M21274 – MATHFUN
Discrete Mathematics and Functional Programming

Functional Programming Assignment 2020/21

Introduction

This assignment aims to give you practice in using the constructs and techniques covered in the functional programming lectures and practicals by developing a complete Haskell program. This work will be marked out of 50 and carries 40% of the module's marks.

You need to submit your program via the module's Moodle site by the deadline of **11pm, Friday 30th April 2021** and are required to demonstrate your program in a practical session between 4th and 13th May. You will need to sign up for a demonstration time on a Google spreadsheet, the link for which will be distributed via email. All marks will be allocated in the demonstrations, so you must attend. If you miss your demonstration, it is your responsibility to arrange an alternative demonstration with me. If you submit or demonstrate your work late, your mark for this assignment will be capped according to University rules. Feedback and the mark for this assignment will be returned to you via email.

Your task

Your task is to write a program in Haskell for querying and updating population figures for a list of major European cities. Each city has a name, a location expressed in degrees north and degrees east, and a list of metropolitan area population figures expressed in thousands of inhabitants: the first value in the list is the current population, the second is the population a year ago, the third is the population two years ago, etc. The list of cities is kept in alphabetical order. You can assume that the population lists have a common length of at least two.

Core functionality [28 marks]

The program should include pure functional (i.e. non-I/O) code that performs the following items of functionality:

i. Return a list of the names of all the cities
ii. Given a city name and a number, return the population of the city that number of years ago (or "no data" if no such record exists); the returned value should be a string representing the population in millions to 3 decimal places (e.g. "9.123m")
iii. Return all the data as a single string which, when output using `putStr`, will display the data formatted neatly into five columns giving the name, location (degrees N & E), this year's population and last year's population. (The populations should be formatted as for (ii).)
iv. Update the data with a list of new population figures (one value for each city); this should increase the length of each of the cities' population lists so that what was the current figure now becomes last year's figure, and so on.
v. Add a new city to the list preserving its alphabetical ordering; the new city should have a population list of length equal to those of the other cities.

vi. For a given city name, return a list of annual percentage population growth figures for that city (i.e., the result list should begin with the percentage increase from last year's figure to this year's; the second value should give the increase from two years ago to last year, etc.). The list will include negative values for shrinking populations.

vii. Given a location and a number, return the name of the closest city with a population bigger than the number, or "no city" if there are no such cities; use Pythagoras' theorem to calculate the distance between locations (i.e. assume the world is flat!)

Each item is worth 4 marks. You should begin by developing purely functional code to cover this core functionality, and then add to this the functions/actions that will output a population map and which comprise the program's user interface (see below).

I recommend that you attempt the above items in order – the first few should be easier than those at the end. If you can't complete all seven items of functionality, try to ensure that those parts that you have attempted are correct and as well-written as possible.

City map [6 marks]

Your program should also allow the user to plot a map of all the cities with their current population figures. Your code should assume that the terminal (shell) window is 80 characters wide and 50 lines long, and should plot, as neatly as possible, the location of each city with a '+' together with its name and current population figure (formatted as for (ii) above) so that the map makes good use of the space available. You should use the functions provided in the template.hs file to help draw the map (see below).

Starting point

Begin by copying and renaming the template.hs file from Moodle; your code should be developed within this file. Your first task will be to decide on a data representation `City` for individual cities and their location and population data. The list of city data will then be of type `[City]`. Now, for example, the functions to perform items (iii) and (iv) above might have the types:

```
citiesToString :: [City] -> String
updatePopulations :: [City] -> [Int] -> [City]
```

where `citiesToString cities` gives a well formatted string version of the `cities` data, and `updatePopulations cities newPopulations` gives a modified version of the `cities` data which includes the `newPopulations` data. You may find it useful to define a few additional "helper" functions to aid in the writing of the program. You may also find functions in the `Data.List` and `Text.Printf` modules useful.

City data. There is a file data.txt containing the data on Moodle. Your program is not required to process this file directly. Instead, you should copy and edit the data so that it is a valid value of type `[City]` given your particular `City` type definition. Include it in your program file as follows:

```
testData :: [City]
testData = [ ... the 12 city values ... ]
```

to allow for easy testing as you develop the program's functionality. It is this data that we will use to test your program, so it is important that you include it fully and accurately. We will use a `demo` function (the structure of which is supplied in the template.hs program). You should replace all the text in this function by corresponding Haskell expressions (this has essentially been done for you for `demo 3`). We will execute `demo 1`, `demo 2` etc. to assess your program's functionality. Running `demo 8` should show your city map. Make sure that the `demo` function can be used to illustrate all implemented functionality, or you might lose marks.

User Interface and File I/O [8 marks]

Your program should provide a textual menu-based user interface to the above functionality, using the I/O facilities provided by Haskell. The user interface should be as robust as possible (it should behave appropriately given invalid input) and for those functions that give results, the display of these results should be well formatted.

Your program should include a `main` function (of type `IO ()`) that provides a single starting point for its execution. When the program begins, the list should be loaded from a cities.txt file, and all the cities' names should be displayed (i.e. operation (i) performed). Your program should then present the menu for the first time giving 9 options (for items (i)-(vii), to draw the map, and to exit the program).

Only when the user chooses to exit the program should the list be written back to the cities.txt file (at no other times should files be used). Saving and loading can be implemented in a straightforward manner using the `writeFile` and `readFile` functions together with `show` and `read` (which convert data to and from strings). It is important that your cities.txt file is in the same folder as your Haskell program and your program refers to the file by its name only (not its full path). This will ensure that your program will work when we run your program on a different account/machine. If your program fails to open your cities.txt file you will lose marks.

Code quality [8 marks]

We will award marks based on your code's completeness, readability and use of functional constructs. Good use of powerful functional programming concepts (e.g. higher-order functions and/or list comprehensions) to achieve concise readable code will be rewarded. (Note that the code for the user interface and file I/O will not be assessed for quality.)

Moodle submission

You should submit a zip file containing your Haskell program and cities.txt file via the module's Moodle site (Functional Programming assignment in the Assessments tab) by the deadline specified above. Make sure that your program file is named using your student number, and that it has a .hs suffix; for example, 123456.hs. If you have not implemented loading/saving then you should just upload your Haskell program without zipping it.

Make sure that your `testData` value and your submitted cities.txt file include an exact copy of the supplied data to allow us to test the program's correctness. If your program gives unexpected results due to incorrect data you will lose marks.

Demonstration

We will begin the demonstration by executing your `demo` function for each item of functionality, although we may edit the function first in order to change the tested values. Make sure that your `testData` value includes an exact copy of the supplied data allow us to test the program's correctness. If your program gives unexpected results due to incorrect data you will lose marks. We will then test you user interface functionality, and check that saving and loading of data works. Make sure that your cities.txt file includes an exact copy of the supplied data allow us to test this. We may ask you technical questions about your code.

You will receive your mark and written feedback via email immediately after your demo. It is your responsibility to notify us if this email is not received.

Important

This is individual coursework, and so the work you submit for assessment must be your own. Submitted programs will be checked electronically for possible plagiarism. Any attempt to pass off somebody else's work as your own, or unfair collaboration, is plagiarism, which is a serious academic offence. Any suspected cases of plagiarism will be dealt with in accordance with University regulations.

Matthew Poole
March 2021