The University of Melbourne
Department of Computing and Information Systems

Declarative Programming
COMP30020/COMP90048

Semester 2, 2017

Project Specification

Project due Thursday, 31 August 2017 at 5pm
Worth 15%

The objective of this project is to practice and assess your understanding of functional programming and Haskell. You will write code to implement the guessing part of a logical guessing game.

The Game of ChordProbe

ChordProbe is a two-player logical guessing game created for this project. You will not find any information about the game anywhere else, but it is a simple game and this specification will tell you all you need to know.

For a ChordProbe game, one player will be the *composer* and the other is the *performer*. The composer begins by selecting a three-pitch musical chord, where each pitch comprises a musical *note*, one of *A, B, C, D, E, F,* or *G*, and an *octave*, one of *1, 2,* or *3*. This chord will be the *target* for the game. The order of pitches in the target is irrelevant, and no pitch may appear more than once. This game does not include sharps or flats, and no more or less than three notes may be included in the target.

Once the composer has selected the target chord, the performer repeatedly chooses a similarly defined chord as a *guess* and tells it to the composer, who responds by giving the performer the following *feedback*:

1. how many pitches in the guess are included in the target (correct pitches)
2. how many pitches have the right note but the wrong octave (correct notes)
3. how many pitches have the right octave but the wrong note (correct octaves)

In counting correct notes and octaves, multiple occurrences in the guess are only counted as correct if they also appear repeatedly in the target. Correct pitches are not also counted as correct notes and octaves. For example, with a target of *A1, B2, A3*, a guess of *A1, A2, B1* would be counted as 1 correct pitch (*A1*), two correct notes (*A2, B1*) and one correct octave (*A2*). *B1* would not be counted as a correct octave, even though it has the same octave as the target *A1*, because the target *A1* was already used to count the guess *A1* as a correct pitch. A few more examples:

Target	Guess	Answer
A1,B2,A3	*A1,A2,B1*	1,2,1
A1,B2,C3	*A1,A2,A3*	1,0,2
A1,B1,C1	*A2,D1,E1*	0,1,2
A3,B2,C1	*C3,A2,B1*	0,3,3

The game finishes once the performer guesses the correct chord (all three pitches in the guess are in the target). The object of the game for the performer is to find the target with the fewest possible guesses.

The Program

You will write Haskell code to implement the *performer* part of the game. This will require you to write a function to return your initial guess, and another to use the feedback from the previous guess to determine the next guess. The latter function will be called repeatedly until it produces the correct guess. You will find it useful to keep information between guesses; since Haskell is a purely functional language, you cannot use a global or static variable to store this. Therefore, your initial guess function must return this game state information, and your next guess function must take the game state as input and return the updated game state as output. You may put any information you like in the game state, but you *must* define a type `GameState` to hold this information. If you do not need to maintain any game state, you may simply define `type GameState = ()`.

You may use any representation you like for notes, octaves, pitches, and chords internally, and may use this representation inside your `GameState` type. However, to avoid prejudicing your choice of representations, we use a very simple representation for the inputs and outputs of your functions. A chord is represented as a list of two-character strings, where the first character is an upper case letter between `'A'` and `'G'` representing the note, and the second is a digit character between `'1'` and `'3'` representing the octave.

You must define following functions:

initialGuess :: ([String],GameState)
: takes no input arguments, and returns a pair of an initial guess and a game state.

nextGuess :: ([String],GameState) → (Int,Int,Int) → ([String],GameState)
: takes as input a pair of the previous guess and game state, and the feedback to this guess as a triple of correct pitches, notes, and octaves, and returns a pair of the next guess and game state.

You must call your (main) source file `Proj1.hs` (or `Proj1.lhs` if you use literate Haskell), and it must contain the module declaration:

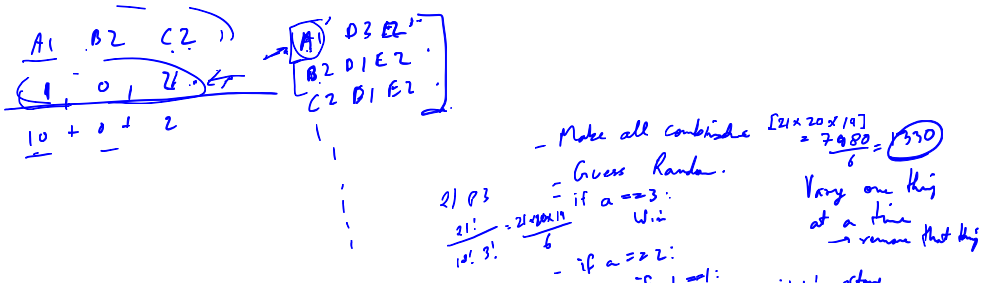
```
module Proj1 (initialGuess, nextGuess, GameState) where
```

You may divide your code into as many files as you like, as long as your main file (and the files it imports) imports all the others. But do not feel you need to divide your program into many files if it is reasonably small.

I will post a test driver program `Proj1Test.hs`, which will operate similarly to how I actually test your code. I will compile and link your code for testing using the command:

```
ghc -O2 --make Proj1Test
```

or similar. To run `Proj1Test`, give it the target as three separate command line arguments, for example `./Proj1Test D1 B1 G2` would search for the target `["D1", "B1", "G2"]`. It will then use your `Proj1` module to guess the target; the output will look something like:



```
Your guess 1:  ["A1","B1","C2"]
My answer:  (1,0,2)
Your guess 2:  ["A1","D1","E2"]
My answer:  (1,0,2)
Your guess 3:  ["A1","F1","G2"]
My answer:  (1,0,2)
Your guess 4:  ["B1","D1","G2"]
My answer:  (3,0,0)
You got it in 4 guesses!
```

Assessment

Your project will be assessed on the following criteria:

70% Quality and correctness of your implementation;

30% Quality of your code and documentation

The correctness of your implementation will be assessed based on whether it succeeds in guessing the targets it is given in the available time. Quality will be assessed based on the number of guesses needed to find the given targets. Full marks will be given for an average of 4.3 guesses per target, with marks falling on a logarithmic scale as the number of guesses rises. Thus moving from taking 5 guesses to 4 will gain similar number of points as going from 7 to 5 guesses. Therefore as the number of guesses drops, further small decreases in the number of guesses are increasingly valuable.

Note that timeouts will be imposed on all tests. You will have at least 5 seconds to guess each target, regardless of how many guesses are needed. Executions taking longer than that may be unceremoniously terminated, leading to that test being assessed as failing. Your programs will be compiled with `GHC -O2` before testing, so 5 seconds per test is a very reasonable limit.

See the Project Coding Guidelines on the LMS for detailed suggestions for coding style. These guidelines will form the basis of the quality assessment of your code and documentation.

Submission

You must submit your project from the student unix server `dimefox.eng.unimelb.edu.au` or `nutmeg.eng.unimelb.edu.au`. Make sure the version of your program source files you wish to submit is on this server, then `cd` to the directory holding your source code and issue the command:

```
submit COMP90048 proj1 Proj1.hs
```

(substitute `Proj1.lhs` if you use literate Haskell). If your code spans multiple source files, add the extra ones to the end of that command line.

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify COMP90048 proj1 | less
```

This will show you the test results from your submission, as well as the file(s) you submitted. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `proj1.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again.

It is your responsibility to verify your submission.

Windows users should see the LMS Resources list for instructions for downloading the (free) Putty and Winscp programs to allow you to use and copy files to the department servers from windows computers. Mac OS X and Linux users can use the `ssh`, `scp`, and `sftp` programs that come with your operating system.

Late Penalties

Late submissions will incur a penalty of 0.5% of the possible value of that submission per hour late, including evening and weekend hours. Late submissions will incur a penalty of 0.5% per hour late, including evening and weekend hours. This means that a perfect project that is much more than 4 days late will receive less than half the marks for the project. If you have a medical or similar compelling reason for being late, you should contact the lecturer as early as possible to ask for an extension (preferably before the due date).

Hints

1. A very simple approach to this program is to simply guess every possible combination of pitches until you guess right. There are only 1330 possible targets, so on average it should only take about 665 guesses, making it perfectly feasible to do in 5 seconds. However, this will give a very poor score for guess quality.

2. A better approach would be to only make guesses that are consistent with the answers you have received for previous guesses. You can do this by computing the list of possible targets, and removing elements that are inconsistent with any answers you have received to previous guesses. A possible target is inconsistent with an answer you have received for a previous guess if the answer you would receive for that guess and that (possible) target is different from the answer you actually received for that guess.

   You can use your `GameState` type to store your previous guesses and the corresponding answers. Or, more efficient and just as easy, store the list of remaining possible targets in your `GameState`, and pare it down each time you receive feedback for a guess.

3. The best results can be had by carefully choosing each guess so that it is most likely to leave a small remaining list of possible targets. You can do this by computing for each remaining possible target the maximum number of possible targets it will leave if you guess it. This you can do by computing, for each remaining possible target, the answer you will receive if it is the actual target, and then compute how many of the remaining possible targets would yield the same output, and take the maximum of all of these. Alternatively, you can take a more probabilistic approach, and compute the

average number of possible targets that will remain after each guess, giving the *expected* number of remaining possible targets for each guess, and choose the guess with the smallest expected number of remaining possible targets.

4. Unfortunately, this is much more expensive to compute, and you will need to be careful to make it efficient enough to use. One thing you can do to speed it up is to laboriously (somehow) find the best first guess and hard code that into your program. After the first guess, there are much fewer possible targets remaining, and your implementation may be fast enough then.

5. You can also remove *symmetry* in the problem space. The key insight needed for this is that given any *guess* and an answer returned for it, the set of remaining possibilities after receiving that answer for that guess will be the same regardless of which target yielded that answer. This suggests collecting all the *distinct* answers for a given guess and for each answer, counting the number of targets that would give that answer. Since there are much fewer answers than possible targets, this can save significant work.

   For example, suppose there are ten remaining candidate targets, and one guess gives the answer (3,0,0), three others give (1,0,2), and the remaining six give the answer (2,0,1). In this case, if you make that guess, there is a 1 in 10 chance of that being the right answer (so you are left with that as the only remaining candidate), 3 in 10 of being left with three candidates, and a 6 in 10 chance of being left with six candidates. This means on average you would expect this answer to leave you with

   $$\frac{1}{10} \times 1 + \frac{3}{10} \times 3 + \frac{6}{10} \times 6 = 4.6$$

   remaining candidates. You just need to select a guess that gives the minimum expected number of remaining candidates.

   Also note that if you do this incorrectly, the worst consequence is that your program takes more guesses than necessary to find the target. As long as you only ever guess a possible target, every guess other than the right one removes at least one possible target, so you will eventually guess the right target.

6. Note that these are just hints; you are welcome to use any approach you like to solve this, as long as it is correct and runs within the allowed time.

Note Well:

This project is part of your final assessment, so cheating is not acceptable. Any form of material exchange between teams, whether written, electronic or any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. If you have questions regarding these rules, please ask the lecturer.