CMPT 383: Assignment #1

Anders Miltner
miltner@cs.sfu.ca

Introduction

In this assignment, we will be using Haskell to implement the game "Spelling Bee."

In this game, the player is provided 7 letters, arranged in a hexagon.

The purpose of the game is simple, use the letters to build as many words as possible. You can use the same letter multiple times, and you need not include all letters. But there's one caveat: there is a special letter that you *must* include somewhere. Furthermore, the game is built in such a way that there is always exactly one word that uses all the letters.

You can run the game by running `stack build` and `stack exec a1-exe`. You can run a (partial) test suite by running `stack test`.

As per usual, you are able to use any function available without adding extra imports. For your benefit, we have manually included the implementation of some useful functions. You may want to look those over, as they will likely be helpful in this Assignment.

1 Initializing the Game State

Before we start playing the game, we first must find a puzzle. To do this, you must implement three functions: `toCandidateBasis`, `extractBases` and `basisToPuzzle`. The first function, `toCandidateBasis` turns a string into a *candidate basis*. A candidate basis is a list of 7 characters, put in order, with no duplicates. Thus, one might expect that the type of `toCandidateBasis` should be `String -> String`. The candidate basis corresponding to a given string should be the set of letters used in the string put in order and deduplicated. But remember, a candidate basis must be 7 letters long! If not exactly 7 distinct letters are used in the word, `toCandidateBasis` should return Nothing. If exactly 7 distinct letters are used, it should return Just those letters in alphabetical order (which is the default ordering on Chars). Thus, the type of `toCandidateBasis` is instead `String -> Maybe String`.

The type of extractBases is `[String] -> [String]`. The incoming string list corresponds to a *dictionary* of all words in the language. In the game, this will (by default) be loaded from `words.txt` – a list of the 3000 most common English words. You should create the list of all bases from this list. Recall, for a list of characters to a basis of a puzzle, it must be a valid basis (in other words have exactly 7 distinct characters in the list, in order), and there must be exactly one word corresponding to the [basis] in the dictionary.

Last to implement is basisToPuzzle. Given a basis, and a number from 0 to 6, return a puzzle. A puzzle consists of two parts, a Char, and a list of Chars. The first Char should be the character at the provided index. The list of Chars should be 6 Chars long, and should consist of all Chars in the basis except the one at the provided index.

2 Checking Word Correctness

Now that you have built up a puzzle, it's time to check the correctness of the provided answers. To do this, you need to build up a function for answering questions: `isWordCorrect`.

The `isWordCorrect` function takes as input a Dictionary, a Puzzle, and a string input. The function should return true exactly when both of the following conditions are met:

1. The provided string is must be in the dictionary

2. The provided string must contain only letters in the puzzle, and must contain at least one instance of the designated Char that is the fst element of the puzzle.

3 Providing Feedback to the User

Finally, you must return feedback to the user. There are two functions that must be filled out to do this: `allAnswers` and `finalScore`.

The `allAnswers` function calculates all the valid answers to the puzzle. Given a puzzle `p` and a dictionary `d`, `allAnswers` should return a list of strings that contain exactly one instance of every word `w` where `isWordCorrect d p w` returns `True`. After the player has stopped guessing, they will be provided the list of correct answers, along with their score.

The `finalScores` function rates how well a user did based on the proportion of correct answers that they guessed. If the user guessed 0 correct answers, return `Zero`. If the user guessed $0\% < p < 25\%$ of the correct answers, return `Bad`. If the user guessed $25\% \leq p < 50\%$, return `OK`. If the user guessed $50\% \leq p < 75\%$, return `Good`. If the user guessed $75\% \leq p < 100\%$, return `Great`. If the user guessed all the correct answers, return `Perfect`.

4 Cheating at the Game

I am unfortunately very bad at this game. Perhaps I can use programming to help! In this final exercise, we will cheat at the game. Given a predicate of type `Puzzle -> String -> Bool`, an Integer i, and a Puzzle, we wish to find all valid solutions to the puzzle up to and including i. Note that we do not have access to the dictionary for this problem.

For solving this problem, it may benefit you to create a helper function that simply finds all the possible guesses of size i.