Avaliação 1 de Programação Funcional

ATENÇÃO

- A interpretação dos enunciados faz parte da avaliação.
- A avaliação deve ser resolvida INDIVIDUALMENTE. Plágios não serão tolerados. TODAS as avaliações em que algum tipo de plágio for detectado receberão nota ZERO.
- Se você utilizar recursos disponíveis na internet e que não fazem parte da bibliografia, você deverá explicitamente citar a fonte apresentando o link pertinente como um comentário em seu código.
- Todo código produzido por você deve ser acompanhado por um texto explicando a estratégia usada para a solução. Lembre-se: meramente parafrasear o código não é considerado uma explicação!
- Não é permitido eliminar a diretiva de compilação -Wall do cabeçalho desta avaliação.
- Caso julgue necessário, você poderá incluir bibliotecas adicionais incluindo "imports" no cabeçalho deste módulo.
- Seu código deve ser compilado sem erros e warnings de compilação. A presença de erros acarretará em uma penalidade de 20% para cada erro de compilação e de 10% para cada warning. Esses valores serão descontados sobre a nota final obtida pelo aluno.
- Todo o código a ser produzido por você está marcado usando a função "undefined". Sua solução deverá substituir a chamada a undefined por uma implementação apropriada.
- Todas as questões desta avaliação possuem casos de teste para ajudar no entendimento do resultado esperado. Para execução dos casos de teste, basta executar os seguintes comandos:

```
$> stack build
$> stack exec prova1-exe
```

- Sobre a entrega da solução:

1. A entrega da solução da avaliação deve ser feita como um único arquivo .zip contendo todo o projeto stack usado.
2. O arquivo .zip a ser entregue deve usar a seguinte convenção de nome: MATRÍCULA.zip, em que matrícula é a sua matrícula. Exemplo: Se sua matrícula for 20.1.2020 então o arquivo entregue deve ser 2012020.zip. A não observância ao critério de nome e formato da solução receberá uma penalidade de 20% sobre a nota obtida na avaliação.

3. O arquivo de solução deverá ser entregue usando a atividade “Entrega da Avaliação 1” no Moodle dentro do prazo estabelecido.
4. É de responsabilidade do aluno a entrega da solução dentro deste prazo.
5. Sob NENHUMA hipótese serão aceitas soluções fora do prazo ou entregues usando outra ferramenta que não a plataforma Moodle.

Setup inicial

```
{-# OPTIONS_GHC -Wall #-}

module Main where

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain tests

tests :: TestTree
tests
  = testGroup "Unit tests"
        [
           question01Tests
        ,  question02Tests
        ,  question03Tests
        ,  question04Tests
        ,  question05Tests
        ]
```

Questão 1. Escreva a função

```
question01 :: [Integer] -> [Integer]
question01 = undefined
```

que recebe uma lista de inteiros como entrada e retorna como resultado uma lista de inteiros em que todo número ímpar presente na lista é elevado ao quadrado. Sua implementação deve atender os seguintes casos de teste.

```
question01Tests :: TestTree
question01Tests
      = testGroup "Question 01 Tests"
                  [
                     testCase "Question 01 empty" $
                         question01 [] @?= []
                  ,  testCase "Question 01 all even" $
                         question01 (map (* 2) [1..5]) @?= map (* 2) [1..5]
                  ,  testCase "Question 01 some odd" $
                         question01 [1..5] @?= [1,2,9,4,25]
                  ]
```

Questão 2. Considere o seguinte tipo de dados:

```
data Times = Zero | One | Two
```

Sua tarefa é implementar a função:

```
question02 :: Times -> (a, a, a) -> (a, a, a)
question02 = undefined
```

que a partir de um valor do tipo `Times` e uma tripla de valores de tipo `a`, retorna uma tripla na qual os valores foram rotacionados um número de vezes especificado pelo tipo `Times`. Os casos de teste a seguir apresentam exemplos desta função.

```
question02Tests :: TestTree
question02Tests
      = testGroup "Question 02 Tests"
                  [
                     testCase "Swapping Zero times:" $
                          question02 Zero ("a","b","c") @?= ("a","b","c")
                  ,  testCase "Swapping One time:" $
                          question02 One ("a", "b", "c") @?= ("c", "a", "b")
                  ,  testCase "Swapping Two times:" $
                          question02 Two ("a", "b", "c") @?= ("b", "c", "a")
                  ]
```

Questão 03. Considere o seguinte tipo de dados que representa dados de clientes de uma loja:

```
type Name  = String
type Phone = String
type Email = String

data Client = Client Name Phone Email deriving (Eq, Show)
```

Dizemos que a informação de um cliente é válida se:

a) O nome do cliente possui pelo menos 3 caracteres e é formado exclusivamente por letras e espaços.

b) A informação de telefone é composta apenas por dígitos

c) A string de email deve conter o caractere `@` e ter tamanho maior que 3.

Com base nessas informações, desenvolva a função:

```
question03 :: Client -> Bool
question03 = undefined
```

que verifica se a informação de cliente é ou não válida de acordo com as regras mencionadas anteriormente.

Sua implementação deve considerar os seguintes casos de teste.

```haskell
question03Tests :: TestTree
question03Tests
      = testGroup "Question 03 Tests"
                  [
                     testCase "Valid client" $
                       question03 (Client "Marcos" "123456789" "marcos@bla.com") @?= True
                  ,  testCase "Invalid name - size" $
                       question03 (Client "Mr" "123456789" "marcos@bla.com") @?= False
                  ,  testCase "Invalid name - not all letters" $
                       question03 (Client "Mr22" "123456789" "marcos@bla.com") @?= False
                  ,  testCase "Invalid phone" $
                       question03 (Client "Marcos" "ab23" "marcos@bla.com") @?= False
                  ,  testCase "Invalid email - size" $
                       question03 (Client "Marcos" "123456789" "m@") @?= False
                  ,  testCase "Invalid email - lacking @" $
                       question03 (Client "Marcos" "123456789" "marcobla.com") @?= False
                  ]
```

Questão 04. Um inconveniente da solução apresentada no exercício 03 é que a função não apresenta uma explicação do motivo da validação falhar. Uma alternativa para isso é criar um tipo de dados para representar as possíveis falhas de validação.

```haskell
data Error = NameLengthError        -- invalid size
           | NameCharactersError    -- name with non-letters and space characters
           | PhoneError             -- phone with non numeric chars.
           | EmailSizeError         -- invalid size
           | EmailCharError         -- lacking `@`
           deriving (Eq, Show)
```

Usando a representação de erros de validação, podemos definir um tipo para representar a validação:

```haskell
data Validation = Ok
                | Failure [Error] deriving (Eq, Show)
```

O construtor `Ok` representa que a validação executou com sucesso e o construtor `Failure` representa uma falha de validação e armazena uma lista dos erros encontrados.

Com base no apresentado, implemente a função.

```haskell
question04 :: Client -> Validation
question04 = undefined
```

que realiza a validação de clientes, como apresentado na questão 03, e retorna um valor do tipo `Validation`. Sua implementação deve atender os seguintes casos de teste.

```haskell
question04Tests :: TestTree
```

```
question04Tests
      = testGroup "Question 04 Tests"
                  [
                     testCase "Valid client" $
                       question04 (Client "Marcos" "123456789" "marcos@bla.com") @?= Ok
                  ,  testCase "Invalid name - size" $
                       question04 (Client "Mr" "123456789" "marcos@bla.com") @?= Failure [Na
                  ,  testCase "Invalid name - not all letters" $
                       question04 (Client "Mr22" "123456789" "marcos@bla.com") @?= Failure
                  ,  testCase "Invalid phone" $
                       question04 (Client "Marcos" "ab23" "marcos@bla.com") @?= Failure [Pho
                  ,  testCase "Invalid email - size" $
                       question04 (Client "Marcos" "123456789" "m@") @?= Failure [EmailSizeE
                  ,  testCase "Invalid email - lacking @" $
                       question04 (Client "Marcos" "123456789" "marcobla.com") @?= Failure
                  ,  testCase "Combining errors" $
                       question04 (Client "Mr" "aa" "b@") @?= Failure [NameLengthError, Phon
                  ]
```

Questão 05. Considere o seguinte tipo de dados que representa a configuração de uma aplicação em um sistema gerenciador de janelas:

```
data App
      = App { name :: String   -- application name
            , width :: Int     -- window width
            , height :: Int    -- window height
            }
        deriving (Eq, Show)
```

Aplicações são organizadas de acordo com um layout:

```
data Layout = Vertical [Layout]
            | Horizontal [Layout]
            | Single App
            deriving (Eq, Show)
```

Neste gerenciador de janelas simples, aplicações são organizadas de maneira vertical (construtor `Vertical`), horizontal (construtor `Horizontal`) ou uma janela simples.

Seu objetivo é implementar a função:

```
minimizeAll :: Layout -> Layout
minimizeAll = undefined
```

que minimiza todas as janelas do estado do gerenciador de janelas. Uma janela é minimizada fazendo com que sua altura (height) e comprimento (width) sejam iguais a 1.

Sua implementação deve atender os seguintes casos de teste.

```haskell
question05Tests :: TestTree
question05Tests
      = testGroup "Question 05 Tests"
                  [
                      testCase "Minimize Single" $
                        minimizeAll (Single (App "test" 110 200)) @?= Single (App "test" 1 1
                  ,   testCase "Minimize Vertical" $
                        minimizeAll (Vertical [ Single (App "test" 110 200)
                                              , Horizontal [Single (App "foo" 300 100)]])
                              @?= Vertical [ Single (App "test" 1 1)
                                           , Horizontal [Single (App "foo" 1 1)]]
                  ,   testCase "Minimize Horizontal" $
                        minimizeAll (Horizontal [ Single (App "test" 110 200)
                                                ,   Vertical [Single (App "foo" 300 100)]])
                              @?= Horizontal [ Single (App "test" 1 1)
                                             , Vertical [Single (App "foo" 1 1)]]
                  ]
```