COMP545 Assessment 2 – Herdle (Brief v1.0.3)

1 Overview

The aim of this assignment is to implement *Herdle*, a clone of the game *Wordle* but in Haskell.

Wordle is a single-player game with the following rules: each day a five-letter English word is chosen randomly (by the computer). A player has six attempts to guess the word by inputting five letters. If the player guesses the word correctly, they win. Otherwise, they are given feedback for each letter of their guess which indicates whether the letter is in the word and in the right place, is present in the actual word but in a different position, or is not in the word at all.

For example, if the actual word is `enter` and the player guesses `green`, they are told that:

- `g` occurs nowhere in the word;
- `r` occurs elsewhere in the word;
- `e` occurs elsewhere in the word;
- `e` occurs here in the word;
- `n` occurs elsewhere in the word.

Then on their next turn the player can use this information to make a better guess.

You can play the real game at `https://www.powerlanguage.co.uk/wordle/`.

We will build up our own version of this which can be played from the command line.

Provided files This `Base.hs` module provides some key data types for the game and some helper functions for outputting information to the user. Download this from the Moodle, put it in the same directory as your work, and read through this file and its comments.

The main data types provided are:

- `Status` used to give the status of letters in a guess;
- `Direction` used in `Status` to give optional information on where a letter occurs in a word;
- `Prompt` used to represent responses shown to the player on the command line.

At the top of any Haskell solution files add `import Base` to access these definitions.

Guidelines and submissions instructions The assessment is organised into five parts.

Parts A to D are organised such that you can develop the game iteratively, progressively making it richer in its features and game play. Each part is broken up into three questions which correspond to three iterations.

A good strategy is to complete iteration 1 in each of parts A to D first, which will produce a working game. If you do this all correctly then you've likely passed. The subsequent iterations make the game closer to the described game play and with more features. Part E is an extension.

The appendix gives example runs of the game for the three iterations.

You need **not** submit each iteration, but you may. If you have confidently completely iteration N, then you can just submit this. If however, you have only partially completed iteration N, then feel free to submit separate files for earlier iterations (e.g., `iteration1.hs`, `iteration2.hs`, etc.) which can then be individually marked.

- Submit your files as a `.zip` with your login ID, e.g. `dao7.zip`, including `Base.hs` though you shouldn't need to modify it;
- You may define as many extra functions as you need for your design.
- You may use any library functions you want.
- Your work must be your own. We will run plagiarism checks on the code which can detect changes in names and whitespace.

2 Questions

Part A: Helper functions

This part introduces some basic helper functions used by the later parts.

(**Iteration 1**) Define a function:

```
showStatus :: [Status] -> String
```

which maps a list of status data to a string describing the status of a guess to the user, where:

- `Here` is mapped to `Y`
- `Nowhere` is mapped to `-`
- `Elsewhere Nothing` is mapped to `y`
- `Elsewhere (Just Before)` is mapped to `<`
- `Elsewhere (Just After)` is mapped to `>`

and each symbol is separated by a space.

For example:

```
  showStatus [Nowhere, Here, Elsewhere (Just Before), Here, Nowhere]
= "- Y < Y - "
```

(5 marks)

(**Iteration 2**) Iteration two adds the idea of allowing players to only use letters that have not been ruled out by previous guesses. We will call these the "available" characters.

Define a function of type:

```
updateAvailable :: [Char] -> [(Char, Status)] -> [Char]
```

whose first input is a list of available characters and second input is a list of character-status pairs (i.e., from a guess). The function should produce the output list based on the first input but removing any character that is given a status of `Nowhere` by the second input.

For example:

```
updateAvailable "abcde" [('b', Nowhere), ('e', Here), ('d', Elsewhere Nothing)]
= "acde"
```

(6 marks)

(**Iteration 3**) Define `leftMargin :: String` which computes a string containing just spaces, whose length is that of the message corresponding to the `Start` prompt as output by the `prompt` function in the `Base` module.

(5 marks)

Part B: Core functionality

This part is about implementing the core checking of guesses against the actual word, getting progressively richer in the information it returns.

(Iteration 1) Define a function of type:

```
checkGuess :: String -> String -> [(Char, Status)]
```

where the first input is a guess and the second input is the actual word. The output should return a list of the guess characters (in order) paired with their status with respect to the actual word.

In this iteration, define `checkGuess` such that it marks guess letters only as either `Here` or `Nowhere`. For example:

```
  checkGuess "green" "enter"
= [('g',Nowhere),('r',Nowhere),('e',Nowhere),('e',Here),('n',Nowhere)]
  checkGuess "ender" "enter"
= [('e',Here),('n',Here),('d',Nowhere),('e',Here),('r',Here)]
```

(10 marks)

(Iteration 2) Introduce the idea of being able to mark letters as being `Elswhere` but without a direction (e.g., `Elsewhere Nothing`). For example:

```
  checkGuess "green" "enter"
= [('g',Nowhere),('r',Elsewhere Nothing),('e',Elsewhere Nothing)
  ,('e',Here),('n',Elsewhere Nothing)]]
```

Tip: you need some way of looking at the whole actual word to see if a letter appears in it, rather than just what is left of the actual word as you recursively decompose it.

One approach is an auxiliary function with an extra String input to hold a copy of the actual word, but which we do not recursively decompose so that it can be checked that a letter occurs elsewhere in the actual word. Other approaches are possible with nested functions.

(6 marks)

(Iteration 3) Adapt `checkGuess` so that it now includes the 'direction' information for letters which are elsewhere in the word, i.e., whether they appear before or after the current position.

For example:

```
  checkGuess "green" "enter"
= [('g',Nowhere),('r',Elsewhere (Just After)),('e',Elsewhere (Just Before))
  ,('e',Here),('n',Elsewhere (Just Before))]
```

If a guess letter occurs both before and after its current position, then it should be reported as `Before`. For example, in the following the `e` in the guess occurs both before and after the current position in the actual word, so `Before` is reported:

```
  checkGuess "pearl" "enter"
= [('p',Nowhere),('e',Elsewhere (Just Before)),('a',Nowhere)
  ,('r',Elsewhere (Just After)),('l',Nowhere)]
```

(8 marks)

Part C: User interface - Receiving guesses

(**Iteration 1**) Define a function:

```
getGuess :: Int -> IO String
```

which reads a number of characters from standard input using `getChar' :: IO Char` (this is imported from `Base` and is a cross-platform versin of `getChar`) and returns the string that was input. The number of characters to receive from the user is given by the first argument.

Between each character input by the user, print a space using `putChar :: Char -> IO ()` and after the required number of characters has been read (as given by the first input) then print a newline character before returning. The returned string should contain only lowercase characters.

For example, you should be able to replicate the following behaviour from inside `ghci` (where I have typed the five letters shown on the second line):

```
*Main> getGuess 5
a b c d e
"abcde"
*Main> getGuess 3
A b C
"abc"
```

(10 marks)

(**Iteration 2**) Add another input to `getGuess` so that it takes the list of "available" characters, i.e., those characters which have not been ruled out as appearing nowhere in the word. If a player tries to type a character which is not available then it should be erased by putting the backspace character `'\b'`.

For example, running `getGuess 5 "abcdefg"`, a user trying to input anything but the characters in the second input should have the input letter immediately erased, having the effect that they cannot type that letter and have to guess another.

(5 marks)

(**Iteration 3**) Whilst guessing, if a user inputs a full-stop character '`.`' then delete the previously input letter by outputting the appropriate amount of backspace characters.

The user should not be able to delete more than the number of letters guessed so far, and deleting letters means they can guess that letter again.

(5 marks)

Part D: User interface - Overall gameplay

(**Iteration 1**) Define a recursive function of type:

```
loop :: String -> Int -> IO ()
```

whose inputs are the actual word for the game and the number of attempts remaining and which performs the following IO behaviour:

- If the number of attempts is 0 then output the prompt message for `Lose` (i.e., use the `prompt` function from `Base` to get the string).
- Otherwise:
  - Get a guess of five letters from the user;
  - Check the guess against the actual word (first parameter to `loop`) and print the status to the user using your `showStatus` function;
  - If the status of all the letters is `Here` then show the `Win` prompt and return, otherwise recursively call `loop` with one fewer attempts.

Also define a function `go :: String -> IO ()` which takes a word and starts the `loop` with 6 attempts and mapping all letters in the input word to be lowercase. (15 marks)

(**Iteration 2**) Redefine `loop` to now take the current list of available letters, i.e., of type:

```
loop :: String -> [Char] -> Int -> IO ()
```

where the list of available letters in a recursive call is updated based on any guess, using `updateAvailable` from part A.

The `go` function should now call `loop` with a list with all the letters of the alphabet as the initial list of available letters. (5 marks)

(**Iteration 3**) Adapt `loop` so that:

- The number of attempts is recorded as a parameter to `loop` and the attempt number is reported at the start of each round.
- The user is then shown the `Start` prompt and is allowed to chose to either to make a guess by pressing any key, or quit the game by pressing 'q'.
- If they choose to make a guess (any key), then they are shown the `Guess` prompt, prior to the call to `getGuess`.
- The status message should now be indented by `leftMargin` spaces from part A.

(10 marks)

Part E - Extension

Create a word list file (e.g., `wordlist.txt`) which is read at the start of the game and from which a word is chosen based on the current day.

You will need to research how to read a file and split its lines into a list of strings, and how to get the current day.

Provide a `main :: IO ()` function to start the game, which will allow the game to be compiled and run stand-alone. (10 marks)

Appendix: Example runs of each iteration

Iteration 1

```
*Main> go "enter"
g r e e n
- - - Y -

e n d e r
Y Y - Y Y

e n t e r
Y Y Y Y Y

You got it. Well done!
```

Iteration 2

```
*Main> go "enter"
g r e e n
- y y Y y

e n d e r
Y Y - Y Y

e n t e r
Y Y Y Y Y

You got it. Well done!
```

Iteration 3

```
*Main> go "enter"

Attempt 1
Guess [any] or quit [q]?
Ok. Enter your guess:    g r e e n
                         - > < Y <

Attempt 2
Guess [any] or quit [q]? e
Ok. Enter your guess:    e n d e r
                         Y Y - Y Y

Attempt 3
Guess [any] or quit [q]?
Ok. Enter your guess:    e n t e r
                         Y Y Y Y Y
You got it. Well done!
```