Avaliação 2 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 prova02-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 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.

Setup inicial

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

module Main where

import ParseLib
import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain tests

tests :: TestTree
tests
  = testGroup "Unit tests"
        [
           question01aTests
        ,  question01bTests
        ,  question02aTests
        ,  question02bTests
        ,  question03aTests
        ,  question03bTests
        ]

parse :: Parser s a -> [s] -> Maybe a
parse p s = if null res then Nothing
            else Just $ fst (head res)
     where
       res = runParser p s
```

Turtle-oriented programming

Introdução

Nesta avaliação, você deverá implementar um conjunto de funções para realizar o parsing de uma pequena linguagem de programação para movimentar uma tartaruga em um plano 2D.

Desenvolvimento do parser

Questão 1. Nessa questão desenvolveremos um conjunto de funções para realizar o parsing de uma posição da tartaruga no plano. Posições são representadas pelo seguinte tipo:

```
data Position
      = Position {
           x :: Int
        ,  y :: Int
        } deriving (Eq, Ord, Show)
```

que representa as coordenadas da tartaruga no plano cartesiano.

a) Desenvolva o parser:

```
parseNumber :: Parser Char Int
parseNumber = undefined
```

que processa números inteiros descartando espaços antes e depois da string processada. Sua implementação deve satisfazer os seguintes casos de teste.

```
question01aTests :: TestTree
question01aTests
      = testGroup "Question 01-a Tests"
                  [
                     testCase "Question 01-a success" $
                         parse parseNumber "11" @?= Just 11
                  ,  testCase "Question 01-a success spaces right" $
                         parse parseNumber "11 " @?= Just 11
                  ,  testCase "Question 01-a success spaces left" $
                         parse parseNumber " 11" @?= Just 11
                  ,  testCase "Question 01-a success spaces both" $
                         parse parseNumber "  11  " @?= Just 11
                  ,  testCase "Question 01-a failure no number" $
                         parse parseNumber "abc" @?= Nothing
                  ,  testCase "Question 01-a failure empty" $
                         parse parseNumber "" @?= Nothing
                  ]
```

b) Desenvolva a função

```
parsePosition :: Parser Char Position
parsePosition = undefined
```

que realiza o processamento de uma posição da tartaruga no plano. Uma posição é representada por um par de números naturais separados por um ou mais espaços em branco. Sua implementação deve satisfazer os seguintes casos de teste.

```
question01bTests :: TestTree
question01bTests
      = testGroup "Question 01-b Tests"
                  [
                     testCase "Question 01-b success" $
                         parse parsePosition "11 22" @?= Just (Position 11 22)
                  ,  testCase "Question 01-b success spaces right" $
                         parse parsePosition "11 22  " @?= Just (Position 11 22)
                  ,  testCase "Question 01-b success spaces left" $
                         parse parsePosition " 11 22" @?= Just (Position 11 22)
                  ,  testCase "Question 01-b success spaces both" $
                         parse parsePosition "  11 22 " @?= Just (Position 11 22)
                  ,  testCase "Question 01-b failure no number" $
                         parse parsePosition "abc" @?= Nothing
                  ,  testCase "Question 01-b failure empty" $
                         parse parsePosition "" @?= Nothing
                  ]
```

Questão 2. Nesta questão desenvolveremos um conjunto de funções para implementar um parsing do estado inicial da tartaruga no plano que é representado pelo seguinte tipo.

```
data Turtle
      = Turtle {
          position :: Position
        , facing   :: Facing
        } deriving (Eq, Ord, Show)
```

O tipo `Facing` denota qual a direção em que a tartaruga está caminhando no plano.

```
data Facing = North | South | East | West deriving (Eq, Ord, Show)
```

2-a) Desenvolva a função

```
parseFacing :: Parser Char Facing
parseFacing = undefined
```

que realiza o parsing de uma representação da direção da tartaruga. Representaremos a posição `North` pelo caractere `N`, `South` por `S`, `East` por `E` e `West` por `W`. Sua função deve satisfazer os seguintes casos de teste:

```
question02aTests :: TestTree
question02aTests
      = testGroup "Question 02-a Tests"
                  [
                     testCase "Question 02-a success" $
                         parse parseFacing "N" @?= Just North
                  ,  testCase "Question 02-a success spaces right" $
                         parse parseFacing "N  " @?= Just North
```

```
                    ,  testCase "Question 02-a success spaces left" $
                           parse parseFacing " N" @?= Just North
                    ,  testCase "Question 02-a success spaces both" $
                           parse parseFacing "  N " @?= Just North
                    ,  testCase "Question 02-a failure invalid char" $
                           parse parseFacing "1bc" @?= Nothing
                    ,  testCase "Question 02-a failure empty" $
                           parse parseFacing "" @?= Nothing
                    ]
```

2-b) Desenvolva a função

```
parseTurtle :: Parser Char Turtle
parseTurtle = undefined
```

que processa o estado inicial de uma tartaruga no plano de execução da linguagem turtle. Sua função deve atender os seguintes casos de teste:

```
tur :: Maybe Turtle
tur = Just (Turtle (Position 11 22) North)
```

```
question02bTests :: TestTree
question02bTests
      = testGroup "Question 02-b Tests"
                  [
                     testCase "Question 02-b success" $
                         parse parseTurtle "11 22 N" @?= tur
                  ,  testCase "Question 02-b success spaces right" $
                         parse parseTurtle "11 22 N  " @?= tur
                  ,  testCase "Question 02-b success spaces left" $
                         parse parseTurtle " 11 22 N" @?= tur
                  ,  testCase "Question 02-b success spaces both" $
                         parse parseTurtle "  11 22 N " @?= tur
                  ,  testCase "Question 02-b failure invalid char" $
                         parse parseTurtle "a 11 bc" @?= Nothing
                  ,  testCase "Question 02-b failure empty" $
                         parse parseTurtle "" @?= Nothing
                  ]
```

Questão 3. O objetivo desta questão é o desenvolvimento de um conjunto de funções para fazer o parsing de programas da linguagem turtle.

3-a) Instruções da linguagem turtle são representadas pelo seguinte tipo de dados:

```
data Instr
   = Forward | ToLeft | ToRight | Print
     deriving (Eq, Ord, Show)
```

Desenvolva a função

```haskell
parseInstr :: Parser Char Instr
parseInstr = undefined
```

que realiza o parsing de uma instrução turtle. Cada instrução é representada por uma letra, como se segue: `Forward` é representada pela letra `F`, `ToLeft` por `L`, `ToRight` por `R` e `Print` por `P`. Seu parser deve satisfazer os seguintes casos de teste.

```haskell
question03aTests :: TestTree
question03aTests
      = testGroup "Question 03-a Tests"
                  [
                     testCase "Question 03-a success" $
                         parse parseInstr "F" @?= Just Forward
                  ,  testCase "Question 03-a success spaces right" $
                         parse parseInstr "F" @?= Just Forward
                  ,  testCase "Question 03-a success spaces left" $
                         parse parseInstr " F" @?= Just Forward
                  ,  testCase "Question 03-a success spaces both" $
                         parse parseInstr "  F " @?= Just Forward
                  ,  testCase "Question 03-a failure invalid char" $
                         parse parseInstr "a 11 bc" @?= Nothing
                  ,  testCase "Question 03-a failure empty" $
                         parse parseInstr "" @?= Nothing
                  ]
```

3-b) Programas completos da linguagem turtle são expressos pela configuração inicial da tartaruga no plano e lista de instruções a serem executadas. O tipo `Program` representa programas turtle:

```haskell
data Program
      = Program {
          start :: Turtle
        , code  :: [Instr]
        } deriving (Eq, Ord, Show)
```

O campo `start` representa a posição inicial e code a lista de instruções que deve ser executada. Com base no apresentado, implemente a função:

```haskell
parseProgram :: Parser Char Program
parseProgram = undefined
```

para realizar o parsing de programas turtle. Seu parser deve atender os seguintes casos de teste.

```haskell
prog :: Program
prog = Program (Turtle (Position 11 22) North)
               [Forward, ToLeft, Forward, Forward]

question03bTests :: TestTree
```

```
question03bTests
      = testGroup "Question 03-b Tests"
                  [
                     testCase "Question 03-b success" $
                         parse parseProgram "11 22 N FLFF" @?= Just prog
                  ,  testCase "Question 03-b success spaces right" $
                         parse parseProgram "11 22 N FLFF  " @?= Just prog
                  ,  testCase "Question 03-b success spaces left" $
                         parse parseProgram " 11 22 N FLFF" @?= Just prog
                  ,  testCase "Question 03-b success spaces both" $
                         parse parseProgram "  11 22 N FLFF  " @?= Just prog
                  ,  testCase "Question 03-b failure invalid char" $
                         parse parseProgram "b 11 bc" @?= Nothing
                  ,  testCase "Question 03-b failure empty" $
                         parse parseProgram "" @?= Nothing
                  ]
```