CIS 194: Homework 3

Due Friday, September 19, 2014

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
  deriving (Show, Eq)
```

(The mysterious `deriving (Show, Eq)` annotations allow these types to be printed in a GHCi session and allow equality comparison among member of the types. We’ll learn more about these later.)

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.

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. Functions which may (or may not) be useful to you include `lines`, `words`, `unwords`, `take`, `drop`, `(.)`, `map`, and `filter`.

Exercise 1 The first step is figuring out how to parse an individual message. However, perhaps the file is corrupted even worse than we thought: maybe individual lines are garbled. So, we can’t be sure that a line from the input will be a valid `LogMessage`. Thus, we define a type (included in the provided `Log.hs`) to allow for the possibility of failure:

```
data MaybeLogMessage = ValidLM LogMessage
                     | InvalidLM String
  deriving (Show, Eq)
```

As you can see, a `MaybeLogMessage` either contains a proper `LogMessage` or just an unformatted string.

Now, you can define a function

```
parseMessage :: String -> MaybeLogMessage
```

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

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

parseMessage "I 29 la la la"
    == ValidLM (LogMessage Info 29 "la la la")

parseMessage "This is not in the right format"
    == InvalidLM "This is not in the right format"
```

There is a very useful function toward the bottom of `Log.hs` that you will need to use here!

Exercise 2 It's not terribly hard to make `parseMessage` work over all the lines in a file. But, doing so would produce a `[MaybeLogMessage]`, where we really just want a `[LogMessage]` for further processing – let's throw out the invalid messages.

Write a function

```
validMessagesOnly :: [MaybeLogMessage] -> [LogMessage]
```

that throws out invalid messages.

Exercise 3 Now, we can put these pieces together to define

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

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

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"
```

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!

Exercise 4 Any sorting function is going to need to compare two `LogMessages` to see which one should come first. But, since we've just created the `LogMessage` type, there is no way for the computer to know how to compare two of them. We must write a comparison function!

In general, comparing two items for ordering can yield one of three results: less-than, equal-to, or greater-than. Haskell codifies this idea as a datatype

```
data Ordering = LT | EQ | GT
```

`Ordering` is part of the `Prelude` (the set of things automatically included), so its definition doesn't appear in `Log.hs` nor should it appear in your code.

Define a function

```
compareMsgs :: LogMessage -> LogMessage -> Ordering
```

that compares two `LogMessages` based on their timestamps.

Here are some examples:

```
compareMsgs (LogMessage Warning 153 "Not a speck of light is showing, so the danger must be growing...")
            (LogMessage Info 208 "the Weighted Companion Cube cannot talk")
  == LT

compareMsgs (LogMessage (Error 101) 2001 "My God! It's full of stars!")
            (LogMessage Info 2001 "Daisy, Daisy, give me your answer do.")
  == EQ
```

Exercise 5 Now that you have said how to compare messages, you can sort the list. Write a function

```
sortMessages :: [LogMessage] -> [LogMessage]
```

that sorts the list of messages. Do *not* write out a full sorting algorithm! Instead, poke around in the `Data.List` module looking for a very convenient function!

Log file postmortem

Exercise 6 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.

Exercise 7 Take a look at the output from a run of `testWhatWentWrong`. For both log files provided, a certain condiment seems to stand out (different condiments for the different files!). It might be more informative to see both warnings and errors that mention that particular condiment.

Define a function

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

that filters a list of `LogMessages` to include only those messages that contain the string provided. *Make your function case-insensitive*, so that `"relish"` matches `"Relishsign detected!!"`.

Exercise 8 We now wish to see a combined output, with both all high-severity messages and all messages containing a certain keyword included. Write a function

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

that makes a list including both all high-severity errors and all messages containing the provided string. Note that you will likely have to edit (that is, *refactor*) previously-written code to be able to do this without repeating yourself much. To test your function, run something like this in GHCi:

```
LogAnalysis> testWhatWentWrong parse (whatWentWrongEnhanced "relish") "error.log"
```

(Why does it work to pass only one parameter to `whatWentWrongEnhanced`?)

You get kudos if you can manage to use the following function:

```
(|||) :: (LogMessage -> Bool) -> (LogMessage -> Bool) -> LogMessage -> Bool
(|||) f g x = f x || g x    -- (||) is Haskell's ordinary "or" operator
```