Functional Graphics with Gloss

Functional Programming with Haskell: Homework 2

Exercises

Your main goal this week is to complete your game of Pong! However, if you would like to get some extra practice, I've included a few functions to implement under the "Extra Practice" section.

Important

Make sure your implementation of Pong has the following aspects:

- You have a data structure describing your game of Pong. This data structure should include at least the following:
  - The two players, each as a value of some `Player` data type. Each player should have at least their current position vertically, though you may include other aspects (player speed, acceleration, `x`-position, etc.).
  - The ball, along with its `x` and `y` positions and velocities.
- A function that converts a game of Pong (in your data structure) to a `Picture` using the functions provided in `Graphics.Gloss` and `Graphics.Gloss.Picture`.
- An example game initialized with sensible parameters for all the components.
- A `main` function which calls `display` to show your example game.

Highly Recommended

Make sure you have all the components of your Pong game outlined in the "Important" section. Once you have that, add time-dependence and animation using the `simulate` function from `Graphics.Gloss`. This simulate function requires you to be able to step the state of your game by creating a new game state which represents the new game. This should, at the very least, advance the position of the ball using the current velocity of the ball. Once this is done, you should have the following:

- A `main` function that uses `simulate` to show a non-interactive simulation of Pong.
- A ball which starts somewhere in the middle of your window and then flies off one of the edges.

Once you have that, implement collision detection with the paddles. Then, position the ball and set its velocity so that it bounces when it hits a paddle. It should also bounce when it hits one of the edges of the screen that doesn't have a paddle. At this point, you should have the following:

- A `main` function that uses `simulate` to show a non-interactive simulation of Pong.

- A ball which starts somewhere in the middle of your window, flies towards one of the paddles, bounces off the paddle, bounces off a wall, and then flies off the screen on the other side.

Finally, add interactivity to let the players control the ball. In order to do this, you will want to use the `play` function from `Graphics.Gloss` in your `main` function. This `play` function requires an updater function that updates the game state given an event that happened, such as the user pressing a button. Implement this function in such a way that the users can control the paddles. At this point, you should have the following:

- A `main` function that uses `play` to run a game of Pong.
- A fully functional (no pun intended) game of two-player Pong! One user should be able to use the arrow keys to control their paddle, and the other should be able to use the WASD keys to control theirs.

Good Practice

If you are feeling particularly ambitious, write a small AI for your game of pong. The computer should move their paddle in the direction that the ball is currently going, to try to track it.

If you're feeling up for some more conventional exercises, here are a few functions that are not related to the lectures, but will give you a lot of practice with Haskell. These are just a few functions that are used very commonly throughout programming in Haskell, so you should know that they exist and how to implement them.

Thus, implement the following functions:

```haskell
-- Dealing with potentially empty values in elegant ways.
-- Recall that the Maybe data type is defined in Haskell as follows:
--     data Maybe a = Nothing | Just a
-- You don't need to code this yourself - it's already implemented.

-- Get all the Just values out of a list of Maybes.
-- Example: catMaybes [Just 3, Nothing, Just 5] == [3, 5]
catMaybes :: [Maybe a] -> [a]

-- Collect results from mapping a function that might return Nothing over a list.
-- Don't include any result that yields nothing!
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```