CIS 194: Homework 2

Due Monday January 28

Something has gone terribly wrong!

- Files you will need: `Log.hs`, `error.log`, `sample.log`
- Files you should submit: `LogAnalysis.hs`

Log file parsing

We're really not sure what happened, but we did manage to recover the log file `error.log`. It seems to consist of a different log message on each line. Each line begins with a character indicating the type of log message it represents:

- `'I'` for informational messages,
- `'W'` for warnings, and
- `'E'` for errors.

The error message lines then have an integer indicating the severity of the error, with 1 being the sort of error you might get around to caring about sometime next summer, and 100 being epic, catastrophic failure. All the types of log messages then have an integer time stamp followed by textual content that runs to the end of the line. Here is a snippet of the log file including an informational message followed by a level 2 error message:

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

It's all quite confusing; clearly we need a program to sort through this mess. We've come up with some data types to capture the structure of this log file format:

```
data MessageType = Info
                 | Warning
                 | Error Int
                 deriving (Show, Eq)

type TimeStamp = Int

data LogMessage = LogMessage MessageType TimeStamp String
                | Unknown String
                deriving (Show, Eq)
```

Note that `LogMessage` has two constructors: one to represent normally-formatted log messages, and one to represent anything else that does not fit the proper format.

We've provided you with a module `Log.hs` containing these data type declarations, along with some other useful functions. Download `Log.hs` and put it in the same folder where you intend to put your homework assignment. Please name your homework assignment `LogAnalysis.hs` (or `.lhs` if you want to make it a literate Haskell document). The first few lines of `LogAnalysis.hs` should look like this:

```
{-# OPTIONS_GHC -Wall #-}
module LogAnalysis where

import Log
```

which sets up your file as a module named `LogAnalysis`, and imports the module from `Log.hs` so you can use the types and functions it provides.

Exercise 1 The first step is figuring out how to parse an individual message. Define a function

```
parseMessage :: String -> LogMessage
```

which parses an individual line from the log file. For example,

```
parseMessage "E 2 562 help help"
    == LogMessage (Error 2) 562 "help help"
```

```
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"
```

Once we can parse one log message, we can parse a whole log file. Define a function

```
parse :: String -> [LogMessage]
```

which parses an entire log file at once and returns its contents as a list of `LogMessages`.

To test your function, use the `testParse` function provided in the `Log` module, giving it as arguments your `parse` function, the number of lines to parse, and the log file to parse from (which should also be in the same folder as your assignment). For example, after loading your assignment into GHCi, type something like this at the prompt:

```
testParse parse 10 "error.log"
```

Don't reinvent the wheel! (That's so *last* week.) Use Prelude functions to make your solution as concise, high-level, and functional as possible. For example, to convert a `String` like `"562"` into an `Int`, you can use the `read` function. Other functions which may (or may not) be useful to you include `lines`, `words`, `unwords`, `take`, `drop`, and `(.)`.

Putting the logs in order

Unfortunately, due to the error messages being generated by multiple servers in multiple locations around the globe, a lightning storm, a failed disk, and a bored yet incompetent programmer, the log messages are horribly out of order. Until we do some organizing, there will be no way to make sense of what went wrong! We've designed a data structure that should help—a binary search tree of `LogMessages`:

```
data MessageTree = Leaf
                 | Node MessageTree LogMessage MessageTree
```

node with subtrees make up a whole tree

Note that `MessageTree` is a recursive data type: the `Node` constructor itself takes two children as arguments, representing the left and right subtrees, as well as a `LogMessage`. Here, `Leaf` represents the empty tree.

A `MessageTree` should be sorted by timestamp: that is, the timestamp of a `LogMessage` in any `Node` should be greater than all timestamps of any `LogMessage` in the left subtree, and less than all timestamps of any `LogMessage` in the right child.

// using binary-search-tree property

`Unknown` messages should not be stored in a `MessageTree` since they lack a timestamp.

Exercise 2 Define a function

```
insert :: LogMessage -> MessageTree -> MessageTree
```

which inserts a new `LogMessage` into an existing `MessageTree`, producing a new `MessageTree`. `insert` may assume that it is given a sorted `MessageTree`, and must produce a new sorted `MessageTree` containing the new `LogMessage` in addition to the contents of the original `MessageTree`.

However, note that if `insert` is given a `LogMessage` which is `Unknown`, it should return the `MessageTree` unchanged.

Exercise 3 Once we can insert a single `LogMessage` into a `MessageTree`, we can build a complete `MessageTree` from a list of messages. Specifically, define a function

```
build :: [LogMessage] -> MessageTree
```

which builds up a `MessageTree` containing the messages in the list, by successively inserting the messages into a `MessageTree` (beginning with a `Leaf`).

Exercise 4 Finally, define the function

```
inOrder :: MessageTree -> [LogMessage]
```

which takes a sorted `MessageTree` and produces a list of all the `LogMessage`s it contains, sorted by timestamp from smallest to biggest. (This is known as an *in-order traversal* of the `MessageTree`.)

With these functions, we can now remove `Unknown` messages and sort the well-formed messages using an expression such as:

```
inOrder (build tree)
```

[Note: there are much better ways to sort a list; this is just an exercise to get you working with recursive data structures!]

Log file postmortem

Exercise 5 Now that we can sort the log messages, the only thing left to do is extract the relevant information. We have decided that "relevant" means "errors with a severity of at least 50".

Write a function

```
whatWentWrong :: [LogMessage] -> [String]
```

which takes an *unsorted* list of `LogMessages`, and returns a list of the messages corresponding to any errors with a severity of 50 or greater, sorted by timestamp. (Of course, you can use your functions from the previous exercises to do the sorting.)

For example, suppose our log file looked like this:

```
I 6 Completed armadillo processing
I 1 Nothing to report
E 99 10 Flange failed!
I 4 Everything normal
I 11 Initiating self-destruct sequence
E 70 3 Way too many pickles
E 65 8 Bad pickle-flange interaction detected
W 5 Flange is due for a check-up
I 7 Out for lunch, back in two time steps
E 20 2 Too many pickles
I 9 Back from lunch
```

This file is provided as `sample.log`. There are four errors, three of which have a severity of greater than 50. The output of `whatWentWrong` on `sample.log` ought to be

```
[ "Way too many pickles"
, "Bad pickle-flange interaction detected"
, "Flange failed!"
]
```

You can test your `whatWentWrong` function with `testWhatWentWrong`, which is also provided by the `Log` module. You should provide `testWhatWentWrong` with your `parse` function, your `whatWentWrong` function, and the name of the log file to parse.

Miscellaneous

- We will test your solution on log files other than the ones we have given you, so no hardcoding!
- You are free (in fact, encouraged) to discuss the assignment with any of your classmates as long as you type up your own solution.

Epilogue

Exercise 6 (Optional) For various reasons we are beginning to suspect that the recent mess was caused by a single, egotistical

hacker. Can you figure out who did it?