Log Message Format

March 6, 2024

1 Introduction

In this assignment, we will discuss the format of log messages consisting of different types of log entries. Each log message starts with a character indicating its type, followed by additional information.

2 Log Message Format

The log message format consists of the following components:

- **I**: Informational messages
- **W**: Warning messages
- **E**: Error messages

Error messages include an integer indicating the severity level, ranging from 1 to 100. Informational and warning messages do not include severity levels.

3 Example Log Messages

Here is a snippet of a 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
```

Certainly it will be necessary to write a program to understand the message and to be able to solve the problem. The following data types help to solve the problem:

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

```
type TimeStamp = Int

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

4 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]
```

That function will parse the entire log file

5 Ordering the logs

We need to have the logs in order, to do so the following data can be used:

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

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

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

6 Excercise 2

Define the following 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.

7 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).

8 Exercise 4

Define the function:

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

which takes a sorted MessageTree and produces a list of all the LogMessages 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)
```

9 Exercise 5

It is necessary to have the relevant messages extracted, 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.
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
```

There are four errors, three of which have a severity of greater than 50. The output of whatWentWrong 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.