CSCC24 & CSC324 Winter 2018 – Assignment 3
Due: Sunday, March 24, midnight
This assignment may be done in pairs.
This assignment is worth 10% of the course grade.

Goals

In this assignment, you will practice using custom monads and polymorphism for both implementing the game logic of a simple game (in the real world it could be business logic) and performing testing on such implementations. The "polymorphism" part means that you just need to write the game logic once, but polymorphically; then one instantiation becomes production code, and another becomes the target of black-box mock testing.

As a 3rd-year computer science student and soon-to-be professional software developer, you will also practice switching back-and-forth between two opposite roles, or two sides of a fence:

- Sometimes, you are writing the game logic. You are given one abstract method *gmAction* that you can call to send a message to the "player", and it returns the player's reply. You do not know, and should have no need to care, how this method works; it just does. Another way to put it: You do not know how your game server will be used, e.g., whether *gmAction* uses a terminal to converse with a human player or is actually part of a test bot.

  All you know is that you need to use *gmAction* to send correct hints to the "player" (be it human or bot), and then based on their reply, and according to the rules of the game, determine how the game proceeds, when it ends, and whether the player wins or loses.

- Some other times, you are black-box testing the game logic. You are given an alleged game master (game server) that you need to perform black-box testing on. You do not know, and should have no need to care, how it was coded up. All you know is that you need to verify its hints and give it various player messages to see how it reacts.

  Your leverage, though, is that this time you have total control over how to instantiate the polymorphic game master and what *gmAction* actually does, and you exploit this for your testing purposes.

As usual, you should also aim for reasonably efficient algorithms and reasonably good and simple coding style.

The Guessing Game

The Guessing Game is played as follows. Suppose a secret integer between 1 and 100 is given. The game master knows this secret number, and the player tries to guess it. It goes like this:

1. The game master hints the player of the currently known range (conveyed by two numbers, the lower bound and the upper bound). (Initially the range is 1...100, but this will get updated as follows.) Then the player replies.

2. The player can surrender; then the game ends and the player loses; the game master also reveals the secret number.

   Or the player can make a guess g. Then:

   - If g is out of range, no change to the currently known range. Go back to 1. (Basically the game master tries again.)
   - If g is the secret number, the game ends and the player wins.
   - If g is less than the secret number, the lower bound is updated to $g + 1$, so as to give the player a better hint next round. Go back to 1.
   - If g is greater than the secret number, the upper bound is updated to $g - 1$, so as to give the player a better hint next round. Go back to 1.

The Questions

1. [8 marks] Implement the game master as the function *guessingGame*. Note that it is polymorphic as stated in the goals, so that it can become either production code or testable code by suitable instantiation.

   Suggestion: Write a local function for the game loop, and it takes the lower and upper bounds as parameters. So "go back to 1" is just a recursive call with possibly new parameters for the bounds.

   In GameMasterDef.hs, there is a *goofy* game master that implements a different, silly game. Its purpose is to demonstrate the toolbox you can use.

   You can do human-subject tests of *guessingGame* by loading GameApps.hs and running *guessIO*; you can also do the same to arbitrary game masters (given as a parameter, such as *goofy*) using *playIO*. Both are based on making *IO* an instance of *MonadGameMaster*, in which *gmAction* painstakingly converses with a human (hopefully) via stdio.

2. [8 marks] To facilitate testing (actually as well as all other ways of instantiating *gmAction*), a data type is defined to represent the behaviour of arbitrary programs of type

   $$MonadGameMaster\, m \Rightarrow m\, a$$

   Note that as far as we care, such a program is only going to call *gmAction* a number of times, and then quits and returns a value (of type a). As a first cut, we can represent this by a custom-made singly-linked list type like:

```
-- Attempt #1, not finalized
data L = Done | GMAction Integer Integer L
```

   But it is missing two aspects and needs amendments.

   The easy amendment is that when this "list" ends, we also have to carry a return value:

```
-- Attempt #2, not finalized
data L a = Done a | GMAction Integer Integer (L a)
```

The difficult amendment is that we need to recognize that "the rest of the list" is not fixed, but varies by the player's reply. This can be modelled by a function that takes the player's reply as a parameter to determine the rest of the "list":

```
-- Attempt #3, almost finalized except renaming
data L a = Done a | GMAction Integer Integer (PlayerMsg -> L a)
```

The starter code has the finalized version and naming (*FreeGameMaster*).

Your job in this question is to make *FreeGameMaster* an instance of Functor, Applicative, Monad, and MonadGameMaster.

You can do human-subject tests by loading GameApps.hs and running *guessFreeIO* and *playFreeIO*. These are based on detecting every occurence of *GMAction* and conversing with the human player accordingly. If you do this question right, *guessFreeIO* and *playFreeIO* behave the same as *guessIO* and *playIO*.

3. [8 marks] We can instantiate any polymorphic game master to the *FreeGameMaster* type for mock testing. The idea is that since it is almost data (almost a list) that represents the important events of a game master, you can check those events against what you expect. Examples:

   - If you see a *Pure a*, at this point do you expect to see *Pure*, i.e., do you expect the game master to end now? (If not, the game master is wrong.) If so, is a the correct return value?
   - If you see a *GMAction i j f*, this means the game master is calling *gmAction* now; are you expecting this call now? Are i and j the correct lower and upper bounds at this point?

     Furthermore, you can make up a player's reply and pass it to f to see what the game master does next. Rinse and repeat...

   Your job in this question is to implement a tester as the function *testGame* to test whether a game master is conforming to the rules of the Guessing Game.

   As examples, *testGame guessingGame* should be True (assuming *guessingGame* is correct), and *testGame* $(\lambda\_ \to goofy)$ should be False because *goofy* implements some other game.

End of questions.