Avaliação 3 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.
- Essa avaliação possui vários casos de teste para cada exercício. Grande parte do texto desse enunciado, consiste de código de testes dos exercícios propostos nesta avaliação.
- 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 3" 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 ParseLib

import Test.Tasty
import Test.Tasty.HUnit
```

Parsing de configurações

Introdução

Um elemento muito importante em um sistema de software é o armazenamento de informações. Existem diversos formas de resolver esse problema: usando um sistema gerenciador de banco de dados ou mesmo usando um formato de arquivo específico.

O objetivo desta avaliação é o desenvolvimento de um algoritmo para processamento de uma versão simplificada de um formato de textual para configurações. A seguir, apresentamos um exemplo deste formato de arquivo:

```
[pedidos]
produto = pizza
quantidade = 20

[bebidas]
cerveja = pilsen
pagou = False
```

Um arquivo é formado por uma ou mais configurações. Cada configuração é formada por um identificador entre colchetes e uma sequência de definições. Por sua vez, definições são compostas por um identificador, o símbolo de igual e um valor. Valores podem ser strings quaisquer, números inteiros positivos ou valores booleanos (True ou False).

Representação do formato

Representaremos cada um dos elementos de um arquivo de configuração usando tipos Haskell. Identificadores serão presentados pelo tipo `Id`, que armazena uma string:

```
newtype Id
      = Id { ident :: String }
        deriving (Eq, Ord)
```

Como exemplos de identificadores, considere os seguintes valores.

```haskell
pedidos :: Id
pedidos = Id "pedidos"

produto :: Id
produto = Id "produto"

quantidade :: Id
quantidade = Id "quantidade"

cerveja :: Id
cerveja = Id "cerveja"

pagou :: Id
pagou = Id "pagou"

bebidas :: Id
bebidas = Id "bebidas"
```

Valores serão representados pelo seguinte tipo de dados que possui um construtor para cada categoria de valores aceitos.

```haskell
data Value
      = VBool Bool
      | VInt  Int
      | VString String
      deriving (Eq, Ord)
```

A seguir, apresentamos exemplos de valores:

```haskell
pizza :: Value
pizza = VString "pizza"

pilsen :: Value
pilsen = VString "pilsen"

twenty :: Value
twenty = VInt 20

false :: Value
false = VBool False
```

Usando tipos para identificadores e valores, podemos declarar o tipo de definições como:

```haskell
data Definition
      = Def Id Value
        deriving (Eq, Ord)
```

Usando o tipo `Definition`, podemos representar as equações

```
produto = pizza
quantidade = 20
cerveja = pilsen
pagou = False
```

como

```
defpizza :: Definition
defpizza = Def produto pizza

defquant :: Definition
defquant = Def quantidade twenty

defcerv :: Definition
defcerv = Def cerveja pilsen

defpagou :: Definition
defpagou = Def pagou false
```

Configurações são representadas pelo tipo `Config` que armazena o nome da configuração e uma sequência de definições.

```
data Config
  = Config {
      name :: Id
    , definitions :: [Definition]
    } deriving (Eq, Ord)
```

A configuração abaixo

```
[pedidos]
produto = pizza
quantidade = 20
```

é representada pelo seguinte valor do tipo `Config`:

```
config1 :: Config
config1 = Config pedidos [defpizza, defquant]
```

e a configuração

```
[bebidas]
cerveja = pilsen
pagou = False

config2 :: Config
config2 = Config bebidas [defcerv, defpagou]
```

Finalmente, um arquivo de configurações é representado pelo tipo `CfgData` que armazena uma lista de valores do tipo `Config`:

```
newtype CfgData
      = CfgData [Config]
        deriving (Eq, Ord)
```

O arquivo de exemplo apresentado é representado pelo seguinte valor:

```
cfgData :: CfgData
cfgData = CfgData [config1, config2]
```

Considerando o apresentado, desenvolva o que se pede.

Parte 1. Visualizando a estrutura de dados

As primeiras questões dessa avaliação versam sobre o problema de construir instâncias da classe `Show` para os tipos que representam o formato de configurações.

Questão 1. (Valor 1,0 pt). Implemente uma instância de `Show` para o tipo `Id` de forma que o resultado de `show` seja a string associada a um identificador.

```
instance Show Id where
  show = undefined
```

Os seguintes casos de teste devem satisfeitos por sua implementação.

```
showIdTests :: TestTree
showIdTests
  = testGroup "Show Id tests"
              [
                testCase "produto" $ show produto @?= "produto"
              , testCase "pedidos" $ show pedidos @?= "pedidos"
              ]
```

Todos os testes presentes nessa avaliação utilizam o operador `@?=` que verifica se o resultado de uma chamada de função (lado esquerdo do operador) é igual ao valor esperado (lado direito). Portanto, a expressão

```
show produto @?= "produto"
```

será um teste executado com sucesso se o resultado de `show produto` for igual a string "produto".

Questão 2. (Valor 1,0 pt). Implemente uma instância de `Show` para o tipo `Value` de forma que os seguintes casos de teste sejam satisfeitos por sua implementação.

```
instance Show Value where
  show = undefined

showValueTests :: TestTree
showValueTests
  = testGroup "Show Value tests"
              [
                testCase "False" $ show false @?= "False"
              , testCase "20" $ show twenty @?= "20"
              , testCase "pilsen" $ show pilsen @?= "pilsen"
              ]
```

Questão 3. (Valor 1,0 pt). Implemente uma instância de `Show` para o tipo `Definition` de forma que os seguintes casos de teste sejam satisfeitos por sua implementação.

```haskell
instance Show Definition where
   show = undefined

showDefinitionTests :: TestTree
showDefinitionTests
   = testGroup "Show Definition tests"
              [
                testCase "Defpizza" $ show defpizza @?= "produto = pizza"
              , testCase "Defquantidade" $ show defquant @?= "quantidade = 20"
              ]
```

Questão 4. (Valor 1,0 pt). Implemente uma instância de `Show` para o tipo `Config` de forma que os seguintes casos de teste sejam satisfeitos por sua implementação.

```haskell
instance Show Config where
   show = undefined

showConfigTests :: TestTree
showConfigTests
   = testGroup "Show Config tests"
              [
                testCase "config1" $ show config1 @?= unlines list1
              , testCase "config2" $ show config2 @?= unlines list2
              ]
     where
       list1 = ["[pedidos]", "produto = pizza", "quantidade = 20"]
       list2 = ["[bebidas]", "cerveja = pilsen", "pagou = False"]
```

Questão 5. (Valor 1,0 pt). Implemente uma instância de `Show` para o tipo `CfgData` de forma que os seguintes casos de teste sejam satisfeitos por sua implementação.

```haskell
instance Show CfgData where
   show = undefined

showCfgDataTest :: TestTree
showCfgDataTest
   = testGroup "Show CfgData test"
              [
                testCase "cfgData" $ show cfgData @?= cf1 ++ "\n" ++ cf2 ++ "\n"
              ]
     where
       list1 = ["[pedidos]", "produto = pizza", "quantidade = 20"]
       list2 = ["[bebidas]", "cerveja = pilsen", "pagou = False"]
       cf1 = unlines list1
       cf2 = unlines list2
```

Parte 2. Construindo o parser de configurações

A segunda parte desta avaliação envolve a criação de um parser para o formato de configurações. Pode ser conveniente a eliminação de espaços em branco presentes entre diferentes componentes do formato. Para isso, utilize a função

```
strip :: Parser Char a -> Parser Char a
strip p = whitespace *> p
```

que executa um parser `p` após descartar espaços em branco presentes no início da entrada.

Questão 6. (Valor 1,0 pt). Implemente um parser para identificadores do formato considerado (representados pelo tipo Id). *Dica:* utilize o parser `identifier` da biblioteca de parsing.

```
idParser :: Parser Char Id
idParser = undefined
```

Seu parser de identificadores deve satisfazer os seguintes testes.

```
parserIdTests :: TestTree
parserIdTests
   = testGroup "Parser Id tests"
               [
                 testCase "produto" $ r1 @?= r1'
               , testCase "pedidos" $ r2 @?= r2'
               ]
      where
        r1 = runParser idParser "produto"
        r1' = runParser (succeed produto) ""
        r2 = runParser idParser "pedidos"
        r2' = runParser (succeed pedidos) ""
```

Questão 7. (Valor 1,0 pt). Implemente um parser para valores (tipo `Value`). *Dica:* Construa parsers para cada um dos tipos de valores: inteiros, booleanos e strings e combine-os para formar o parser de valores.

```
valueParser :: Parser Char Value
valueParser
  = undefined
```

Seu parser deverá atender os seguintes casos de teste.

```
valueParserTests :: TestTree
valueParserTests
  = testGroup "Parser Value tests"
      [
        testCase "Int" $ r1 @?= r1'
      , testCase "Bool" $ r2 @?= r2'
      , testCase "String" $ r3 @?= r3'
```

```
    ]
    where
      r1 = head $ runParser valueParser "20"
      r1' = head $ runParser (succeed twenty) ""
      r2 = head $ runParser valueParser "False"
      r2' = head $ runParser (succeed false) ""
      r3 = head $ runParser valueParser "pilsen"
      r3' = head $ runParser (succeed pilsen) ""
```

Questão 8. (Valor 1,0 pt). Implemente um parser para definições (tipo `Definition`). Uma definição consiste de um identificador, seguido de um símbolo de igualdade e de um valor.

```
definitionParser :: Parser Char Definition
definitionParser
   = undefined
```

Sua definição deve atender os casos de teste a seguir.

```
definitionParserTests :: TestTree
definitionParserTests
   = testGroup "Parser Definition tests"
       [
         testCase "defquant" $ r1 @?= defquant
       , testCase "defcerv" $ r2 @?= defcerv
       , testCase "defpagou" $ r3 @?= defpagou
       ]
     where
       g = fst . head
       r1 = g (runParser definitionParser (show defquant))
       r2 = g (runParser definitionParser (show defcerv))
       r3 = g (runParser definitionParser (show defpagou))
```

Questão 9. (Valor 1,0 pt). Implemente um parser para um grupo de configurações (tipo `Config`). Um grupo de configurações inicia com um identificador entre colchetes e é seguido por equações separadas por quebra de linha.

a) (Valor 0,5 pt). O primeiro componente de um grupo de configurações é o cabeçalho. Construa um parser que processa um identificador entre colchetes e retorna o valor do tipo `Id` correspondente ao identificador.

```
header :: Parser Char Id
header
   = undefined
```

Seu parser deve satisfazer os seguintes testes.

```
headerParserTests :: TestTree
headerParserTests
  = testGroup "Parser Header tests"
```

```
    [
       testCase "[pedidos]" $ r1 @?= pedidos
    ,  testCase "[bebidas]" $ r2 @?= bebidas
    ]
    where
      g = fst . head
      r1 = g (runParser header "[pedidos]")
      r2 = g (runParser header "[bebidas]")
```

b) (Valor 0,5 pt). O próximo passo é construir um parser para processar um conjunto de equações e retorná-las em uma lista. Para isso, implemente o parser `body` que processa as equações de uma configuração, separadas por quebras de linha. *Dica:* para implementar esse parser, utilize a função *endBy* presente na biblioteca de parsing desenvolvida em classe.

```
body :: Parser Char [Definition]
body = undefined
```

Seu parser deve atender os seguintes casos de teste.

```
bodyParserTests :: TestTree
bodyParserTests
   = testGroup "Parser body tests"
       [
         testCase "list 1" $ r1 @?= [defpizza, defquant]
       , testCase "list 2" $ r2 @?= [defcerv, defpagou]
       ]
     where
       g = fst . head
       r1 = g (runParser body (unlines list1))
       r2 = g (runParser body (unlines list2))
       list1 = ["produto = pizza", "quantidade = 20"]
       list2 = ["cerveja = pilsen", "pagou = False"]
```

Finalmente, os parsers desenvolvidos por você são combinados para formar o parser de um grupo de configurações.

```
configParser :: Parser Char Config
configParser
      = Config <$> header <*> body
```

Questão 10. (Valor 1,0 pt). Um arquivo de configurações consiste de uma sequência de configurações. Implemente o parser:

```
dataParser :: Parser Char CfgData
dataParser = undefined
```

que produz a estrutura de dados correspondente a um arquivo de configurações. Seu parser deve satisfazer o seguinte caso de teste.

```haskell
dataParserTests :: TestTree
dataParserTests
    = testGroup "Parser dataParser tests"
        [
          testCase "cfgData" $ r1 @?= cfgData
        ]
        where
          g = fst . head
          r1 = g (runParser dataParser (show cfgData))
```

O restante do código é destinado a executar a bateria de testes fornecida como parte desta avaliação. Para executar os testes basta usar o seguinte comando no terminal:

```
$> stack build
$> stack exec prova3-exe
```

Função principal para execução dos testes

```haskell
tests :: TestTree
tests
 = testGroup "tests"
      [
        showIdTests
      , showValueTests
      , showDefinitionTests
      , showConfigTests
      , showCfgDataTest
      , parserIdTests
      , valueParserTests
      , definitionParserTests
      , headerParserTests
      , bodyParserTests
      , dataParserTests
      ]
```

Função main.

```haskell
main :: IO ()
main = defaultMain tests
```