COMP3941/5941 Functional Programming 2016

Coursework 2

30 marks (30% of module assessment)

Issue date: 16 November 2016, 10:00 am
Due date: 7 December 2016, 10:00 am

Overview

The aim of this coursework is to implement a word game similar to "Boggle[1]", where players attempt to form English words from letters formed from the top faces of a 4x4 grid of cubes. Each round of Boggle starts by shaking the grid, causing cubes to be re-arranged and a new set of letters to appear as the upper faces of the cubes. Each player then takes it in turns to form words. Any word must be formed by starting from one letter in the grid, and then moving to neighbouring letters (either vertically, horizontally, or diagonally). No letter on the grid can be used more than once (although there may be multiple copies of some letters available). For example, given the following grid,

```
y q a r
e v c i
a a g p
t d e n
```

the following words are possible: `ravaged`, `arcade`, `caveat`, `adage`, `caged`, `crave` .. (In fact I cheated here - I got the example grid and solutions by running my code and getting my computer opponent to find words!). The letter "q" requires some care, as English words require "q" to be followed by "u". As this is relatively unlikely to happen in a random grid of letters, the "u" is considered to be present implicitly. That is, the letter appearing as "q" on the board and in users' input is considered to be the substring "qu" when checking words and scoring turns.

Words are checked against an agreed dictionary. I have provided you with a list of English words in the file "words.txt". Words are scored using the following algorithm:

less than 4 letters in length	0 points
4 letters long	1 point
5 letters long	3 points
6 or more letters	the number of letters in the word.

[1] ™ Parker Bros.

Your tasks

The coursework 2 area on the VLE provides, in addition to this briefing, three files:

1. words.txt A list of English words
2. boggle An executable version of my implemented solution, so that you can play the game yourself[2]! Note that this is a linux executable, you will need to run it on the school's workstations.
3. boggle.hs A subset of my boggle implementation, containing many of the data types, type declarations for some (but not all) of the functions that I used, and a significant amount of code (my solution has 435 lines including comments, of which I am giving you 301 lines of code and comments!)

Your challenge is to complete a working implementation of the game. Specifically, you will need to implement the following five functions, whose existence is assumed in the remainder of the game:

a) makeGrid :: [Cube] -> Grid

b) checkWord :: Word -> Grid -> Bool

c) shake :: Grid -> IO Grid

d) score :: Dictionary -> Grid -> [Word] -> [Word] -> [(Word, Result)]

e) computerTurn :: Game -> IO Game

In order to implement these functions you may find it useful to define further data types and further functions to help. You are welcome to place these wherever you wish (as local definitions, or as top-level functions).

In terms of implementing the computer player, you may want to consider the following strategy. Given the existing game dictionary, construct an "inverse" dictionary that contains lists of words paired with a string containing the letters in the word, but sorted alphabetically and organised in decreasing order of size. For example, if a dictionary contains the words

`adage`, `arcade`, `caged`,`cave`, `caveat`, `crave`, `ravaged`
then the "inverse" dictionary would be as follows:

```
[ ("aadegrv", "ravaged")
, ("aacder", "arcade")
, ("aacetv", "caveat")
, ("aadeg", "adage")
, ("acdeg", "caged")
, ("acerv", "crave")
, ("acev", "cave") ]
```

[2] You are of course welcome to try and disassemble the code to reverse-engineer a solution. Good luck with navigating your way through GHC's runtime system and execution model!

Given an inverse dictionary, you can then find possible computer moves as follows:

1. take the game grid and extract the letters that lie on the top face of each cube.
2. sort this list of 16 letters
3. scanning through the inverse from the front, check the letters that are needed to make a word with the available letters on the grid; if the letters needed are not available in the grid, there is no need to consider that word any further.
4. assuming the word could possibly be made, use your checkWord function to see if the word can be formed legally from the letters on the grid. If it can, its a possible move by the computer.

The "checkWord" function is therefore important. Writing it is non-trivial. You may find it helpful to implement a helper function with the following type:

placeAt :: Word -> Grid -> [Square] -> Square -> Bool

Given a word w, a grid g, a list ss of already-used squares, and a starting square s, "placeAt w g ss s" returns true if and only if the word w can be placed at the position s without using any of the squares in s. It can be implemented using recursion over the list of letters in the word w. If the list is empty, the task is trivial. If it is non-empty, then check whether the first letter in w matches the selected square s, and that s isn't in the list of squares already in use. If either condition is false, you can't place the word there. If both conditions are true, then you can place the word if and only if you can place the remainder of the word starting from one of the squares adjacent to s and with s now included in the used squares.

Marking Scheme

Marks will be awarded both for correctness and for effective use of functional programming concepts and Haskell language features. The number of marks allocated for good functional style are shown in a separate column in the table below. As you can see, 2/3 of the mark are for writing code that works, 1/3 of the marks are for writing good code that works!

Task	Marks Available	Style Marks
makeGrid	2	0
checkWord	8	4
shake	6	2
score	4	2
computerTurn	10	4
TOTAL	30	10

Submission

Your code should be submitted via the Coursework 2 submission link on the VLE. The submission should consist of a single .hs file.

Your submission must be individual work, and your attention is drawn to the University's rules regarding plagiarism. Any suspected cases of plagiarism will be investigated, and if confirmed could result in serious penalties.

Questions: Any question about this coursework should be sent by email, to D.J.Duke@leeds.ac.uk. I will reply individually to all questions; any information to be broadcast to the class will again be sent via email.