Avaliação 2 de 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.
- 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.
- 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

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

module Main where

import Data.Time.Clock
import Data.Time.Calendar
import ParseLib
import System.Environment (getArgs)
```

Gerenciador de tarefas

Introdução

O objetivo dessa avaliação é a criação de uma ferramenta de linha de comando para o gerenciamento de tarefas. A ferramenta a ser implementada lerá um arquivo contendo as tarefas e produzirá um relatório contendo:

1. Tarefas atrasadas.
2. Tarefas que devem ser concluídas no dia atual.
3. Tarefas a serem concluídas até a próxima semana.

Para concluir a implementação dessa ferramenta, desenvolva os exercícios especificados a seguir.

Questão 1. (Valor 3,0 pontos). Uma importante tarefa em um software para gestão de tarefas é a correta manipulação de datas. O tipo `Date` representa datas formadas por seu dia, mês e ano.

```haskell
data Date
 = Date {
     day   :: Int
   , month :: Int
   , year  :: Int
   } deriving Eq
```

a) *(Valor 0,5 ponto)* Implemente uma instância de `Show` para o tipo `Date` de forma que uma data seja exibida no formato DD/MM/YYYY, em que DD denota o dia, MM o mês e YYYY o ano.

```haskell
instance Show Date where
   show = undefined
```

b) *(Valor 0,5 ponto)* Outra tarefa importante sobre o tipo data é comparação entre valores. Para isso, implemente uma instância de `Ord` para o tipo `Date`. Para implementar a instância de `Ord` basta implementar a função `<=` para datas.

```haskell
instance Ord Date where
   _ <= _ = undefined
```

c) *(Valor 1,0 ponto)* A função a seguir obtém o valor da data atual:

```
currentDate :: IO Date
currentDate
      = f <$> getCurrentTime
        where
          g (y,m,d) = Date d m (fromInteger y)
          f = g . toGregorian . utctDay
```

Observe que essa função utiliza o tipo `IO` e, portanto, realiza uma operação de entrada e saída. Além disso, ela deve ser chamada dentro de um bloco `do`.

Usando a função `currentDate` como modelo, implemente a função

```
sevenDaysAfter :: IO Date
sevenDaysAfter = undefined
```

que retorna a data correspondente a sete dias depois da data atual.

d) *(Valor 1,0 ponto)* Considerando o formato textual de datas, implemente um parser para o tipo `Date`.

```
parseDate :: Parser Char Date
parseDate = undefined
```

As próximas questões irão envolver o tipo `Task` que representa uma tarefa armazenada em um arquivo fornecido como entrada.

```
data Task
   = Task {
       deadline    :: Date
     , description :: String
     }  deriving (Eq, Ord)
```

O significado dos campos do tipo `Task` é como se segue: `deadline` especifica a data limite para a realização da tarefa e `description` a descrição textual da tarefa em questão.

Tarefas são representadas textualmente de forma simples: primeiro especificamos a data limite para a tarefa, seguida de um ou mais espaços e de sua descrição. Usamos o caractere de ';' para separar diferentes tarefas em uma mesma string. A seguir, apresentamos um exemplo deste formato de tarefas:

```
28/06/2021 Tarefa 1 ;
26/07/2021 Tarefa 2 ;
02/08/2021 Tarefa 3 ;
```

Questão 2. (Valor 1,0 ponto) Desenvolva uma instância de `Show` para o tipo `Task` que produza uma string idêntica à sua descrição textual.

```
instance Show Task where
   show = undefined
```

*Questão 3. (Valor 3,0 pontos)*Com base no formato do arquivo de tarefas, construa um parser que o processe e retorne uma lista das tarefas nele armazenadas.

```
taskFileParser :: Parser Char [Task]
taskFileParser = undefined
```

Questão 4. (Valor 1,0 ponto) O objetivo desta questão é a produção de um relatório contendo as tarefas atrasadas, que devem ser concluídas na data atual e que devem ser concluídas em até 7 dias. O tipo `Report` agrupa essas informações:

```
data Report
  = Report {
      late  :: [Task]
    , today :: [Task]
    , nexts :: [Task]
    } deriving Show
```

Desenvolva a função

```
report :: [Task] -> Report
report = undefined
```

Questão 5. (Valor 1,0 ponto) O resultado final da ferramenta é um relatório classificando as tarefas. Considerando o seguinte arquivo de modelo:

```
18/06/2021 Tarefa 1 ;
26/07/2021 Tarefa 2 ;
30/07/2021 Tarefa 3 ;
```

Temos que a Tarefa 1 está atrasada, a Tarefa 2 possui deadline para hoje e a Tarefa 3 deve ser concluída em até 7 dias. Sua ferramenta deverá apresentar o seguinte relatório

```
Tarefas atrasadas:

18/06/2021 Tarefa 1 ;

Tarefas para hoje:

26/07/2021 Tarefa 2;

Tarefas para concluir em uma semana:

30/07/2021 Tarefa 3
```

A partir da descrição anterior, implemente a função:

```
printReport :: Report -> String
printReport = undefined
```

que gera uma string no formato apresentado para os dados do relatório (valor do tipo `Report`).

Questão 6. (Valor 1,0 ponto) De posse de todas as implementações anteriores, implemente a função `main` de sua ferramenta:

```
main :: IO ()
main = do
         _ <- getArgs
         undefined
```

A partir de um nome de arquivo, sua ferramenta deverá lê-lo, realizar seu parsing e exibir o relatório de tarefas. Para obter o nome de arquivo, você deverá usar a função `getArgs`, para obter os argumentos passados por linha de comando para sua ferramenta.

Para executar seu programa usando o stack e passar argumentos de linha de comando basta:

```
stack exec prova02-exe -- argumento
```

em que argumento é o valor que você deseja passar como argumento adicional para a execução de seu programa.