The Game

Proj2 is a simple 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.

The game is somewhat akin to the game of Battleship™, but somewhat simplified. The game is played on a 4×8 grid, and involves one player, the *searcher* trying to find the locations of three battleships hidden by the other player, the *hider*. The searcher continues to guess until they find all the hidden ships. Unlike Battleship™, a guess consists of *three different* locations, and the game continues until the exact locations of the three hidden ships are guessed in a single guess. After each guess, the hider responds with three numbers:

1. the number of ships exactly located;
2. the number of guesses that were exactly one space away from a ship; and
3. the number of guesses that were exactly two spaces away from a ship.

Each guess is only counted as its closest distance to any ship. For example if a guessed location is exactly the location of one ship and is one square away from another, it counts as exactly locating a ship, and not as one away from a ship. The eight squares adjacent to a square, including diagonally adjacent, are counted as distance 1 away. The sixteen squares adjacent to those squares are considered to be distance 2 away, as illustrated in this diagram of distances from the center square:

2	2	2	2	2
2	1	1	1	2
2	1	0	1	2
2	1	1	1	2
2	2	2	2	2

Of course, depending on the location of the center square, some of these locations will actually be outside the board.

Note that this feedback does not tell you *which* of the guessed locations is close to a ship. Your program will have to work that out; that is the challenge of this project.

We use a chess-like notation for describing locations: a letter *A*–*H* denoting the column of the guess and a digit *1*–*4* denoting the row, in that order. The upper left location is *A1* and the lower right is *H4*.

A few caveats:

- The three ships will be at three *different* locations.
- Your guess must consist of exactly three *different* locations.
- Your list of locations may be written in any order, but the order is not significant; the guess *A3*, *D1*, *H1* is exactly the same as *H1*, *A3*, *D1* or any other permutation.

Here are some example ship locations, guesses, and the feedback provided by the hider:

Locations	Guess	Feedback
H1, B2, D3	*B3, C3, H3*	0, 2, 1
H1, B2, D3	*B1, A2, H3*	0, 2, 1
H1, B2, D3	*B2, H2, H1*	2, 1, 0
A1, D2, B3	*A3, D2, H1*	1, 1, 0
A1, D2, B3	*H4, G3, H2*	0, 0, 0
A1, D2, B3	*D2, B3, A1*	3, 0, 0

Here is a graphical depiction of the first example above, where ships are shown as **S** and guessed locations are shown as **G**:

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
1								**S**
2		**S**						
3		**G**	**G**	**S**				**G**
4								

The game finishes once the searcher guesses all three ship locations in a single guess (in any order), such as in the last example above. The object of the game for the searcher is to find the target with the fewest possible guesses.

The Program

You will write Haskell code to implement both the *hider* and *searcher* parts 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(es) to determine the next guess. The former function will be called once, and then the latter function will be called repeatedly until it produces the correct guess. You must also implement a function to determine the feedback to give to the hider, given his guess and a target.

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 new 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 must also define a type `Location` to represent grid locations in the game, and you must represent your guesses as lists of `Location`s. Your `Location` type must be an instance of the `Eq` and `Show` type classes. Of course, two `Location`s must be considered equal if and only if they are identical. A `Location`

You must also define a type `Location` to represent grid locations in the game, and you must represent your guesses as lists of `Location`s. Your `Location` type must be an instance of the `Eq` and `Show` type classes. Of course, two `Location`s must be considered equal if and only if they are identical. A `Location` must be `show`n as a two-character string of the upper-case column letter and row numeral, as shown throughout this document. You must also define a function to convert a `Location` into a string.

What you must define

In summary, in addition to defining the `GameState` and `Location` types, you must define following functions:

toLocation :: String → Maybe Location
: gives `Just` the `Location` named by the string, or `Nothing` if the string is not a valid location name.

fromLocation :: Location → String
: gives back the two-character string version of the specified location; for any location *loc*, `toLocation (fromLocation` *loc*`)` should return `Just` *loc*.

fromLocation :: Location → String
gives back the two-character string version of the specified location; for any location *loc*, `toLocation (fromLocation` *loc*`)` should return `Just` *loc*.

feedback :: [Location] → [Location] → (Int,Int,Int)
takes a target and a guess, respectively, and returns the appropriate feedback, as specified above.

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

nextGuess :: ([Location],GameState) → (Int,Int,Int) → ([Location],GameState)
takes as input a pair of the previous guess and game state, and the feedback to this guess as a triple of the number of correct locations, the number of guesses exactly one square away from a ship, and the number exactly two squares away, and returns a pair of the next guess and new game state.

You must call your source file `Proj2.hs`, and it must have the following module declaration as the first line of code:

```
module Proj2 (Location, toLocation, fromLocation, feedback,
              GameState, initialGuess, nextGuess) where
```

In the interests of simplicity, please put all your code in the single `Proj2.hs` file.

Assessment

Testing your code

The **Compile** button in Grok will not work for this project, because it is intended to be used to compile a full Haskell program with a `main` function. You'll want to test your code either on your own computer using `ghci`, or in Grok by using the **Terminal** button. Normally once you click **Terminal** in Grok, you'll automatically have access to the functions you've defined and be able to test them, however since your code file for this project is `Proj2.hs`, Grok doesn't automatically load the file for you. You can load the file in Grok using the following `ghci` command:

```
:load Proj2.hs
```

(actually, you can abbreviate that to `:l Proj2` and you can get help on the valid `ghci` commands with the `:help` command). This is how you must use `ghci` when it is installed on your own computer, as well.

When you are ready to submit your code through Grok, hit the **Mark** button. This will perform a few tests and show you the feedback. But for this project, it will only run some sanity checks. **Passing these sanity checks does not guarantee that your code will pass all tests.** You must thoroughly test your own code.

The first sanity check will check the correctness of your feedback function. Following checks will check your guessing code (your`initialGuess` and `nextGuess` functions) with decreasing limits on the number of guesses. If your code passes all of these checks (with no failed checks), it is likely your code will achieve a fairly low average number of guesses when the final testing is performed (so it is likely to receive most of the points for average number of guesses). If your code fails a sanity check, you should check the message to determine the cause. It may simply be that your code took too many guesses for a particular test case; that does not mean you code is wrong, just that it may not receive a high mark for the quality of its guessing.

Hints

1. Start by defining your `Location` type. Take care to design an appropriate type. Then write your `toLocation` function and your function to convert a `Location` to a `String`. You'll need to write an `instance` declaration so that `Location` is in the `Show` class (`deriving Show` will not give you a correct `show` function).
   Next write your `feedback` function and test it very carefully. If your `feedback` function is erroneous, correct guessing code can easily go wrong.

   Finally, write your `initialGuess` and `nextGuess` functions. I suggest starting with a simple implementation, and get it working, before trying to reduce the number of guesses. Below are several hints for that.

2. A very simple approach to this program is to simply guess every possible combination of locations until you guess right. There are only 4960 possible targets, so on average it should only take about 2480 guesses, making it perfectly feasible to do in 5 seconds. However, this will give a very poor score for guess quality.

score for guess quality.

3. 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. That way you don't need to remember past guesses or feedback.

4. 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 *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.

of remaining possible targets.

5. 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.
   The choice of a good first guess is quite important, and the best first guess might not be what you'd intuitively expect. It turns out you get more information from feedback like (0,0,0) (which tells you there are no ships within 2 spaces of any of the guessed locations) than from feedback like (1,1,1), which says there are ships near all your guesses, but not where they are.

6. 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. In other words, all the guesses that yield the same feedback will leave you with the same set of remaining possibilities — specifically, the set of guesses that yield that feedback.
   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

1 2 6

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. In general, the formula is:

$$\sum_{f \in F} \frac{\text{count}(f)^2}{T}$$

where *F* is the set of all distinct feedbacks ((3,0,0), (1,0,2), and (2,0,1) in the example above), *count*(*f*) is the number of occurrences of the feedback *f* (1 for (3,0,0), 3 for (1,0,2), and 6 for (2,0,1) in the example), and *T* is the total number of tests (10 in the example).

Once you've computed this for each possible guess, you can just pick one 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.

7. 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.