Avaliação 2 de BCC222 - Programação Funcional

ATENÇÃO

- A interpretação dos enunciados faz parte da avaliação.
- A avaliação deve ser resolvida INDIVIDUALMENTE. Não serão tolerados plágios de nenhum tipo.
- 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.
- Seu código deve ser compilado sem erros de compilação. A presença de erros acarretará em uma penalidade de 10% para cada erro de compilação. 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.

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 2" 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.
6. Não será aceito o envio de soluções em formato ".hs". Avaliações enviadas nesse formato não serão consideradas para correção.

Setup Inicial

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances    #-}

module Main where
```

```haskell
import Data.Char
import Data.List
import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck as QC hiding (Success, Failure, listOf)

main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Test suite"
            [ exercise1Tests
            , exercise2Tests
            , exercise3Tests
            ]
```

Manipulação de Horas

Considere a tarefa de implementar a lógica de validação de uma aplicação para registrar pontos visitados por turistas em uma cidade. A aplicação registra informações sobre a hora em que turistas visitam uma certa atração.

O tipo Time representa informações sobre a hora de uma visita.

```haskell
data Time
  = Time {
      hour   :: Int
    , minute :: Int
    } deriving (Eq, Ord)
```

Por sua vez, o tipo Register armazena uma lista de informações de horas.

```haskell
data Register = Register [Time]
                deriving (Eq, Ord)
```

Com base no apresentado, faça o que se pede.

Exercício 1: Classes de tipos

O objetivo deste exercício é a construção de instâncias da classe Show para os tipos apresentados.

a) (Valor 1,0 pt.) Desenvolva uma instância de Show para o tipo Time:

```haskell
instance Show Time where
   show = undefined

testShowTime :: TestTree
testShowTime
  = testGroup "Teste para Show Time."
```

```
        [
          testCase "Instância de Show para Time" $
               show (Time 10 43) @?= "10:43"
        ]
```

b) (Valor 1,0 pt.) Desenvolva uma instância de Show para o tipo Register de forma que cada valor de hora na string produzida seja separado por um ";"

```
instance Show Register where
   show (Register ts) = undefined
```

Sua implementação deve satisfazer os testes a seguir.

```
testShowRegister :: TestTree
testShowRegister
  = testGroup "Teste para Show Register"
       [
          testCase "Empty" $ show (Register []) @?= ""
       ,  testCase "One"   $ show (Register [t1]) @?= "10:10"
       ,  testCase "Two"   $ show (Register [t1,t2]) @?= "10:10;20:20"
       ]
       where
         t1 = Time 10 10
         t2 = Time 20 20

exercise1Tests :: TestTree
exercise1Tests
  = testGroup "Exercício 1."
       [
         testShowTime
       , testShowRegister
       ]
```

Exercício 2. Parsing

O objetivo deste exercício é construir um parser para uma lista de informações de hora.

a) (Valor 2,0 pts.) Desenvolva um parser para informações sobre a hora de visita de um ponto turístico:

```
parseTime :: Parser Char (Int, Int)
parseTime = undefined
```

Note que seu parser deve retornar um par de números inteiros, em que o primeiro componente do par representa a informação sobre horas e o segundo a informação sobre minutos. Seu parser deve atender os seguintes testes:

```
testParseTime :: TestTree
testParseTime
```

```
  = testGroup "Test parseTime"
      [
        testCase "parseTime Ok" $ runParser parseTime "10:45"
            @?= [((10,45),"")]
      , testCase "parseTime failure" $ runParser parseTime "10"
            @?= []
      , QC.testProperty "parseTime/show Time" $
            \ (h,m) -> let h' = abs h
                           m' = abs m
                           s  = show (Time h' m')
                       in runParser parseTime s == [((h',m'), "")]
      ]
```

b) (Valor 2,0 pts.) Desenvolva um parser para processar uma lista de informações de horas separadas por ";".

```
parseRegister :: Parser Char [(Int,Int)]
parseRegister = undefined
```

Sua implementação deve atender os seguintes casos de teste:

```
testParseRegister :: TestTree
testParseRegister
  = testGroup "Teste para parseRegister"
      [
        testCase "One" $ runParser parseRegister "10:10" @?= [([(10,10)],"")]
      , testCase "Three" $ runParser parseRegister "10:10;20:20;15:39"
          @?= [([(10,10), (20,20),(15,39)],"")]
      ]

exercise2Tests :: TestTree
exercise2Tests
  = testGroup "Exercício 2"
      [
         testParseTime
      ,  testParseRegister
      ]
```

Exercício 3. Validação

O objetivo desta questão é realizar a validação de informação produzida pelos parsers construídos na questão 2. Para realizar a validação, vamos utilizar applicative functors.

a) (Valor 1,0 pt.) Considere o tipo TimeError que representa os possíveis erros de validação de dados de hora.

```
data TimeError
  = InvalidHour Int
```

```
  | InvalidMinute Int
  deriving (Eq, Show)
```

O construtor InvalidHour representa inteiros que não estão no intervalo válido para valores de hora: [0,23].

Com base no apresentado, desenvolva a função checkHour:

```
checkHour :: Int -> Validation [TimeError] Int
checkHour = undefined
```

que realiza a validação de um inteiro para o valor de horas. Sua implementação deve satisfazer os seguintes casos de testes:

```
testCheckHour :: TestTree
testCheckHour
  = testGroup "test checkHour"
       [
         testCase "checkHour ok" $ checkHour 20 @?= Success 20
       , testCase "checkHour fail 1" $ checkHour 40 @?= Failure [InvalidHour 40]
       , testCase "checkHour fail 2" $ checkHour (-10) @?= Failure [InvalidHour (-10)]
       ]
```

b) (Valor 1,0 pt.) Implemente a função

```
checkMinute :: Int -> Validation [TimeError] Int
checkMinute  = undefined
```

que realiza a validação da informação sobre minutos. Consideramos que um inteiro representa um valor válido de minutos se este está no intervalo [0,59]. Sua implementação deverá atender os seguintes testes:

```
testCheckMinute :: TestTree
testCheckMinute
  = testGroup "test checkMinute"
       [
         testCase "checkMinute ok" $ checkMinute 20 @?= Success 20
       , testCase "checkMinute fail 1" $ checkMinute 70 @?= Failure [InvalidMinute 70]
       , testCase "checkHour fail 2" $ checkMinute (-10) @?= Failure [InvalidMinute (-10)]
       ]
```

c) (Valor 1,0 pt.) Combinando as funções checkHour e checkMinute, implemente a função

```
checkTime :: (Int, Int) -> Validation [TimeError] Time
checkTime = undefined
```

que valida se uma determinada informação de hora é válida. Sua implementação deve satisfazer os seguintes testes:

```
testCheckTime :: TestTree
testCheckTime
```

```
  = testGroup "Função checkTime"
      [
        testCase "Hora inválida" $ checkTime (50,13)
           @?= Failure [InvalidHour 50]
      , testCase "Minuto inválido" $ checkTime (12,80)
           @?= Failure [InvalidMinute 80]
      , testCase "Hora e minuto inválidos" $ checkTime (50,80)
           @?= Failure [InvalidHour 50, InvalidMinute 80]
      , testCase "Hora e minutos válidos" $ checkTime (1,38)
           @?= Success (Time 1 38)
      ]
```

d) (Valor 1,0 pt.) Utilizando a função para validação de hora, implemente a validação de uma lista destes valores. Sua função deve utilizar checkTime e coletar todos os erros encontrados na lista de dados de horário fornecida como argumento.

```
checkRegister :: [(Int, Int)] -> Validation [TimeError] Register
checkRegister = undefined
```

Sua implementação deve atender os seguintes casos de teste:

```
testCheckRegister :: TestTree
testCheckRegister
  = testGroup "Test checkRegister"
      [
        testCase "Empty" $ checkRegister [] @?=
          (Success (Register []))
      , testCase "One"   $ checkRegister [(50,80)] @?=
          (Failure [InvalidHour 50, InvalidMinute 80])
      , testCase "Two" $ checkRegister [(10,10),(15,30)] @?=
          (Success $ Register [Time 10 10, Time 15 30])
      ]

exercise3Tests :: TestTree
exercise3Tests
  = testGroup "Exercício 3"
       [
         testCheckHour
       , testCheckMinute
       , testCheckTime
       , testCheckRegister
       ]
```

Apêndice A: Biblioteca de parsing

- Definição do tipo de parsing

```haskell
newtype Parser s a
     = Parser {
         runParser :: [s] -> [(a,[s])]
       }
```

- O tipo Parser é um Functor

```haskell
instance Functor (Parser s) where
   fmap f (Parser p)
      = Parser (\ inp -> [(f x, xs) | (x,xs) <- p inp])
```

- O tipo Parser é um Applicative Functor

```haskell
instance Applicative (Parser s) where
   pure = succeed
   (Parser p) <*> (Parser q)
      = Parser (\ inp -> [(f x, xs) | (f, ys) <- p inp
                                    , (x, xs) <- q ys])
```

- Realizando o parsing de um símbolo no início da entrada.

```haskell
symbol :: Eq s => s -> Parser s s
symbol s = Parser (\ inp -> case inp of
                              [] -> []
                              (x : xs) -> if x == s
                                          then [(s,xs)]
                                          else [])
```

- Realizando o parsing de uma string que é prefixo da entrada.

```haskell
token :: Eq s => [s] -> Parser s [s]
token s
  = Parser (\ inp -> if s == (take n inp)
                     then [(s, drop n inp)]
                     else [])
    where
      n = length s
```

- Realizando o parsing do primeiro símbolo da entrada que satisfaz uma condição.

```haskell
sat :: (s -> Bool) -> Parser s s
sat p = Parser (\ inp -> case inp of
                           [] -> []
                           (x : xs) -> if p x
                                       then [(x,xs)]
                                       else [])
```

- Realizando o parsing de um dígito,retornando-o como um caractere.

```haskell
digitChar :: Parser Char Char
digitChar = sat isDigit
```

- Realizando o parsing de um dígito, retornando-o como um inteiro.

```
digit :: Parser Char Int
digit = f <$> digitChar
        where
          f c = ord c - ord '0'
```

- Parsing que não consume nenhum símbolo da entrada, retornando um valor padrão, fornecido como argumento, como resultado.

```
succeed :: a -> Parser s a
succeed v = Parser (\ inp -> [(v,inp)])
```

- Parser que sempre falha.

```
failure :: Parser s a
failure = Parser (\ _ -> [])
```

- Escolha entre dois parsers.

```
infixr 4 <|>

(<|>) :: Parser s a -> Parser s a -> Parser s a
(Parser p) <|> (Parser q)
   = Parser (\ inp -> p inp ++ q inp)
```

- Execução opcional de um parser.

```
option :: Parser s a -> a -> Parser s a
option p d = p <|> succeed d
```

- Repetindo um parser zero ou mais vezes.

```
many :: Parser s a -> Parser s [a]
many p = ((:) <$> p <*> many p) <|> succeed []
```

- Repetindo um parser uma ou mais vezes

```
many1 :: Parser s a -> Parser s [a]
many1 p = (:) <$> p <*> many p
```

- Parsing de um número natural

```
natural :: Parser Char Int
natural = foldl f 0 <$> greedy digit
    where
      f ac d = ac * 10 + d
```

- Descartando os resultados intermediários de um parser.

```
first :: Parser s a -> Parser s a
first (Parser p)
  = Parser (\ inp -> let r = p inp
                     in if null r then []
                        else [head r])
```

- Repetindo um parser zero ou mais vezes, descartando resultados intermediários.

```
greedy :: Parser s a -> Parser s [a]
greedy = first . many
```

- Repetindo um parser uma ou mais vezes, descartando resultados intermediários.

```
greedy1 :: Parser s a -> Parser s [a]
greedy1 = first . many1
```

- Realizando o parsing de um identificador.

```
identifier :: Parser Char String
identifier
   = (:) <$> letter <*> greedy (sat isAlphaNum)
     where
       letter = sat isLetter
```

- Realizando o parsing de uma lista com separadores entre elementos.

```
listOf :: Parser s a -> Parser s b -> Parser s [a]
listOf p sep
  = (:) <$> p <*> many ((\ x y -> y) <$> sep <*> p)
```

- Realizando o parsing de valores entre dois separadores, especificados como argumentos.

```
pack :: Parser s a -> Parser s b ->
        Parser s c -> Parser s b
pack p q r = (\ _ x _ -> x) <$> p <*> q <*> r
```

- Realizando o parsing de um valor entre parêntesis.

```
parenthesized :: Parser Char a -> Parser Char a
parenthesized p = pack (symbol '(') p (symbol ')')
```

- Realizando o parsing de uma lista em que separadores vem ao final de um item.

```
endBy :: Parser s a -> Parser s b -> Parser s [a]
endBy p sep = greedy ((\ x _ -> x) <$> p <*> sep)
```

Apêndice B: Tipo Validation

- Definição do tipo Validation.

```
data Validation err a
  = Success a
  | Failure err
  deriving (Eq, Ord, Show)
```

- Validation é um Functor

```haskell
instance Functor (Validation err) where
  fmap f (Success x) = Success (f x)
  fmap _ (Failure y) = Failure y
```

- Validation é um Applicative Functor

```haskell
instance Semigroup err => Applicative (Validation err) where
  pure = Success
  (Failure x) <*> (Failure y) = Failure (x <> y)
  (Failure x) <*> _           = Failure x
  _           <*> (Failure y) = Failure y
  (Success f) <*> (Success x) = Success (f x)
```