NATIONAL UNIVERSITY OF SINGAPORE

Department of Computer Science, School of Computing

IT5100A—Industry Readiness: Typed Functional Programming

Academic Year 2024/2025, Semester 1

ASSIGNMENT 3

MONADS

Due: 10 Nov 2024 **Score**: [20 marks] (20%)

PREAMBLE

In this assignment, we will be developing a game of *monadic* tic-tac-toe. The goal of this assignment is to get you used to writing monadic programs and to show that monads provide great abstractions for users. The rules of tic-tac-toe can be found here: `https://en.wikipedia.org/wiki/Tic-tac-toe`. Our game has several features:

- Users play crosses, and the bot plays circles
- Users always start first
- Program allows users to enter their desired board position from the console.
- Bot plays random positions on the board

An example execution of the program is shown in Figure 1.

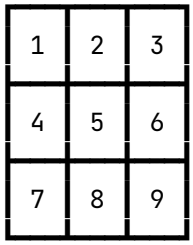
This assignment is worth **20 marks** which constitutes **20%** of your overall grade. It is due on **10 Nov 2024**. **To allow us to release the assignment solutions before the practical exam, late submissions will not be graded**.

Ensure that your submission is compilable; uncompilable code will be penalized heavily. You are not allowed to use any external dependencies other than `transformers` and `mtl`. Other than these libraries, you can only use modules from Haskell's `base` library (which includes `Prelude`). Style will also be a factor in grading; since we are exclusively working in a monad, use of `do` notation is strongly encouraged.

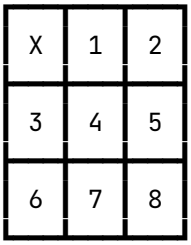
You are given a `cabal` project with all of the required files. To begin the assignment, run `cabal install` to install dependencies and `cabal haddock --haddock-all` to generate the module documentation. The assignment makes extensive use of `transformers` and `mtl`, and you can view documentation for those on Hackage. There are also some testing functions in `Main.hs` which you may use to test your code. However, you should test your code manually via the command line. To run your code, enter `cabal run` in the console. For quick testing with GHCI, you can run `cabal repl` which loads your entire program.

You are to submit *only* `app/Game.hs` (not `app/Data/Game.hs`) and `app/Game/Monad.hs` to Canvas. We will use a separate private test suite to test your code.

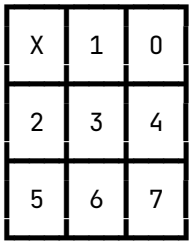
© 2024 National University of Singapore. All rights reserved.



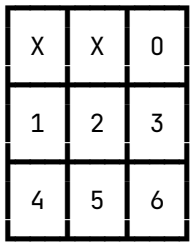

```
Enter a board position.
1
```



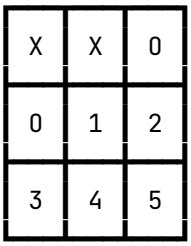

```
Bot plays position 2
```



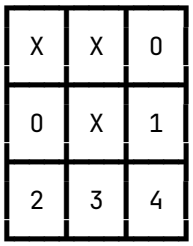

```
Enter a board position.
1
```



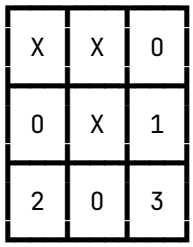

```
Bot plays position 1
```



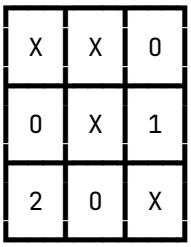

```
Enter a board position.
1
```




```
Bot plays position 3
```




```
Enter a board position.
3
```




```
Human wins!
```

Figure 1: Example gameplay (to save space, we show each step of the game from left to right and top to bottom).

TOUR OF THE PROJECT

THE `Data.Game` MODULE

The data structures and their supporting functions are all defined for you in `app/Data/Game.hs`. This file should not be modified. You can view the documentation in your browser after having run `cabal haddock --haddock-all`.

The **`Data.Game`** module consists of:

- **`Data.Game.GameBoard`** is a data structure representing a game board. The constructor is private to prevent users from creating invalid boards. Some boards have already been defined for you, such as **`Data.Game.`**`blankBoard` and **`Data.Game.`**`testBoard`, although you will not be needing those. The **`Data.Game.`**`countBlanks` and **`Data.Game.`**`setBlank` functions count the number of blanks in a board and sets the n^{th} blank position of the board respectively. Note that **`Data.Game.`**`setBlank` automatically sets the piece of the current player (X for users, O for bots).

© 2024 National University of Singapore. All rights reserved.

- `Data.Game.Player` is a data structure describing a player.
- `Data.Game.GameStatus` describes the status of the current game. A game can either have a winner, conclude with a draw, or be still ongoing with the current player's turn. The status of the game is uniquely determined by the board (see `Data.Game.gameStatus`).

You may read the source code for more information.

THE `Game.Monad.GameM` MONAD

All of our code will be written against the `Game.Monad.GameM` monad defined in (see `app/Game/Monad.hs`), which is essentially the `IO` monad enriched with the `Control.Monad.State.StateT` monad transformer.

The state which `Game.Monad.GameM` uses consists of a `Data.Game.GameBoard` and a `Game.Monad.RandomSeed` which is just a plain `Integer` (with arbitrary precision). We will be defining many helper monad operations in this file. The type declarations for `RandomSeed`, `GameState` and `GameM` should **not** be modified.

THE `Game` MODULE

We will be defining the main logic of the game (with helper functions) in the `Game` module, found in `app/Game.hs`.

THE `Main` MODULE

The `Main` module is always the entry point of the program. In particular, the `main` function is what is being executed by the Haskell runtime. You should see that the `main` function is defined as `playRandomGame` which plays tic-tac-toe with a random seed. There are also other functions you can use to test your code as you are writing the solutions for the assignment. To run those tests, just replace `playRandomGame` with that testing function in `main`. For example, if you want to `testGetsPuts`, then define `main` like so:

```
-- ...
main :: IO ()
main = testGetsPuts
-- ...
```

and run the program.

Now you should have enough information to begin working on the assignment.

© 2024 National University of Singapore. All rights reserved.

BUILDING THE MONAD [10 marks]

Monads are great because they allow us to perform composition in context. However, they are also great for us to abstract implementation details away from the user! In this section, we shall enhance the definition of `GameM` so that users of the monad can work with it seamlessly. We will work exclusively in `app/Game/Monad.hs` in this section.

Question 1 (Getting and Updating State) [2 marks]. The state used by `GameM` is `(GameBoard, RandomSeed)`. Therefore, using `get` and `put` (from `Control.Monad.State`) can be tedious, having to destructure the state and potentially only update parts of it. As such, we shall write some simple getter and setter functions to receive or update a part of the game state.

Write four functions:

1. `getBoard` retrieves the `GameBoard` from the `GameState`
2. `getSeed` retrieves the `RandomSeed` from the `GameState`
3. `putBoard` updates the `GameState` with a new `GameBoard`
4. `putSeed` updates the `GameState` with a new `RandomSeed`

The `get` and `put` from `Control.Monad.State` (see `transformers` and `mtl` documentation) should be of great use to you.

Question 2 (I/O) [2 marks]. The game involves I/O, thus we should write some helper functions for printing output to the console and reading user input.

Write the following functions:

1. `putMsg` prints a string to the console, just like `putStrLn`, but within the `GameM` monad
2. `readLine` gets a line of user input, just like `getLine`, but within the `GameM` monad

> **Tip**: The `liftIO` function in `mtl` (`Control.Monad.IO`) will be of great use to you.

Once you have completed all your `get`, `put` functions and the `putMsg` and `readLine` functions, you may run some tests by amending the definition of `main` in `app/Main.hs` to:

```
main :: IO ()
main = testGetsPuts
```

Run the program with `cabal run` and ensure that the outputs are sensible to you.

© 2024 National University of Singapore. All rights reserved.

Question 3 (Getting the status of the game) [2 marks]. Although we have the ability to get the current `GameBoard`, we also want to get the current status of the game. Write a function `getGameStatus` which obtains the current status of the game.

> **Tip**: There is no need to use `do` notation here. Just use `getBoard` which you've defined earlier, `getGameStatus` from `Data.Game` and `fmap`.

Once you have completed this question, you may run some tests by amending the definition of `main` in `app/Main.hs` to:

```
main :: IO ()
main = testGameStatus
```

Run the program with `cabal run` and ensure that the outputs are sensible to you.

Question 4 (Random Number Generation) [2 marks]. We would like to play tic-tac-toe against a bot that makes random moves, thus, we need a random number generator. Write a function `randomInt` that computes a new pseudorandom number. Given a current seed s, the new seed will be $5 \times s$ modulo 12356713, and the random number will be s modulo 133711.

> **Tip**: Use `getSeed` and `putSeed` which you've defined earlier to get and update the random seed, and use `fromIntegral` to convert an `Integer` value to an `Int`.

Once you have completed this question, you may run some tests by amending the definition of `main` in `app/Main.hs` to:

```
main :: IO ()
main = testRandomInt
```

Run the program with `cabal run`. You should see the numbers `64875`, `54169` and `104942` in the console.

Question 5 (Setting a Blank Position in the Board) [2 marks]. Every time a player makes a move, they are setting a blank position on the board with their piece. Write a function `setAt n` which sets the n^{th} position `GameBoard`. `n` can be any integer value.

> **Tip**: Use `getBoard` and `putBoard` which you've defined earlier to get and update the game board, and use `setBlank` from `Data.Game` to set the n^{th} board position with the current player's piece.

Once you have completed this question, you may run some tests by amending the definition of `main` in `app/Main.hs` to:

```
main :: IO ()
main = testSetAt
```

Run the program with `cabal run` and ensure that the outputs are sensible to you.

© 2024 National University of Singapore. All rights reserved.

CREATING THE GAME [10 marks]

Now that the `GameM` monad has set us up for success, it is time to define the main game logic! In this section, we shall work exclusively with the `app/Game.hs` file.

Question 6 (Bot's Random Play) [2 marks]. Write a function `randomPosition` to generate a random blank position on the board as an `Int`. The blank board positions are numbered 1 to n where n is the number of blank positions on the board, therefore, your random position should also be within these bounds.

> **Tip**: If you want a random integer between (and including) 1 and n, generate a random integer x, and compute (x modulo n)+1. x should come from `randomInt` you wrote earlier, and n should be the number of blank positions on the game board (use `getBoard` from earlier, and `countBlanks` from `Data.Game`).

Question 7 (User's Play) [3 marks]. Write a function `readPosition` that repeatedly reads input from the user until the user enters a valid blank board position. Remember that valid blank board positions are numbered from 1 onwards.

> **Tip**: The `putMsg` and `readLine` functions defined earlier should be of great use to you. In addition, recall that user input comes in the form of a `String`. Thus, to convert a `String` to an `Int`, use the `readMaybe` function from `Text.Read`. Importantly, *you must specify the type you want to convert the string to.* For example, you should write:
>
> ```
> do -- ...
>    user_input <- readLine
>    let mn :: Maybe Int = readMaybe user_input
>    -- ...
> ```

Question 8 (The Game) [5 marks]. Now we can finally write the game logic! Write a function `game` that plays a game of tic-tac-toe. Use all the functions defined throughout this assignment to help you. In particular, the `getGameStatus` should guide the function on what to do next—whether to terminate the game with a winner or a draw, or to let the user or bot make a move and continue with the game.

Once you have completed this assignment, test your game by modifying `main` in `app/Main.hs` to:

```
main :: IO ()
main = playRandomGame
```

Run the program with `cabal run` and ensure that the game operates correctly. In addition, the user should be able to input anything they want to the console, and the program should not crash with an exception or behave unexpectedly.

– End of Assignment 3 –

© 2024 National University of Singapore. All rights reserved.