COMPSCI4021 Coursework: Wikipedia Scraper

Jeremy Singer

Handout: 13 Nov 2019*

1 Introduction

This exercise will give you some experience in developing and testing a complete, small-scale application in the Haskell language, while making use of standard build tools and third party Haskell libraries. You will begin by producing a library that scrapes a wikipedia web page and identifies most frequent word on that page. You will go on to identify the most frequent words on the pages for several countries, and displaying this information in a programmatically generated visual form. Finally you will write a short report to summarize your experience with functional programming during this development exercise.

1.1 Intended Learning Outcomes

The key learning outcomes we cover in this coursework exercise are:

- Demonstrate understanding of how to structure programs using monads, how to use the most common standard monads (including IO, Maybe, and State), and how to use a monad transformer.
- Develop substantial software applications including GUIs and system interaction.

However, developing your own Haskell code should consolidate all the other learning outcomes from earlier in the course. The best way to learn a programming language is to read and write code in that language.

2 Tasks

There are three distinct tasks you should complete for this assessed exercise.

2.1 Task 1: Wikipedia Scraping Library

You should use the *Scalpel* library (introduced in lecture 12) to build a module named `WikiScrapeLib` with an exported function:

```
mostfrequentwordonpage :: URL -> IO (Maybe String)
```

Recall that URL is an alias for the String type, defined in Text.HTML.Scalpel.Internal.Scrape.URL. The function `mostfrequentwordonpage` will evaluate to `IO Nothing` when the URL cannot be fetched. When a page HTML can be fetched, the function parses the plain text (i.e. excluding HTML/CSS/JavaScript) in the body of the page and counts word frequencies, identifying the word that occurs most frequently on this page. A supplied file `stopwords.txt` contains a list of *stop words* which should be ignored in your analysis of the plain text — these are common words like `the` and `and`. Your program should read the list of stop words from the supplied text file. Other words to be treated as stop words are single letter words, the sequence 's at the end of a word, words containing non-alphabetic characters (e.g. dates like 2019), and any word that starts with the first four letters of the article name — e.g. for the Wikipedia article on *Scotland*, you should ignore all of: Scotland, Scottish, Scots, etc. Further, punctuation characters should be ignored and all words should be normalized to lower case. If there are *no* uncommon words (i.e. the page consists entirely of stop words) then the function evaluates to `IO Nothing`. If one word is most frequent, then this word is returned. If multiple words have equal frequency then any one of these words may be returned, wrapped up as an `IO Just "string"` value.

*version `bf4f02fc226067c887a2d5418b30802bed4ed077` on `master`

You may develop internal helper functions in your WikiScrapeLib module. However only the `mostfrequentwordonpage` function should be visible externally.

You may use types, typeclasses and functions from Prelude, Scalpel, Text.String, Data.List, etc — reasonable use of appropriate base utilities and other Hackage libraries is encouraged. You should document these dependencies in your YAML config files.

This task is worth **40% of the overall mark**.

2.2 Task 2: Information Representation Program

Write an application (with a top-level `main`) that uses your WikiScrapeLib module to visualize the most frequent word in the following English language versions of these Wikipedia pages:

1. Scotland
2. England
3. United_Kingdom
4. USA
5. Brazil
6. France
7. Germany
8. Italy
9. Japan
10. China
11. Russia

You should access the relevant page with the URL prefix `https://en.wikipedia.org/wiki/` and then the country name.

Your visualization should be an output file from the program — it could be a TXT file, a PDF file, or an image file like a JPG or PNG. You can use any libraries on hackage to help generate your visualization. Example visualizations and their corresponding libraries might include:

- A PDF containing a table generated with the pandoc or HaTeX libraries
- A neatly typeset PDF containing a well-rendered tag cloud or similar, using the Pango library
- A static HTML webpage containing the data, generated with the hakyll library
- A CSV file output containing the data, generated with a CSV processing library
- a graph (of some kind) generated with haskell-chart
- Native language output for each Country and Word, using the Google translate API binding in Haskell
- something else ...

There is real scope for creativity and innovation here. However, you should be aware of some constraints. First, I want to be able to build and execute your code via stack on a networked Virtual Box Linux instance, so you will need to have an appropriately configured project files (i.e. project.yaml and stack.yaml).

Second, this part of the coursework is worth **20% of the overall mark**.

2.3 Task 3: Reflective Report

Write a short (500 word max) report, as a plaintext file called report.txt. In this document, you should summarize your experiences of developing this project code in Haskell. How is it similar to more mainstream languages like Python or Java? How is it different? Are there any Haskell-specific features you really like? Why? Are there any features you intensely dislike? Why?

This task is worth **40% of the overall mark**.

3 How to Proceed

Clone (or fork) the github repository for this project from `https://github.com/jeremysinger/cw-scraper`. This will give you an initial working project setup, configured for the Stack tool. You will need to do a `stack build` to get things downloaded etc. Note I have included some skeleton source code and unit tests for you already. Run the unit tests with a `stack test`. You should try to keep the same project structure as much as possible, since it will make things easier for me to mark. In particular, I have some 'hidden' unit tests that rely on the name of the module and its exported function remaining the same.

Since the majority of the marks are available for tasks 1 and 3, you are recommended to focus on these tasks. Task 2 (the open-ended coding exercise) is almost an 'optional extra' if you are enjoying Haskell.

Overall, this coursework is worth 20% of the final mark for COMPSCI 4021 (Functional Programming). As such, we expect you will spend *no more than 20 hours* working on this exercise. The exam makes up the remaining 80% of the final mark.

4 What to Hand in

Please submit your coursework via moodle, on the *Scraping Coursework* Assignment submission slot on the FP(H) moodle page. Please submit a single zip file named according to your matriculation number, e.g. 2123456x.zip - when I unzip this file I want to see a top-level directory called cw-scraper that is a stack project directory.

The stack project should be *clean*ed and *purge*d - i.e. I don't want any hidden directories containing massive binaries. I want you to submit Haskell source files (for tasks 1 and 2), YAML config files, an output file from your task 2, and a report.txt file from your task 3. You should also submit a status.txt file explaining how far you got on each sub-task, what you did for task 2, whether and where the sample output file is included in the zip, and whether there are any issues I should be aware of during marking. There are skeleton .txt files in the git repo you will clone/fork when you start the project — edit these and leave them in the same locations. The diagram below gives the expected directory hierarchy layout inside the zip file.

```
cw-scraper
├── project.yaml
├── stack.yaml
├── cw-scraper.cabal
├── report.txt
├── status.txt
├── stopwords.txt
├── Setup.hs
├── README.md
├── app
│   ├── Main.hs
│   └── (other haskell files from Task 2)
├── src
│   ├── Lib.hs
│   └── (other haskell files from Task 1)
└── test
    ├── Spec.hs
    └── (other unit test files, optional)
```

Please adhere to these submission guidelines. Include your name and matriculation number in the status.txt file, also in the project.yaml file. Email me if anything is unclear. Rapid and happy marking will take place if it's straightforward for me to assess your work. Slow and grumpy marking will take place otherwise!

5 Marking Guidelines

5.1 Task 1: Wikipedia Scraping Library

This task is worth 40% of the overall mark for this coursework.

A	Code compiles with no errors, minimal warnings. Hidden unit tests pass ok. Idiomatic Haskell code. Highly efficient implementation.
B	Code compiles with no errors. Substantial majority of hidden unit tests pass. Fairly idiomatic Haskell code. Efficient implementation.
C	Code may compile, given minimal intervention. Majority of hidden unit tests pass. Some idiomatic Haskell code. Somewhat efficient implementation.
D	Code may compile, given some intervention. Some hidden unit tests pass. Potentially idiomatic Haskell code. Potential for some inefficiencies.
E–F	Code may not compile. Few hidden unit tests pass. Little or no idiomatic Haskell code. Code is inefficient or incorrect.
G–H	Code does not compile. No hidden unit tests pass. No idiomatic Haskell code. No attention given to efficiency of code execution.

5.2 Task 2: Information Representation Program

This task is worth 20% of the overall mark for this coursework. It should be viewed as an 'optional extra'.

A	Novel library. Striking output. Clean, compilable code. Idiomatic Haskell. Intelligent use of library and utility functions. Efficient code.
B	Appropriate library. Sensible output. Fairly clean, compilable code. Moderately idiomatic Haskell. Some appropriate use of library and utility functions. Moderately efficient code.
C	Simple but relevant library. Some meaningful output. Code compiles with minimal intervention. Attempt to write in functional style. Minimal use of library and utility functions. Perhaps inefficient code.
D	Simple but somewhat relevant library. Attempt to produce meaningful output. Code might compile, with some intervention. Attempt to write in functional style. Possible use of library and utility functions. Inefficient code.
E–F	Trivial library. Simple output. Code submitted may not compile. Non-idiomatic Haskell.
G–H	No meaningful engagement with third-party Haskell modules. Small codebase. No serious attempt to generate useful output.

5.3 Task 3: Reflective Report

This task is worth 40% of the overall mark for this coursework.

A	Highly readable. Excellent contrasts between Haskell and other languages/toolchains. Intelligent assessment of strengths and weaknesses of different language paradigms.
B	Readable. Very good contrasts between Haskell and other languages/toolchains. Pragmatic assessment of strengths and weaknesses of different language paradigms.
C	Moderately readable. Good contrasts between Haskell and other languages/toolchains. Some assessment of strengths and weaknesses of different language paradigms.
D	Somewhat disjointed. Some contrasts between Haskell and other languages/toolchains. Attempted assessment of strengths and weaknesses of different language paradigms.
E–F	Disjointed. Minimal contrasts between Haskell and other languages/toolchains. Minimal assessment of strengths and weaknesses of different language paradigms.
G–H	Largely unreadable. Little contrast between Haskell and other languages/toolchains. No meaningful assessment of strengths and weaknesses of different language paradigms.

6 Frequently Asked Questions

Q: When is the submission deadline? Check on the FP moodle page in case of changes.. At the time of writing, the deadline is set at 4:30pm on **Monday 2nd December 2019**.

Q: Can I implement my own unit tests? Yes, this might be sensible although it is not mandatory. However be aware that I will assess the correctness of your code with *my* unit tests which I haven't released to you.

Q: Can I have multiple attempts at Task 2, with different libraries to generate different outputs? You could have several attempts at Task 2, but this might not be an efficient use of your time. I will only mark *one* of your attempts, so you must indicate what you want me to mark . . . make this clear in your status.txt file.

Q: My code doesn't compile. Should I still submit what I have done? Yes. Partial credit is awarded for code that does not compile or run. It's always better to submit something rather than nothing. Mention your difficulties in your status.txt file when you submit.

Q: Does it matter if my report.txt is longer than 500 words? I don't promise to read it all, if it is too long. So you need to either (a) front-load all the interesting comments into the first 500 words, (b) trim it to 500 words, or (c) hope that I am hooked by your fascinating findings and will read on past the first 500 words.

Q: I really like my solution to the coursework. Can I post it on github? Please refer to the School policy on uploading your coursework to github. Inah or Gethin can give you more info on this.

Q: I am *very* confused about Haskell. What should I do? Contact me by email for some advice or pointers about how to get started. Don't worry—everyone feels like this at some stage!

Q: Can I use a non-Haskell library or service to generate my beautiful visualization for Task 2? Well that is possible, but remember that marks are only awarded for Haskell code. So I would have to assess the elegance of your Haskell foreign function calls or proxy stub code, which might not be enough to get you an excellent grade.

Q: Can we work in groups to develop our code? No, code should be developed individually. It's fine to chat to other people about your ideas, but please don't share code. The School of Computing Science has strict policies on plagiarism. This restriction also applies to reusing verbatim code you copy from online sources.