CSCC24 2021 Summer – Assignment 3

Due: Monday, July 26, midnight

This assignment is worth 10% of the course grade.

In this assignment, a domain-specific monadic type class is given, i.e., there are domain-specific methods, along with the usual Monad, Applicative, and Functor methods as connectives. You will work on both sides of the fence:

- You will implement a program in the domain. It uses only the methods, and so it is polymorphic in the type class. It can be instantiated one way for an application, and another way for testing.
- You will implement an instance of this class. (This means a model or representation.)

In question 1, code quality is worth 10% of the marks.

Solitaire Knight

The Solitaire Knight game in this assignment is played on an $n \times n$ chess board, $n \geq 1$. A knight is at a certain initial position on the board, and it is the only game piece. The legal moves are exactly the legal knight moves. The goal is to get the knight to a certain destination (known to player). We use integers in $1, \ldots, n$ for x- and y-coordinates.

Question 1: Game Server (10 marks)

The job of a game server is to converse with a user interface (which, normally, converses with a player), perform bookkeeping, and enforce the game rules.

The following type class is defined for the methods that the game server uses to converse with a user interface:

```
class Monad m => MonadKnight m where
    tellPosAskNext :: Pos -> m (Int,Int)
    buzz :: m ()

data Pos = MkPos Int (Int,Int) (Int,Int) [(Int,Int)]
    deriving (Eq, Show)
-- field order: board size "n", destination, current knight position,
--              legal next positions
```

`tellPosAskNext` is for asking the user interface for where the player wants to move the knight to. The `Pos` parameter is to remind the player of the current game information: board size (n), destination, current knight position, and the list of legal next positions (you may use the provided `knightNext` to compute this). When this method returns, the answer is the player's choice.

The `buzz` method is for notifying the user interface that the player's choice is illegal.

Implement the game server

```
knightServer :: MonadKnight m => Int -> (Int,Int) -> (Int,Int) -> m Integer
```

The parameters mean: board size (n), initial knight position, destination. The `Integer` answer should be the number of legal moves the player takes to reach the destination.

The workflow of the game server should be:

1. If the current knight position is already at the destination, give the move count as the answer. (Base case: If the initial position is already at the destination, the move count is 0, and there is nothing else to do.)
2. Call `tellPosAskNext` to give the current game information and receive the player's choice for the next knight position.
3. Check whether the player's choice is legal.
4. If yes, repeat, but note that you have a new position and the move count goes up.
5. If no, call `buzz` and repeat, but note that the position and the move count stay the same.

You can easily keep track of the current knight position and the move count by making them parameters of your recursive helper.

You can test your game server by `playIO` from KnightApp.hs, e.g.,

```
playIO (knightServer 8 (1,1) (1,3))
```

This instantiates the game server to actually converse with you via stdio. Here is a sample playlog:

```
> playIO (knightServer 8 (1,1) (1,3))
8x8 board, destination=(1,3), current=(1,1)
your next position can be one of: [(3,2),(2,3)]
Please enter the next position: (4,4)
Sorry, that was an illegal move.
8x8 board, destination=(1,3), current=(1,1)
your next position can be one of: [(3,2),(2,3)]
Please enter the next position: (3,2)
8x8 board, destination=(1,3), current=(3,2)
your next position can be one of: [(5,3),(4,4),(2,4),(1,3),(1,1),(5,1)]
Please enter the next position: (1,3)
You have used 2 moves.
```

Question 2: Game Trace (6 marks)

The behaviour of a game server, or part of it, correct or incorrect, or even an arbitrary program that happens to use `MonadKnight` methods, can be represented in a data form:

```
data KnightTrace a
  = Answer a
  | Step Pos ((Int,Int) -> KnightTrace a)
  | Buzz (KnightTrace a)
```

- The `Answer` case is when the program gives an answer.

- The `Buzz` case is when the program calls `buzz`. The field is what the program does next.
- The `Step` case is when the program calls `tellPosAskNext`. The `Pos` field records what `Pos` parameter the program passes. The 2nd field is what the program does next; it is a function because what happens next depends on what `(Int,Int)` value the program receives.

  So for example if you call that function with `(2,3)`, it means you want to simulate that `tellPosAskNext` returns `(2,3)`; then you can find out how the program reacts to that.

`KnightTrace` can be made an instance of both `Monad` and `MonadKnight`. Implement them. Some hints:

- `return` means "here is the answer" so it is a pretty easy one.
- `foo >>= cb` is sequential composition plus passing `foo`'s answer to `cb`.
- `Answer a >>= cb` can only be one thing, dictated by a monad law.
- `Buzz k >>= cb` can only be a `Buzz` again. How do you combine `k` and `cb` so that they happen in that order?.
- `Step pos f >>= cb` is similar, just harder, with the extra `(Int,Int)` parameter.
- For the `buzz` method, the tough question is what should you put in the field for "what happens next"? The answer is anti-climatic after you consider: What if there is a program that calls `buzz` and that's it? What is its "next thing to do"?
- `tellPosAskNext` is similar.

You can test those by your game server and `playTrace` from KnightApp.hs, e.g.,

```
playTrace (knightServer 8 (1,1) (1,3))
```

It instantiates your game server to the `KnightTrace` type, then acts as an interpreter.

Question 3: Checker (4 marks)

A benefit of `KnightTrace` in data form is that it is checkable. In this question, you will implement a simple checker:

```
simpleCheck :: KnightTrace Integer -> Bool
```

It is meant for `knightServer 8 (1,1) (3,2)`, and the things to check are:

1. We expect the `Step` case right away, so if you see the other two cases, return False.
2. We should then check the `Pos` field. For simplicity, let's just check that the current knight position is `(1,1)`.
3. If we now move to `(3,2)`, the game ends and we have made only 1 move. This means if we call the function field with `(3,2)`, we expect `Answer 1`; any other case or value is wrong.
4. If all of the above conditions are met, return True.

End of questions.