Avaliação 2 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.
- Cada questão desta avaliação possui o valor de 1,0 ponto.
- 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 System.Environment
import ParseLib
```

Processando arquivos DOT

DOT é uma linguagem simples para descrição de grafos. Um descrição de grafo é iniciada pelas palavras reservadas `graph` ou `digraph` seguida de um nome para o grafo definido e uma listagem de arestas.

Considere o seguinte grafo.

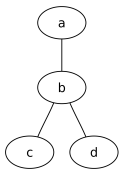

Figure 1: Grafo não directionado

O código dot correspondente é dado por

```
graph graphname {
    a--b;
    b--c;
    b--d;
}
```

Note que arestas entre dois nós são especificadas pelos caracteres `--`. Para especificar um grafo direcionado, basta iniciar a definição usando a palavra reservada `digraph` e especificar arestas usando `->`, como se segue.

```
digraph graphname {
    a -> b;
    b -> c;
    b -> d;
}
```

A figura 2 ilustra o grafo direcionado descrito pelo trecho de código anterior.

O objetivo desta avaliação é o desenvolvimento de um parser para arquivos dot. Para esse intuito, faça o que se pede a seguir.

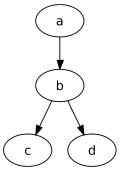

Figure 2: Grafo não directionado

Definindo arquivos DOT

O primeiro passo é definir um tipo de dados para representar arquivos dot. O tipo `Graph` a seguir representa grafos direcionados e não direcionados.

```
data Graph = Undirected Id [Edge]
           | Directed Id [Edge]
           deriving Eq
```

O tipo `Id` é usado para representar identificadores e é simplesmente um sinônimo para strings.

```
type Id = String
```

O tipo `Edge` especifica arestas que são representadas por pares de identificadores.

```
type Edge = (Id,Id)
```

Usando os tipos de dados acima, podemos descrever o grafo não direcionado apresentado anteriormente pelo seguinte valor:

```
undirected :: Graph
undirected = Undirected "graphname"
                        [ ("a", "b")
                        , ("b", "c")
                        , ("b", "d")]
```

Questão 1. Elabore uma instância de `Show` para o tipo Graph de forma que a string retornada seja o código dot correspondente ao grafo fornecido como argumento de entrada. Como exemplo, considere a seguinte string gerada ao executar a função `show` sobre o grafo `undirected`

```
$> show undirected
graph graphname {\na -- b ; b -- c ; b -- d ; \n}
```

```
instance Show Graph where
  show = undefined
```

Questão 3. A biblioteca de parsing possui a função

```
identifier :: Parser Char String
```

que realiza o parsing de um identificador. Porém, idealmente, esse e outros parsers da biblioteca deveriam descartar espaços em branco do texto fornecido na entrada.

Infelizmente, esse não é o caso. A biblioteca fornece a função

```
whitespace :: Parser Char ()
```

que consome todos os caracteres em branco de um prefixo da entrada.

Combine essas funções para implementar o parser

```
ident :: Parser Char String
ident = undefined
```

que faz o parsing de um identificador descartando caracteres em branco presentes no início da entrada.

Questão 4. A biblioteca de parsing possui a função

```
token :: String -> Parser Char String
```

que realiza o parsing de uma string fornecida como entrada. Assim como a função `identifier`, `token` não remove os caractees em branco presentes no início da entrada. Implemente a função:

```
tok :: String -> Parser Char String
tok = undefined
```

que processa a string s, fornecida como argumento, descartando quaisquer caracteres em branco que possam estar presentes no início da entrada.

Questão 5. Usando as funções anteriores construa o parser

```
edgeParser :: String -> Parser Char Edge
edgeParser = undefined
```

que processa uma aresta no formato dot. A função `edgeParser` recebe como parâmetro uma string que representa o separador dos nós que formam uma aresta no grafo.

Questão 6. O objetivo desta questão é implementar um parser para a lista de arestas presente entre chaves "{" e "}" em um arquivo dot. A função `bodyParser` deve receber como argumento uma string que denota o separador entre nós que compõe as arestas a serem processadas.

```
bodyParser :: String -> Parser Char [Edge]
bodyParser = undefined
```

Para impelmentar essa função, utilize as funções anteriores e a função

```
pack :: Parser s a -> Parser s b ->
        Parser s c -> Parser s b
```

da biblioteca de parsing.

Questão 7. De posse das funções anteriores, podemos criar a função

```
undirParser :: Parser Char Graph
undirParser = undefined
```

que realiza o parsing de uma string em formato dot que descreve um grafo não direcionado.

Questão 8. Implemente a função

```
dirParser :: Parser Char Graph
dirParser = undefined
```

que realiza o parsing de uma string em formato dot que descreve um grafo direcionado.

Questão 9. Usando as duas funções anteriores, implemente o parser para strings descrevendo grafos usando o formato dot.

```
dotParser :: Parser Char Graph
dotParser = undefined
```

Questão 10. O último componente do seu parser é ser capaz de processar arquivos texto contendo código dot usando o parser construído nas questões anteriores e exibindo, na saída padrão, o nome do grafo processado e a quantidade de arestas presente neste grafo.

Para isso, você deve ser capaz de receber um nome de arquivo ao executar seu código em linha de comando usando a função `getArgs`, presente na biblioteca `System.Environment`. Para usar essa biblioteca, inclua a seguinte definição de import no cabeçalho deste arquivo, como se segue:

```
module Main where

import System.Environment
import ParseLib
```

A função `getArgs :: IO [String]` retorna uma lista contendo os argumentos de linha de comando fornecidos quando da execução de seu programa no terminal. Argumentos adicionais podem ser fornecidos da seguinte maneira via stack:

```
$> stack build
$> stack exec prova2-exe -- teste.dv
```

Considerando o acima exposto, implemente a função `main :: IO ()` que, a partir de um nome de arquivo fornecido como argumento, faz o parsing da descrição no formato dot e imprime, na saída padrão, o nome do grafo e o seu número de arestas.

```
main :: IO ()
main = undefined
```