Coursework Implementation Advice

I found people are struggling with the coursework. I am going to give advice on the points I found people were struggling most with which are:

1. Moving (swapping) unhappy agents to random empty cells.
2. Overall structure of the data-flow.
3. Getting all neighbours.
4. Generating Random Numbers.
5. Testing the code.

Moving (swapping) unhappy agents to random empty cells

The tricky bit of randomly swapping unhappy agents with empty cells is to update the empty cells (and the random number generator) along the way. I think it is clear that in this step every unhappy agent needs to be moved, thus we recur/iterate over a list of unhappy agents. The question is then how to deal with empty cells because they need to be updated during swapping. The whole process can be implemented using the foldr function using the following approach:

```
(unhappyAgents', emptyCells', g') = foldr moveCell ([], emptyCells, g) unhappyAgents

moveCell :: Cell  -- ^ the unhappy agent we want to move
         -> ([Cell], [Cell], StdGen)  -- ^ all unhappy agents which we have already moved, the currently empty cells, the rando
         -> ([Cell], [Cell], StdGen)  -- ^ same as above plus the unhappy agent we just moved, the same as above minus the old
moveCell ...
```

Overall structure of the data-flow

It is a bit confusing how to design data-flow in a programming language which has immutable data. There are obviously different ways but the one I use is basically as follows:

```
step cs g ratio = (cs', g')
  where
    emptyCells    = filter isEmpty cs
    agents        = filter (not . isEmpty) cs
    happyAgents   = filter (isHappy ratio cs) agents
    unhappyAgents = filter (not . isHappy ratio cs) agents

    (unhappyAgents', emptyCells', g') = foldr swapRand ([], emptyCells, g) unhappyAgents

    cs' = happyAgents ++ unhappyAgents' ++ emptyCells'
```

Getting all neighbours

Use relative coordinates to define your neighbourhood which you use to compute absolute coordinates which in turn you use to filter out the neighbours from all cells.

1. When you have an (unhappy) agent at position (x,y) the 8 potential neighbours are at [(x-1,y-1), (x, y-1), (x+1, y-),...].
2. Define a constant list 'moore' which has the 8 relative offsets.
3. Define a function genNeighbours :: [Coord] -> Coord -> [Coord], which takes 'moore' as its first argument and an absolute coordinate of an (unhappy) agent as second argument and returns the list of coordinates of all potential neighbours.
4. Filter all cells which have the coordinates generated by genNeighbours - these are the neighbours of the (unhappy) agent. Note that this works independent whether the agent is at the border or not! In case the agent is at the border, the potential coordinates include e.g. (-1,-1) but using filter, no cell will be found for it.

Generating Random Numbers

When generating random numbers with `randomR` you need to make sure to use the new random-number generator returned from `randomR` in the next call to `randomR`! Consider the following code example:

```
let n = length xs - 1
let (x, g0) = randomR (0, n) g
let (y, g1) = randomR (0, n) g
let (z, g2) = randomR (0, n) g
...
```

The values of x, y and z will ALWAYS be the same because always the same random-number generator g is used. A correct approach would be:

```
let n = length xs - 1
let (x, g0) = randomR (0, n) g
let (y, g1) = randomR (0, n) g0
let (z, g2) = randomR (0, n) g1
...
```

Now values of x, y and z are guaranteed to be independent from each other because always the new random-number generator is used! (Note that they could be the same because they are randomly drawn but its very unlikely).

Testing the code

Run the code you have written! Don't code the whole coursework without ever running the code you have just written until the very end: make changes and check their impact. To test your functions you can use stack OR I rather suggest to use Debug.Trace. To use it import Debug.Trace in your Schelling.hs and use the function trace :: String -> a -> a. This function prints the String from the first argument to the console and takes another argument a and returns it after the String was printed. Note that this allows interactive programming (IO) which hides the interactive part (IO) but this should be only used for debugging. Here is a code example how to use it:

```
-- need to import Debug.Trace to use trace
import Debug.Trace

-- when returning result from the step function use trace to print some debug output to the console
step cs g _ratio = trace ("isHappy returns: " ++ show b) (cs, g)
  where
    -- testing the isHappy function by trying it with the first cell in the list
    b = isHappy (head cs) ratio

    -- Note that you can also use the trace function to print information within isHappy !
    isHappy :: Double -> [Cell] -> Cell -> Bool
    isHappy ...
```