Problema 3: Análise de ficheiros *log*

Algo correu mal no *datacenter* do DCC e precisamos da vossa ajuda para perceber o que se passou. Felizmente temos os registos dos acontecimentos (*logs*). São ficheiros de texto com uma mensagem por linha; cada linha começa com uma letra indicando o tipo de mensagem:

- `I` para mensagens informativas;
- `W` para mensagens de aviso (*warnings*);
- `E` para mensagens de erro.

As mensagens de erros têm também um nível inteiro entre 1 e 100 que indica a gravidade da ocorrência (1 é o menos grave e 100 o mais grave). Todas as mensagens têm um inteiro que representa o tempo da ocorrência (*timestamp*); o resto da linha é o conteúdo da mensagem.

Eis um extrato de duas linhas com uma mensagem informativa no instance $t = 147$ seguida de um erro de nível 2 em $t = 148$:

```
I 147 mice in the air, I'm afraid, but you might catch a bat, and
E 2 148 #56k istereadeat lo d200ff] BOOTMEM
```

O ficheiro `sample.log` contém um pequeno exemplo destas mensagens; o ficheiro `error.log` contém todas as mensagens recuperadas do *datacenter*. Como este último ficheiro é longo, vamos escrever um programa para auxiliar a filtrar a informação. Começamos pelas declarações de tipos para estruturar a informação:

```haskell
-- tipos de mensagens
data MessageType = Info
                 | Warning
                 | Error Int    -- argumento: nível do erro
                 deriving (Show, Eq)

type TimeStamp = Int    -- instante de tempo

-- entradas num ficheiro *log*
data LogEntry = LogMessage MessageType TimeStamp String
```

```
  | Unknown String
deriving (Show, Eq)
```

Note que `LogEntry` tem dois construtores: `LogMessage` representa mensagens corretamente formatadas e `Unknown` representa outras linhas de texto que não sigam o formato indicado acima.

Preparação

Deve descarregar o ficheiro `Log.hs` com as declarações acima e colocá-lo no mesmo diretório em que vai desenvolver o seu código (num outro módulo). Para usar as definições do módulo `Log.hs` no seu programa deve colocar a seguinte declaração

```
import Log
```

no início do seu módulo. Não coloque as suas definições no módulo `Log.hs` porque não vai submeter esse ficheiro!

Exercício 1

Vamos começar por definir uma função para analisar uma linha de texto e converter num valor `LogEntry` apropriado.

- Exercício 1: Análise de uma mensagem

Depois de fazer a análise de uma linha, podemos definir uma função para analisar o ficheiro completo usando `lines` para partir o conteúdo do ficheiro em linhas:

```
parseLog :: String -> [LogEntry]
parseLog txt = map parseMessage (lines txt)
```

Exercício 2

Agora que conseguimos fazer análise do ficheiro de *log* deparamos com um problema: as entradas estão fora de ordem! Vamos corrigir isso colocando as mensagens numa estrutura de árvore de pesquisa.

- Exercício 2: Inserir uma entrada numa árvore de pesquisa

Exercício 3

Vamos usar a função anterior para colocar as mensagens por ordem.

- Exercício 3: Colocar mensagens por ordem

Conclusão

Será que consegue juntar todas as componentes anteriores para descobrir o que aconteceu no incidente do *datacenter*? Escreva um programa que:

(1) leia o ficheiro de texto `error.log` usando `readFile`;
(2) converta numa lista de `LogEntry` usando `parseLog`;
(3) coloque as mensagens por ordem usando `sortMessages`;
(4) filtre e imprima as mensagens de erro com gravidade 50 ou superior.

(Este último programa não é para submeter.)

Adaptado de um exercício do curso CIS 194, University of Pensilvania.

Pedro Vasconcelos, 2021

Exercício 1: Análise de uma mensagem

Vamos começar por definir uma função para analisar uma linha de texto do ficheiro *log* e converter num valor `LogEntry` apropriado.

Escreva uma função

```
parseMessage :: String -> LogEntry
```

que converte uma linha de texto numa entrada de *log*.

Deve importar as definições de tipos do ficheiro Log.hs e não definir o tipo `LogEntry` no seu módulo.

As seguintes funções do prelúdio poderão ser úteis: `read` (para converter uma `String` num inteiro), `words` (para partir uma `String` em palavras), `unwords` (para juntar palavras numa `String`).

Exemplos

```
parseMessage "E 2 562 Missed timeout; ignoring"
  == LogMessage (Error 2) 562 "Missed timeout; ignoring"
parseMessage "I 29 la la la"
  == LogMessage Info 29 "la la la"
parseMessage "This is not in the right format"
  == Unknown "This is not in the right format"
```

Exercício 2: Inserir uma entrada numa árvore de pesquisa

Agora que conseguimos fazer análise do ficheiro de *log* deparamos com um problema: as entradas estão fora de ordem! Vamos corrigir a situação colocando-as numa estrutura de árvore de pesquisa (também declarada no ficheiro `Log.hs`):

```
data MessageTree = Empty
                 | Node LogEntry MessageTree MessageTree
                 deriving (Show, Eq)
```

A árvore de mensagens é uma estrutura recursiva: ou é *vazia* ou é um *nó* com uma entrada e duas sub-árvores esquerda e direita.

Estas árvores devem ser ordenadas pelo *timestamp* das entradas, ou seja, o *timestamp* da entrada em qualquer nó deve ser maior do que os das entradas à esquerda e menor do que os das entradas à direita.

Defina uma função recursiva

```
insert :: LogEntry -> MessageTree -> MessageTree
```

que insere uma nova mensagem numa árvore mantendo a ordem de *timestamps*.

- Pode assumir que a `MessageTree` dada respeita a propriedade de ordenação acima. Além disso todas as entradas na árvore são da forma `LogMessage` (e por isso contêm um *timestamp*).
- No caso da `LogEntry` a inserir ser da forma `Unknown`, então a função deve devolver a árvore original inalterada.
- No caso de já existir na árvore uma entrada com exatamente o mesmo *timestamp* então a nova entrada deve ser inserida à sua direita na árvore (ou seja, a última entrada a ser inserida deve ser considerada maior do que qualquer anterior).

Exercício 3: Colocar mensagens por ordem

Vamos colocar por ordem uma lista de mensagens. Para tal necessitamos de duas funções:

```
build :: [LogEntry] -> MessageTree     -- construir uma árvore ordenada

inOrder :: MessageTree -> [LogEntry]   -- listar mensagens por ordem
```

A função `build` deve introduzir uma lista de mensagens numa árvore vazia usando a função `insert` de forma a garantir que a árvore fica ordenada.

A função `inOrder` recebe uma árvore ordenada e produz uma lista; para cada nó deve recursivamente listar as mensagens à esquerda, seguida da mensagem no nó e depois recursivamente listar as mensagens à direita. Isto chama-se uma *travesia por ordem infixa* da árvore.

Finalmente a função para ordenar mensagens é simplemente a composição destas duas: primeiro construimos a árvore e depois listamos as mensagens.

```
sortMessages :: [LogEntry] -> [LogEntry]
sortMessages msgs = inOrder (build msgs)
```

[Note que haveria outras formas mais eficientes de ordenar a lista; neste problema pretende-se exercitar o uso de estruturas recursivas!]