# Avaliação 1 de Programação Funcional

## ATENÇÃO

- A interpretação dos enunciados faz parte da avaliação.
- A avaliação deve ser resolvida INDIVIDUALMENTE. Se você discutir soluções com outros alunos da disciplina, deverá mencionar esse fato como parte dos comentários de sua solução.
- 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 modificar a seção Setup inicial do código, seja por incluir bibliotecas ou por eliminar a diretiva de compilação -Wall.
- 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"
        [
          question1Tests
        , question2Tests
        , question3Tests
        , question4Tests
        , question5Tests
        , question6Tests
        , question7Tests
        , question8Tests
        , question9Tests
        , question10Tests
        ]
```

## Uma ferramenta para ensino remoto

Devido a recente pandemia de COVID-19, toda a rede de ensino no Brasil e no mundo teve que se adaptar rapidamente à realidade de aulas remotas.

O objetivo dessa avaliação é a criação de parte de uma ferramenta para elaboração e correção de questionários. Para isso, esta avaliação será dividida em duas partes: a primeira lidará com a modelagem de questionários e a segunda com a sua correção e respectiva elaboração de relatório de notas.

### Parte 1. Validando um questionário

Nesta primeira parte da avaliação, vamos construir tipos de dados e funções para criação de questionários. O tipo central para questionários é o tipo Question:

```
data Question
 = Question [Subject]
```

```
    Statement
    [Choice]
    Value
    deriving Show
```

O tipo Question é composto por um único construtor, Question, que recebe uma lista de tópicos abordados pela questão (tipo Subject), o enunciado da questão (tipo Statement) e uma lista das possíveis respostas para a questão (tipo Choice).

Os tipos Subject e Statement são sinônimos para strings.

```
type Subject = String
type Statement = String
```

O tipo Choice representa um possível item para um questionário e é composto pelo texto da resposta, uma string, e um booleano que indica se este item é a resposta correta para a questão.

```
data Choice
  = Choice String Bool
    deriving Show
```

O tipo Value denota o total de pontos para a questão considerada e é representado como um sinônimo para Int.

```
type Value = Int
```

Representamos um questionário por seu nome (tipo String), o total de pontos distribuídos (tipo Value) e as questões que o compõe.

```
data Questionary
  = Questionary String Value [Question]
    deriving Show
```

As respostas para um determinado estudante são representadas pelo tipo AnswerSet que é composto por um valor de tipo Student, que representa o nome do estudante e uma lista das respostas do aluno. O tipo Answer representa uma resposta para uma questão e consiste de um inteiro indicando a alternativa escolhida (construtor TheAnswer) ou um indicativo que a questão está em branco (construtor Blank).

```
data AnswerSet
  = AnswerSet Student [Answer]
    deriving Show

type Student = String

data Answer
    = TheAnswer Int
    | Blank
    deriving Show
```

Com base no apresentado, faça o que se pede.

*Questão 1.* Dizemos que uma questão é bem formada se essa possui uma e apenas uma resposta (campo booleano do tipo Choice igual a verdadeiro) e uma pontuação não negativa e nem nula. Implemente a função:

```
wfQuestion :: Question -> Bool
wfQuestion = undefined
```

que determina se uma questão é ou não bem formada de acordo com o critério apresentado no enunciado desta questão.

Sua implementação deve satisfazer os seguintes casos de teste.

```
question1Tests :: TestTree
question1Tests
      = testGroup "Question 1"
              [
                 testCase "Test question 1 (success):" $
                              wfQuestion question1 @?= True
              ,  testCase "Test question 1 (failure):" $
                              wfQuestion questionf1 @?= False
              ,  testCase "Test question 1 (failure):" $
                              wfQuestion questionf2 @?= False
              ,  testCase "Test question 1 (failure):" $
                              wfQuestion questionf3 @?= False
              ]

question1 :: Question
question1 = Question ["História"] stmt choices 1
   where
     stmt = "Qual a cor do cavalo branco de Napoleão?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "Azul" False
     ch2 = Choice "Preto" False
     ch3 = Choice "Branco" True

questionf1 :: Question
questionf1 = Question ["História"] stmt choices 1
   where
     stmt = "Qual a cor do cavalo branco de Napoleão?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "Azul" False
     ch2 = Choice "Preto" True
     ch3 = Choice "Branco" True

questionf2 :: Question
questionf2 = Question ["História"] stmt choices 0
   where
     stmt = "Qual a cor do cavalo branco de Napoleão?"
```

```
    choices = [ch1, ch2, ch3]
    ch1 = Choice "Azul" False
    ch2 = Choice "Preto" False
    ch3 = Choice "Branco" True

questionf3 :: Question
questionf3 = Question ["História"] stmt choices (-2)
  where
    stmt = "Qual a cor do cavalo branco de Napoleão?"
    choices = [ch1, ch2, ch3]
    ch1 = Choice "Azul" False
    ch2 = Choice "Preto" False
    ch3 = Choice "Branco" True
```

*Questão 2.* Um questionário é bem formado se todas as suas questões são bem formadas e se o valor total do questionário coincide com a soma das questões que o compõe. Desenvolva a função

```
wfQuestionary :: Questionary -> Bool
wfQuestionary = undefined
```

que determina se um questionário é bem formado ou não.

Sua implementação deve satisfazer os seguintes casos de teste.

```
question2Tests :: TestTree
question2Tests
  = testGroup "Question 2"
       [
          testCase "Test question 2 (success):" $
                   wfQuestionary question2 @?= True
       ,  testCase "Test question 2 (failure):" $
                   wfQuestionary question2a @?= False
       ,  testCase "Test question 2 (failure):" $
                   wfQuestionary question2b @?= False
       ]

question2 :: Questionary
question2
  = Questionary
      "Teste de história"
      6
      [question11, question1]

question2a :: Questionary
question2a
  = Questionary
      "Teste de história"
      6
      [question11, questionf1]
```

```haskell
question2b :: Questionary
question2b
   = Questionary
        "Teste de história"
        6
        [question11]

question11 :: Question
question11 = Question ["História"] stmt choices 5
   where
     stmt = "Qual a cor do cavalo branco de Napoleão?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "Azul" False
     ch2 = Choice "Preto" False
     ch3 = Choice "Branco" True
```

*Questão 3.* Dizemos que as respostas para um estudante são bem formadas se sua solução possui um valor do tipo Answer para cada valor do tipo Question presente em um questionário (tipo Questionary). Implemente a função:

```haskell
wfAnswerSet :: Questionary -> AnswerSet -> Bool
wfAnswerSet = undefined
```

que determina se as respostas de um estudante (valor de tipo AnswerSet) são bem formadas.

Sua implementação deve satisfazer os seguintes casos de teste.

```haskell
question3Tests :: TestTree
question3Tests
      = testGroup "Question 3"
              [
                testCase "Test question 3 (success):" $
                   wfAnswerSet question2 answer11 @?= True
              , testCase "Test question 3 (failure):" $
                   wfAnswerSet question2 answer12 @?= False
              ]

answer11 :: AnswerSet
answer11 = AnswerSet "João da Silva" [TheAnswer 2, TheAnswer 2]

answer12 :: AnswerSet
answer12 = AnswerSet "João da Silva" [TheAnswer 2]
```

*Questão 4.* Uma medida da dificuldade de um teste é a quantidade de respostas em branco presentes em uma avaliação. Desenvolva a função:

```haskell
countBlanks :: AnswerSet -> Int
countBlanks = undefined
```

que conta o número de respostas deixadas em branco em uma avaliação entregue por um aluno. Sua implementação deve satisfazer os seguintes casos de teste.

```
question4Tests :: TestTree
question4Tests
      = testGroup "Question 4"
              [
                testCase "Test question 4 (1)" $ countBlanks answer13 @?= 1
              , testCase "Test question 4 (0)" $ countBlanks answer11 @?= 0
              ]

answer13 :: AnswerSet
answer13 = AnswerSet "João da Silva" [TheAnswer 2, Blank]
```

*Questão 5.* Em bancos de questões, uma funcionalidade importante é a de selecionar questões de um determinado tópico. O banco de questões é representado pelo tipo QuestionDB:

```
type QuestionDB = [Question]
```

Implemente a função

```
selectBySubject :: Subject -> QuestionDB -> [Question]
selectBySubject _
    = foldr step base
      where
        step = undefined
        base = undefined
```

que seleciona todas as questões de um tópico fornecido como entrada. Sua implementação deve ser feita obrigatoriamente usando o template acima, usando foldr. Caso julgue necessário inclua uma variável para representar o primeiro parâmetro de entrada para a função selectBySubject.

Sua implementação deve satisfazer os seguintes casos de teste.

```
questionDB :: QuestionDB
questionDB = [ question1
             , question5]

question5 :: Question
question5 = Question ["Math"] stmt choices 5
   where
     stmt = "Quanto é 2 + 2?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "0" False
     ch2 = Choice "22" False
     ch3 = Choice "4" True

question5Tests :: TestTree
question5Tests
```

```
      = testGroup "Question 5"
                  [
                     testCase "Question 5 (non-empty):" $
                        length (selectBySubject "Math" questionDB) @?= 1
                  ,  testCase "Question 5 (empty) $ " $
                        length (selectBySubject "Geo" questionDB) @?= 0
                  ]
```

## Parte 2. Relatórios

A segunda parte desta avaliação consiste em construir um conjunto de funções para calcular o resultado obtido por alunos e também a impressão de um relatório contendo as notas obtidas no teste.

*Questão 6.* Uma tarefa fundamental de uma aplicação para formular testes é a sua correção. Desenvolva a função

```
grade :: Questionary -> AnswerSet -> Int
grade = undefined
```

que calcula a nota obtida por um aluno ao realizar um teste. A nota é calculada pela soma da pontuação das respostas corretas do aluno.

Sua implementação deve satisfazer os seguintes casos de teste:

```
question6Tests :: TestTree
question6Tests
      = testGroup "Question 6"
              [
                 testCase "Question 6 (success):" $
                      grade question2 answer11 @?= 6
              ,  testCase "Question 6 (blank):" $
                      grade question2 answer12 @?= 5
              ]
```

O seguinte tipo de dados representa os testes resolvidos por uma classe de alunos.

```
data Class
  = Class Questionary [AnswerSet]
    deriving Show
```

Uma importante tarefa deste sistema de elaboração de provas é construir um relatório de notas, que deve conter o nome e a respectiva nota de cada aluno. As próximas questões envolverão a tarefa de construir uma string que representa esse relatório de notas de uma turma.

*Questão 7.* Para construir o relatório de notas, um primeiro passo é calcular as notas de uma turma. Representaremos as notas de uma turma usando o tipo a seguir:

```
type Report = [(Student, Int)]
```

O tipo Report consiste de uma lista de pares formados pelo nome de um estudante e de sua respectiva nota em um teste. Desenvolva a função

```
createReport :: Class -> Report
createReport = undefined
```

que constrói um valor de tipo Report a partir dos resultados da aplicação de um questionário (representado pelo tipo Class).

Sua implementação deve satisfazer os seguintes casos de teste.

```
exampleClass :: Class
exampleClass = Class questionary answers

questionary :: Questionary
questionary = Questionary "Math" 10
                      [
                        questiona
                      , questionb
                      ]

questiona :: Question
questiona =  Question ["Math"] stmt choices 5
   where
     stmt = "Quanto é 2 + 2?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "0" False
     ch2 = Choice "22" False
     ch3 = Choice "4" True

questionb :: Question
questionb =  Question ["Math"] stmt choices 5
   where
     stmt = "Quanto é 1 + 1?"
     choices = [ch1, ch2, ch3]
     ch1 = Choice "0" False
     ch2 = Choice "11" False
     ch3 = Choice "2" True

answerj :: AnswerSet
answerj = AnswerSet "João da Silva" [TheAnswer 1, Blank]

answerm :: AnswerSet
answerm = AnswerSet "Ana Maria" [TheAnswer 2, TheAnswer 2]

answers :: [AnswerSet]
answers = [answerj, answerm]

question7Tests :: TestTree
question7Tests
      = testGroup "Question 7"
```

```
            [
               testCase "Question 7 result" $
                    createReport exampleClass @?= [("João da Silva", 0),
                                                   ("Ana Maria" , 10)]
            ]
```

As próximas questões envolvem a criação de uma string que representa o relatório de notas de uma turma. Um exemplo de tal relatório é apresentado a seguir.

```
+------------+------+
| Nome       | Nota |
+------------+------+
| João Silva | 0    |
+------------+------+
| Ana Maria  | 10   |
+------------+------+
```

*Questão 8.* Observe que para construir esse relatório é necessário calcular o tamanho das colunas referentes ao nome do aluno e sua respectiva nota. Desenvolva a função

```
type ColumnSize = (Int, Int)

columnsSize :: Report -> ColumnSize
columnsSize = undefined
```

que retorna um par correspondente ao tamanho das duas colunas da tabela de notas.

A sua implementação deve satisfazer o seguinte caso de teste:

```
rep :: Report
rep = [("João da Silva", 0), ("Ana Maria" , 10)]

question8Tests :: TestTree
question8Tests
      = testGroup "Question 8"
            [
               testCase "Question 8 result:" $
                    columnsSize rep @?= (13, 4)
            ]
```

*Questão 9.* De posse do tamanho das colunas e da informação de nota de um aluno, podemos gerar a string correspondente a uma linha do relatório. Desenvolva a função

```
lineReport :: (Student, Int) -> ColumnSize -> String
lineReport = undefined
```

que irá produzir a string correspondente a uma linha da tabela de notas.

A sua implementação deve satisfazer o seguinte caso de teste:

```haskell
question9Tests :: TestTree
question9Tests
      = testGroup "Question 9"
              [
                testCase "Question 9 result:" $
                  lineReport ("João da Silva", 0) (13, 4) @?= lineRep
              ]

lineRep :: String
lineRep = "| João Silva | 0    |"
```

*Questão 10.* Utilizando as funções anteriores, desenvolva a função

```haskell
printReport :: Class -> String
printReport = undefined
```

que produz a string correspondente ao relatório de notas das avaliações de uma turma representada por um valor de tipo Class.

Sua implementação deve satisfazer o seguinte caso de teste:

```haskell
question10Tests :: TestTree
question10Tests
      = testGroup "Question 10"
              [
                testCase "Question 10 result:" $
                   printReport exampleClass @?= tableRep
              ]

tableRep :: String
tableRep
 = concat ["+------------+------+\n"
          ,"| Nome       | Nota |\n"
          ,"+------------+------+\n"
          ,"| João Silva | 0    |\n"
          ,"+------------+------+\n"
          ,"| Ana Maria  | 10   |\n"
          ,"+------------+------+\n"]
```