CSCC24 Summer 2019 – Assignment 2
Due: Saturday, July 20, 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 and when it ends.

- 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 can instantiate for testing purposes.

As usual, you should also aim for reasonably efficient algorithms and reasonably lucid code.

**The Jug Game**

The Jug Game starts with an array of jugs, each jug with a capacity (stays constant during the game) and an amount of water (changes during the game). Capacities and amounts are in non-negative integers of litres. The goal is of the form "jug $\#i$ has exactly $x$ litres of water" (no constraint on the other jugs). At each turn, the player requests to "pour from jug $\#i$ to $\#j$"; this means transferring water from jug $\#i$ to $\#j$ until jug $\#i$ is emptied or jug $\#j$ is fully filled, whichever happens earlier. Important: The player only gets to specify the source jug and the target jug, not how much water to transfer; how much water to transfer is always governed by the given rule.

Example: A jug game starts with:

| jug # | capacity | initial amount |
|---|---|---|
| 0 | 2 | 0 |
| 1 | 3 | 0 |
| 2 | 4 | 4 |

and the goal is to have 1 litre in jug #0. The player needs to request:

1. pour from #2 to #1
2. pour from #2 to #0

**The Questions**

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

   The game master is done when the goal is reached.

   If the player's transfer request makes no sense (e.g., jug indexes out of range), no change to the amounts in the jugs, just loop back for another turn.

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

2. [4 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 (Important: not just correct game masters) of type

$$MonadGame\, m \Rightarrow m\, a$$

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 | Step (Array Int Jug) Goal 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 | Step (Array Int Jug) Goal (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 | Step (Array Int Jug) Goal (PlayerMsg -> L a)
```

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

Your job in this question is to make *GameTrace* an instance of Functor, Applicative, Monad, and MonadGame. Important: Again, this is for arbitrary programs of type $MonadGame\, m \Rightarrow m\, a$, not just correct game masters; the rules of the Jug Game do not belong here.

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

3. [4 marks] We can instantiate any polymorphic game master to the *GameTrace* 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 you see a $Step\, jugs\, goal\, f$, this means the game master is calling *gmAction* now; are you expecting this call now? Does *jugs* have the right numbers 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 two testers:

   - *testOneStep* tests whether the game master works correctly for one turn. Start with these jugs:

| jug # | capacity | initial amount |
|---|---|---|
| 0 | 7 | 5 |
| 1 | 4 | 2 |
| 2 | 5 | 2 |

And use the goal "jug #2 has 1 litre".

There are 3 jugs, so there are 6 choices for the 1st move. Check the outcomes of all of them. Check everything: The jug arrays and the goals carried by the *Step*s you receive must all be checked.

- *testUntilDone* tests whether the game master finishes after you have made the right moves. Start with the 3 jugs and the goal in the example in the game description; make the two moves listed in that example; lastly check that the game master finishes, i.e., expect *Pure*(). It is OK to omit checking the jug arrays and the goals carried by *Step* here.

These testers have *Maybe*() as the codomain. Use *Just*() to indicate passing your test; use *Nothing* to indicate failing your test. Recall that *Maybe* is a Monad instance—this helps writing fairly clean code for a sequence of checks that automatically quits at the first issue!

End of questions.