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
        ]
```

Introdução

Considere o seguinte tipo de dados que modela uma locomotiva que pode transportar diferentes tipos de produtos.

```
data Train a
   = Empty
   | Wagon a (Train a)
   | Locomotive Weight (Train a)
   deriving (Eq, Ord, Show)

type Weight = Int
```

O construtor `Empty` é usado para indicar o fim de uma composição de locomotiva. Por sua vez, o construtor `Wagon` representa um vagão que armazena elementos de um tipo `a` e é ligado a um valor de tipo `Train a`, que representa a continuação da composição ferroviária. Finalmente, o construtor `Locomotive` denota uma locomotiva que capaz de transportar um peso (representado pelo tipo `Weight`) e uma sequência de valores de tipo `Train a`.

Locomotivas podem transportar pessoas e cargas. O tipo de dados `Cargo` modela os diferentes tipos de carga que pode ser armazenada em um trem. O constructor `NoCargo` indica que um elemento está vazio (não leva pessoas ou produtos). Um vagão de pessoas é representado pelo construtor `Persons`, que armazena uma

lista dos pesos das pessoas nele contido. Finalmente, o constructor `Products` armazena o peso total de carga armazenada.

```
data Cargo
    = NoCargo
    | Persons [Weight]
    | Products Weight
    deriving (Eq, Ord, Show)
```

Como exemplo de um valor destes tipos, considere:

```
aTrain :: Train Cargo
aTrain = Locomotive 1000 w1

w1 :: Train Cargo
w1 = Wagon (Products 100) w2

w2 :: Train Cargo
w2 = Wagon (Persons [70, 90, 110, 60]) w3

w3 :: Train Cargo
w3 = Wagon (Products 200) Empty
```

Dizemos que um valor de tipo `Train a` é válido se as seguintes condições são verdadeiras:

1. O primeiro construtor de um valor do tipo `Train a` deve ser `Locomotive`.
2. O último construtor de um valor do tipo `Train a` deve ser Empty.
3. A soma de pesos transportados por um comboio representado por um valor de tipo `Train a` deve ser menor que o peso máximo suportado pela locomotivas que "puxam" a composição.

Com base nas 3 condições, podemos concluir que o valor `aTrain` é válido. Porém, o valor

```
wrong1 :: Train Cargo
wrong1 = Locomotive 100 (Wagon (Products 50) Empty)
```

Não é válido, pois está transportando uma carga com peso superior ao suportado por uma locomotiva.

Composições podem ser formadas por mais de uma locomotiva, como o exemplo a seguir:

```
sample :: Train Cargo
sample = Locomotive 100 (Wagon (Products 90)
         (Locomotive 90 (Wagon (Products 100) Empty)))
```

O valor `sample` é considerado válido por iniciar por atender todas as restrições. Observe que a segunda locomotiva (que suporta uma carga de peso 90) adiciona potência adicional ao comboio para transportar um vagão de peso 100.

De posse da descrição acima, resolva os exercícios a seguir.

Exercícios

1. (Valor 2,0 pts). A função `foldr` para listas é definida como:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr f v (x : xs) = f x (foldr f v xs)
```

De maneira intuitiva, `foldr f v xs` substitui a lista vazia pelo valor `v` e o construtor `:` pela função `f`. Podemos definir uma função `fold` para o tipo `Train a`. A chamada `foldTrain f v t` para `t :: Train a`, remove o construtor `Locomotive`, substitui o construtor `Wagon` pela função `f` e `Empty` por `v`.

Com base no apresentado, implemente a função `foldTrain` para o tipo `Train`.

```
foldTrain :: (a -> b -> b) -> b -> Train a -> b
foldTrain = undefined
```

Sua implementação deve atender os seguintes casos de teste:

```
question1Tests :: TestTree
question1Tests
  = testGroup "Question 1"
        [
           testCase "Test 1" $ foldTrain g [] aTrain @?= [70, 90, 110, 60]
        ,  testCase "Test 2" $ foldTrain g [] sample @?= []
        ]
    where
      g (Persons ps) ac = ps ++ ac
      g _ ac = ac
```

2. (Valor 2,0 pts). A função `filter` para listas é definida como:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x : xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

De maneira intuitiva, `filter p xs` retorna uma lista em que todos os elementos satisfazem a condição expressa pelo predicado `p :: a -> Bool`. Podemos definir uma função `filter` para o tipo `Train a`. A chamada `filterTrain p t` retorna um valor do tipo `Train` em que todos os vagões (construtor `Wagon`) satisfazem a condição expressa pelo predicado `p`.

Com base no apresentado, implemente a função `filterTrain`.

```
filterTrain :: (a -> Bool) -> Train a -> Train a
filterTrain = undefined
```

Sua implementação deve atender os seguintes casos de teste:

```
question2Tests :: TestTree
question2Tests
  = testGroup "Question 2"
        [
          testCase "Test 1" $ filterTrain p aTrain @?= aTrain'
        , testCase "Test 2" $ filterTrain p sample @?= sample
        ]
  where
    p (Products _) = True
    p _            = False
    aTrain' =  Locomotive 1000 w1'
    w1' = Wagon (Products 100) w3
```

3. (Valor 2,0 pts). Termine a implementação da função

```
weight :: Train Cargo -> Weight
weight = foldTrain step base
  where
    step = undefined
    base = undefined
```

que calcula o peso total transportado por um comboio ferroviário representado por um valor de tipo `Train Cargo`. Sua solução deve apenas prover a implementação das funções `step` e `base` de maneira apropriada.

Sua implementação deve atender os seguintes casos de teste:

```
question3Tests :: TestTree
question3Tests
  = testGroup "Question 3"
        [
          testCase "Test 1" $ weight aTrain @?= 630
        , testCase "Test 2" $ weight sample @?= 190
        ]
```

4. (Valor 2,0 pts). A função `map` para listas é definida como:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

De maneira intuitiva, `map f xs` aplica a função `f` a cada um dos elementos da lista `xs`. Podemos definir a função `map` para valores do tipo `Train` de forma similar. A chamada `mapTrain f t` aplicará a função `f` a cada um dos valores de tipo `a` armazenados pelo construtor `Wagon` em um valor de tipo `Train a`.

Com base no apresentado, implemente a função `mapTrain`:

```haskell
mapTrain :: (a -> b) -> Train a -> Train b
mapTrain = undefined
```

Sua implementação deve atender os seguintes casos de teste:

```haskell
question4Tests :: TestTree
question4Tests
  = testGroup "Question 4"
        [
          testCase "Test 1" $ mapTrain f aTrain @?= aTrain'
        , testCase "Test 2" $ mapTrain f sample @?= sample
        ]
        where
          f (Persons _) = Persons []
          f x           = x
          aTrain' =  Locomotive 1000 w1'
          w1' = Wagon (Products 100) w2'
          w2' = Wagon (Persons []) w3
```

5. (Valor 2,0 pts). Implemente a função:

```haskell
buildTrain :: [Cargo] -> Train Cargo
buildTrain = undefined
```

Que a partir de uma lista de itens a serem transportados, retorna um valor do tipo `Train` que representa o comboio ferroviário que transporta toda a carga presente na lista fornecida como segundo argumento. É importante notar que o comboio deve iniciar com uma locomotiva que possui como peso máximo suportado o valor da soma do peso de todos os itens levados em seus vagões.

Sua implementação deve atender os seguintes casos de teste:

```haskell
itens :: [Cargo]
itens = [Persons [10], Products 2000, Products 1000]

question5Tests :: TestTree
question5Tests
  = testGroup "Question 5"
        [
          testCase "Test 1" $ buildTrain itens @?= train1
        ]
        where
          train1 = Locomotive 3010 w1'
          w1' = Wagon (Persons [10]) w2'
          w2' = Wagon (Products 2000) w3'
          w3'  = Wagon (Products 1000) Empty
```