Trabalho Prático 1 - Programação Funcional.

Setup inicial

```
module Main where

import Test.HUnit
```

Introdução

O objetivo desse trabalho é implementar um conjunto de funções para simular a execução de um consulta em um banco de dados. Todos os exercícios podem ser concluídos utilizando funções presentes nas bibliotecas *Prelude*, *Data.Char* e *Data.List*.

Uma boa maneira para conhecer e pesquisar o conteúdo de bibliotecas Haskell é o uso do Hoogle: https://hoogle.haskell.org, uma máquina de busca em documentação de bibliotecas de Haskell.

Bancos de dados textuais

Uma forma de armazenar informação é usando arquivos de texto. Diversos formatos textuais são utilizados com esse intuito, como por exemplo, o formato CSV. Nesse trabalho, utilizaremos um formato simples, em que cada linha representa um registro de banco de dados e cada coluna um campo de um registro. Um exemplo deste tipo de banco de dados é apresentado a seguir.

```
Nome Sobrenome Idade
Astrogildo Tibúrcildo 40
Hermergarda Silvinéia 30
Clodoaldo Murtinho 67
Joaquim Manuel 28
```

Observe que, nesse formato, a primeira linha representa o nome de cada um dos campos que compõe esse banco de dados.

Representamos esse tipo de informação textual usando os seguintes tipos Haskell:

```
type Field = String
type Row   = [Field]
type Table = [Row]
```

Por questões de simplicidade, representamos todo tipo de informação como `String`s.

Parsing de tabelas

Parsing é o processo de conversão de informação representada como uma sequência de símbolos em uma estrutura de dados.

A primeira tarefa desse trabalho consiste na implementação de uma função que realiza o parsing de tabelas.

```
parseTable :: String -> Table
parseTable = undefined
```

Sua função deve ser implementada de forma a satisfazer o seguinte caso de teste.

```
table :: String
table = concat [ "Nome Sobrenome Idade\n"
               , "Astrogildo Tibúrcildo 40\n"
               , "Hermergarda Silvinéia 30\n"
               , "Clodoaldo Murtinho 67\n"
               , "Joaquim Manuel 28"]

tableRep :: Table
tableRep = [ ["Nome", "Sobrenome", "Idade"]
           , ["Astrogildo", "Tibúrcildo", "40"]
           , ["Hermergarda", "Silvinéia", "30"]
           , ["Clodoaldo", "Murtinho", "67"]
           , ["Joaquim", "Manuel", "28"]
           ]


parseTableTest :: Test
parseTableTest = "parseTableTest" ~: parseTable table ~=? tableRep
```

Tabelas bem formadas

Evidentemente, nem todo arquivo de texto consiste de uma tabela válida. Dizemos que uma tabela é válida se:

1. Sua primeira linha é formada pelos nomes dos campos dessa tabela.
2. Todas as linhas possuem o mesmo número de colunas.
3. O arquivo possui pelo menos duas linhas (uma contendo o nome dos campos e outra contendo um registro).

Dadas as restrições acima, implemente a função:

```
validTable :: Table -> Bool
validTable = undefined
```

que verifica se uma tabela fornecida como entrada é ou não válida. O seguinte caso de teste deve ser satisfeito por sua implementação.

```
validTableTest :: Test
validTableTest = "validTableTest" ~: validTable tableRep ~=? True
```

Impressão de resultados

No primeiro exercício, você implementou uma função para converter uma `String` em um valor de tipo `Table`. O objetivo desse exercício é a conversão de uma tabela em um formato textual legível. Considerando a tabela apresentada como exemplo, sua representação em formato legível seria:

```
+-----------+----------+-----+
|NOME       |SOBRENOME |IDADE|
+-----------+----------+-----+
|Astrogildo |Tibúrcildo|40   |
|Hermergarda|Silvinéia |30   |
|Clodoaldo  |Murtinho  |67   |
|Joaquim    |Manuel    |28   |
+-----------+----------+-----+
```

Observe que no exemplo acima, várias operações foram feitas sobre a tabela de entrada:

1. O tamanho de cada coluna é dado pelo maior valor presente nesta coluna (incluindo o nome da coluna).
2. Nomes de colunas são expressos utilizando letras maíusculas.
3. Bordas foram adicionadas para delimitar a tabela e facilitar sua leitura.

Para implementar a impressão de tabelas, basta seguir os seguintes passos.

a) Implemente a função

```
ppLine :: [Int] -> String
ppLine = undefined
```

que a partir de uma lista contendo os comprimentos de cada campo da tabela, imprime uma linha de cabeçalho. Utilize o seguinte teste para guiar sua implementação.

```
ppLineTest :: Test
ppLineTest = "ppLineTest" ~: ppLine [3,4,2] ~=? "+---+----+--+"
```

b) Implemente a função

```
ppField :: Int -> String -> String
ppField = undefined
```

que a partir do comprimento de um campo, o valor do campo atual retorna o valor do campo devidamente formatado incluindo espaços adicionais à esquerda. O seguinte caso de teste ilustra o comportamento esperado por essa função.

```
ppFieldTest :: Test
ppFieldTest = "ppFieldTest" ~: ppField 9 "carlos" ~=? "carlos   "
```

c) Implemente a função

```
ppRow :: [(Int, String)] -> String
ppRow = undefined
```

que a partir de uma lista de pares, cujo primeiro componente é o tamanho do campo e o segundo o valor do campo, formata um registro da tabela.

```
ppRowTest :: Test
ppRowTest = "ppRowTest" ~: ppRow [(9,"carlos"), (5,"20")] ~=? "|carlos   |20   |"
```

d) Implemente a função

```
fieldSizes :: Table -> [Int]
fieldSizes = undefined
```

que calcula o comprimento de cada campo de uma tabela.

```
fieldSizesTest :: Test
fieldSizesTest = "fieldSizesTest" ~: fieldSizes tableRep ~=? [11, 10, 5]
```

e) De posse de todas as implementações anteriores, combine-as para produzir a função

```
ppTable :: Table -> String
ppTable = undefined
```

que produz a versão legível de dados presentes em uma tabela textual.

Função main

```
tests :: Test
tests = TestList [ parseTableTest
                 , validTableTest
                 , ppLineTest
                 , ppFieldTest
                 , ppRowTest
                 , fieldSizesTest]
main :: IO ()
main = runTestTT tests >> return ()
```