Usando a definição de `next`, implemente a função

```
steps :: Int -> Int
steps = tODO
```

que retorna 1, caso o valor fornecido sobre entrada seja 1. Caso contrário, `steps` é chamada recursivamente sobre o resultado de aplicar a função `nexts` sobre o parâmetro da função `steps`.

c) Uma maneira de medir empiricamente o tempo de execução de funções é utilizar um contador de chamadas recursivas. Modifique a implementação da função `steps` (item - b) de forma que esta retorne um par contendo o resultado de sua execução e o número de chamadas recursivas realizadas.

```
stepsCounter :: Int -> (Int,Int)
stepsCounter = tODO
```

Como convenção, considere que o primeiro componente do par retornado armazena o resultado da função e o segundo o número de chamadas recursivas.

d) Você deve ter observado que a função `steps` sempre retorna o número 1 como resultado. Modifique a implementação de `steps` de forma que esta função retorne a sequência númerica gerada por chamadas recursivas até obtero resultado final, o número 1.

```
stepsList :: Int -> ([Int],Int)
stepsList = tODO
```

Exercício 2

O algoritmo para cálculo do máximo divisor comum de dois inteiros é definido pela seguinte função:

$$gcd(a,b) = \begin{cases} a & \text{se } b = 0 \\ gcd(b, a\ mod\ b) & \text{caso contrário} \end{cases}$$

a) Implemente o algoritmo para cálculo do máximo divisor comum em Haskell:

```haskell
gcd :: Int -> Int -> Int
gcd = tODO
```

b) Modifque a implementação da função `gcd` de forma a retornar o número de chamadas recursivas realizadas.

```haskell
gcdCounter :: Int -> Int -> (Int, Int)
gcdCounter = tODO
```

Método de Newton.

Os exercícios seguintes descrevem um conjunto de funções para implementar o método de Newton para cálculo da raiz quadrada de um número.

O método de Newton é uma função que a partir do radicando (valor para o qual desejamos calcular a raiz quadrada) e uma aproximação tenativa produz uma nova aproximação do resultado da raiz quadrada para o radicando em questão.

Exercício 1

Defina uma função chamada `average` que calcula a média de dois valores fornecidos como parâmetro.

```haskell
average :: Double -> Double -> Double
average = tODO
```

Exercício 2

Dizemos que uma tentativa `guess` para a raiz quadrada de `x` é válida se o valor absoluto da diferença do quadrado de `guess` e `x` for menor que 0.001. Baseado nesse fato, implemente a função `goodEnough` que verifica se uma tentativa é boa o suficiente.

```haskell
goodEnough :: Double -> Double -> Bool
goodEnough = tODO
```

Exercício 3

Método de Newton consiste em repetir uma função de melhoria de resultado até atingir uma determinada precisão. A melhoria de uma tentativa é a média desse valor e o resultado de dividir o radicando pela tentativa.

Implemente a função `improve` que a partir de uma tentativa e do radicando retorna uma nova tentativa como resultado.

```
improve :: Double -> Double -> Double
improve = tODO
```

Exercício 4

De posse das funções anteriores, podemos implementar o método de Newton para cálculo da raiz quadrada de um número. Para isso, defina a função

```
sqrtIter :: Double -> Double -> Double
sqrtIter = tODO
```

que a partir de uma tentativa e do radicando, testa se a tentativa é atende a restrição de precisão (usando a função `goodEnough`). Caso a tentativa não atenda a restrição, devemos executar `sqrtIter` usando como nova tentativa o resultado de `improve` sobre a tentativa atual.

Exercício 5

O método de Newton para raízes cúbicas é baseado no fato de que se `y` é uma aproximação da raiz cúbida de `x`, então uma aproximação melhor é dada pela seguinte fórmula:

$$\frac{\frac{x}{y^2} - 2y}{3}$$

Implemente a função `cubicIter` que a partir de uma tentativa e do radicando retorna uma aproximação para raiz cúbica do radicando considerando como tolerância o valor 0.0001.

```
cubicIter :: Double -> Double -> Double
cubicIter = tODO
```

List comprehensions

Exercício 1

A função `sum` calcula a soma de elementos presente em uma lista de números. Usando a função `sum` e list comprehensions construa uma função que calcule a soma dos quadrados dos primeiros `n` números inteiros, em que `n` é um parâmetro da função.

```haskell
squareSum :: Int -> Int
squareSum = tODO
```

Exercício 2

Uma maneira de representar um plano de coordenadas de tamanho $m \times n$ é usando uma lista de pares (x, y) de números inteiros tais que $0 \leq x \leq n$ e $0 \leq y \leq m$. Usando list comprehension, construa a função

```haskell
grid :: Int -> Int -> [(Int,Int)]
grid = tODO
```

que retorna um plano de coordenadas de acordo com a descrição apresentada.

Exercício 3

O produto escalar de duas listas de inteiros de tamanho n é definido como

$$\sum_{k=0}^{n-1}(xs_k \times ys_k)$$

em que xs_k é o k-ésimo elemento da lista xs. Implemente a função

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

que calcula o produto escalar de duas listas fornecidas como argumento.