August Exam (Scala): Chat Log Mining

This coursework is worth 50%. It is about mining a log of an online chat between 85 participants. The log is given as a csv-list in the file `log.csv`. The log is an unordered list containing information which message has been sent, by whom, when and in response to which other message. Each message has also a number and a unique hash code.

Important: Make sure the file you submit can be processed by just calling

```
scala <<filename.scala>>
```

Do not use any mutable data structures in your submission! They are not needed. This means you cannot use `ListBuffer`s, `Array`s, for example. Do not use `return` in your code! It has a different meaning in Scala, than in Java. Do not use `var`! This declares a mutable variable.

Disclaimer

It should be understood that the work you submit represents your own effort! You have not copied from anyone or anywhere else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Background

The fields in the file `log.csv` are organised as follows:

```
counter, id, time_date, name, country, parent_id, msg
```

Each line in this file contains the data for a single message. The field `counter` is an integer number given to each message; `id` is a unique hash string for a message; `time_date` is the time when the message was sent; `name` and `country` is data about the author of the message, whereby sometimes the authors left the country information empty; `parent_id` is a hash specifying which other message the message answers (this can also be empty). `Msg` is the actual message text. **Be careful** for the tasks below that this text can contain commas and needs to be treated special when the line is split up by using `line.split(",").toList`. Tasks (2) and (3) are about processing this data and storing it into the `Rec`-data-structure, which is pre-defined in the file `resit.scala`:

```
Rec(num: Int,
    msg_id: String,
    date: String,
    msg: String,
```

```
author: String,
country: Option[String],
reply_id : Option[String],
parent: Option[Int] = None,
children: List[Int] = Nil)
```

The transformation into a Rec-data-structure is a two-step process where first the fields for parents and children are given default values. This information is then filled in in a second step.

The main information that will be computed in the tasks below is from which country authors are and how many authors are from each country. The last task will also rank which messages have been the most popular in terms of how many replies they received (this will computed according to be the number children, grand-children and so on of a message).

Tasks

(1) The function `get_csv` takes a file name as argument. It should read the corresponding file and return its content. The content should be returned as a list of strings, namely a string for each line in the file. Since the file is a csv-file, the first line (the header) should be dropped in the result. Lines are separated by `"\n"`. For the file `log.csv` there should be a list of 680 separate strings.

[5% Marks]

(2) The function `process_line` takes a single line from the csv-file (as generated by `get_csv`) and creates a Rec(ord) data structure. This data structure is pre-defined in the Scala file.

For processing a line, you should use the function

```
<<some_line>>.split(",").toList
```

in order to separate the fields. HOWEVER BE CAREFUL that the message text in the last field of `log.cvs` can contain commas and therefore the split will not always result in a list of only 7 elements. You need to concatenate anything beyond the 7th field into a single string before assigning the field `msg`.

[10% Marks]

(3) Each record in the log contains a unique hash code identifying each message. For example

```
"5ebeb459ac278d01301f1497"
```

Some messages also contain a hash code identifying the parent message (that is to which question they reply). The function `post_process` fills in the information about potential children and a potential parent message.

The auxiliary function `get_children` takes a record `e` and a record list `rs` as arguments, and returns the list of all direct children (children have the hash code of `e` as `reply_id`). The list of children is returned as a list of `nums`. The `nums` can be used later as indexes in a Rec-list.

The auxiliary function `get_parent` returns the number of the record corresponding to the `reply_id` (encoded as `Some` if there exists one, otherwise it returns `None`).

In order to update a record, say `r`, with some additional information, you can use the Scala code

```
r.copy(parent = ....,
       children = ....)
```

[10% Marks]

(4) The functions `get_countries` and `get_countries_numbers` calculate the countries where message authors are coming from and how many authors come from each country (returned as a `Map` from countries to Integers). In case an author did not specify a country, the empty string should be returned.

[10% Mark]

(5) This task identifies the most popular questions in the log, whereby popularity is measured in terms of how many follow-up questions were asked. We call such questions as belonging to a *thread*. It can be assumed that in `log.csv` there are no circular references, that is no question refers to a follow-up question as parent.

The function `ordered_thread_sizes` orders the message threads according to how many answers were given for one message (that is how many children, grand-children and so on one message has).

The auxiliary function `search` enumerates all children, grand-children and so on for a given record `r` (including the record `r` itself). `Search` returns these children as a list of `Rec`s.

The function `thread_size` generates for a record, say `r`, a pair consisting of the number of `r` and the number of all children as produced by search. The numbers are the integers given for each message—for `log.cvs` a number is between 0 and 679.

The function `ordered_thread_sizes` orders the list of pairs according to which thread in the chat is the longest (the longest should be first).

[15% Mark]