Problem Set #2

due September 17

Some problems adapted from the Haskell class by Brent Yorgey available at Haskell.org. In addition to the starter code `Ps1.hs`, download log files `error.log` and `sample.log`.

The task is to parse a file of log messages where each message is one of the following:

- `I` is an information message,
- `W` is a warning message and
- `E` is an error message.

An `I` message begins with `I` followed by the message id (i.e., time stamp) followed by some text:

```
I 6 Completed armadillo processin
```

A `W` messages begins with `W` followed by the message id, followed by some text:

```
W 2 Completed armadillo processin
```

An `E` message begins with `E` followed by a number indicating the severity of the error, followed by the message id, followed by some text:

```
E 99 9 Completed armadillo processin
```

We will make use of the following ADTs to represent messages

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

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

The task is to parse a file of log messages into structures that one can search over.

Implement the following functions:

(1) `parseMessage::String -> LogMessage`. This function takes a message string and creates a log message. If the string matches one of the message types, `parseMessage` creates the corresponding `LogMessage`, otherwise it creates an `Unknown` message. For example

```
> parseMessage "E 99 9 Completed armadillo processin"
LogMessage (Error 99) 9 "Completed armadillo processin"
```

Hint: library functions `words, unwords`, and `read` will be useful here.

(2) `parse::String -> [LogMessage]`. This function takes a string made up of multiple lines of text and parses each line into a message producing a list of log messages. In addition to `words` and `unwords`, look up `lines` as well. To run your `parse` function on the entire `error.log` then display the first 10 messages, you will need to run the wrapper function `runParse` given as starter code:

```
> runParse parse 10 "error.log"
```

You can also test on the smaller `sample.log` file:

```
> runParse parse 11 "sample.log"
```

Note: we need the wrapper function because of file IO. In Haskell, IO operations are encapsulated into the IO monad and we need a special way of “binding” the monad and `parse` (roughly speaking; we'll study monads in detail soon).

(3) `insert::[LogMessage] -> LogMessage -> [LogMessage]` takes a list *sorted* by message id and a new message and returns a new sorted list of messages inserting the new message at the first place where its id is less than or equal to the next element's id. If the new message is `Unknown`, `insert` just drops it and returns the original list.

(4) `insertionSort::[LogMessage] -> [LogMessage]` takes an unsorted list of messages and produces a sorted list. Naturally, it will make use of `insert`.

(5) Finally, `severeErrors::Int -> [LogMessage] -> [LogMessage]` takes a severity level and a list of messages and retrieves only the `Error` messages with larger severity levels. We will be testing your function using the wrapper `runParse`. For example, the following line parses `sample.log` and displays the first 11 messages of severity higher than 50:

```
> runParse (severeErrors 50 . parse) 11 "sample.log"
```

Some notes. Below is the starter code also given in `Ps2.hs`.

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

module Ps2 where

import Data.Char

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

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

-- | @runParse f n fp@ tests the log file parser @f@ by running it
--   on the log file @fp@ and displaying the first @n@ results.
runParse :: (String -> [LogMessage])
          -> Int
          -> FilePath
          -> IO [LogMessage]
runParse f n fp = (readFile fp) >>= mBind
                where mBind bigstr = return (take n $ f bigstr)

parseMessage::String->LogMessage
parseMessage s = ...
```

Haskell Style Guide. Adapted from Stephanie Weirich. I include the Style Guide again. Read it carefully as I will be marking the TA points on Submitty according to the adherence of your code to the style guide.

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the function first, try deducing the signature and if this doesn't work, there is always the `:t` command.
- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.

- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.
- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.

- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.

- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.

- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

  ```
  f arg1 arg2 = ...  where
     x = fst arg1
     y = snd arg1
     z = fst arg2
  ```

  Use this instead:

  ```
  f (x,y) (z,_) = ...
  ```

- Combine nested case expressions.
  For example, do not use this:

  ```
  case x of
        Red -> case y of
                 Red  -> True
                 Blue -> False
        Blue -> case y of
                 Red  -> False
                 Blue -> True
  ```

Use this instead:

```
case (x,y) of
      (Red,   Red) -> True
      (Blue, Blue) -> True
      (   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.