Functional Programming with Haskell — Summer 2022
Topic 2. Homework

Nikolai Kudasov

Contents

About homework

You are encouraged to use homework as a playground: don't just make some code that solves the problem, but instead try to produce a nice, concise solution. Try to divide a problem until you arrive at simple tasks. Create small functions for every small logical task and combine those functions to build a complex solution. Try not to repeat yourself: for every logical task try to have just one small function and reuse it in multiple places.

Homework assignments provide clear instructions and may provide some examples. However, you are welcome to make the result extra pretty or add more functionality as long as it *does not simplify* the task and does not make the code *too complex*.

Elegant code and prettified solutions add up to your Participation score. Refer to the Code style below.

Submissions

Submit a homework with all solutions as a single file. Every assignment should be implemented as functions named `solution1`, `solution2`, etc. of type `IO ()`. Assignments could then be run by substituting them as body of `main`:

```
main :: IO ()
main = solution1
```

Before submitting, make sure your code compiles. If it does not — ask for help.

On CodeWorld you can use the Share button to get a link to your code (after you successfully Run the program). You can submit that. Otherwise submit the Haskell file.

Q&A and discussions

If you have any questions regarding homework or the course, feel free to ask:

- in the course Telegram group chat,
- directly via Telegram: @fizruk31337,
- directly via e-mail: n.kudasov@innopolis.ru,
- before, during and after classes.

Code style

Write your programs creatively using these constraints as a guide to the beautiful:

- Use `camelCase` to name functions, variables and types.
- Use descriptive names. Let them be as long as needed, but no longer that that. Good: `renderRemaining`. Bad: `rndr`. Ugly: `renderObjectsWeHaven'tRenderedYet`.
- Do not use tabs: Haskell is sensitive to indentation, so tabs can ruin your experience. Configure your editors so that they convert TABs into spaces automatically.
- Try to keep lines at 80 symbols or less. It is normally easier to read code if you don't have to scroll horizontally.
- Specify a type for every top-level function. This improves documentation, helps your thinking and improves compiler error messages. Usually there is no need to specify types for locally defined functions (e.g. with `let` or `where`).
- Provide a comment for every top-level function, explaining what that function does. Sometimes examples can help a lot with that explanation.
- Use `-Wall` or `{-# OPTIONS_GHC -Wall #-}`. This flag turns on a lot of useful warnings.
- Define all functions total, i.e. defined on all inputs. If a function crashes or loops on some input — it is not total.

1 Escape the room

In this homework you are going to implement a playable escape the room game with one level in a few stages.

1.1 Limited movement

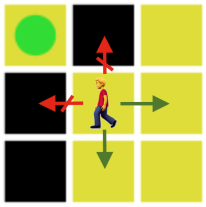

Figure 1: Example of possible character movements.

Make character to only be able to walk over `Floor`, `Button` and `Exit` tiles. Specifically, implement a function

```
-- | Try move character in a given direction.
tryMove :: Dir -> Coords -> Coords
```

Then use it in `handleWorld` so that character would only move where it can.

Hint: a `canMove :: Tile -> Bool` function might be helpful.

Define `solution1` as an implementation of the game with `tryMove`.

1.2 Draw any level map

Now let's make prepare our functions to handle different maps and also make them a bit bigger. Implement a function to draw any given level map of size 21×21:

```
-- | Draw a given level map of size 21x21.
drawLevelMap :: (Coords -> Tile) -> Picture
```

Define a new map and use it for `solution2`.

Try to make the new map interesting! You can add more colors for doors, keep some doors open at the beginning and add more types of tiles.

We will be collecting all maps to make a **multi-level game** later.

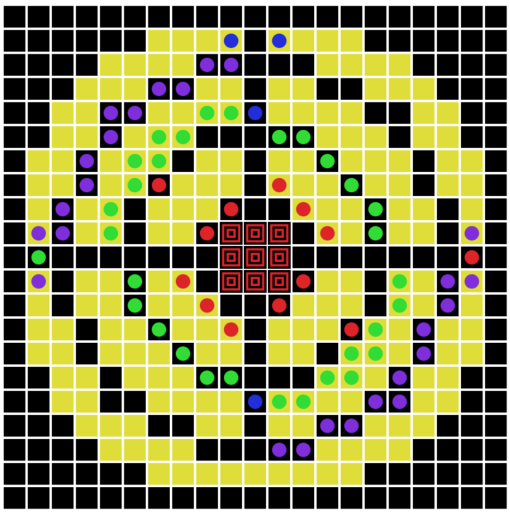

Figure 2: Sample 21×21 map.

1.3 Open doors

Define a function to open doors of given colors:

```
-- | Open some doors.
openDoors
  :: [DoorColor]       -- ^ Doors to open.
  -> (Coords -> Tile) -- ^ Original map.
  -> (Coords -> Tile) -- ^ Map with doors open.
```

Opening a door just replaces it with a `Floor` tile.

Define `solution3` like `solution2` but with all doors open using `openDoors`.

Hint: you might want to define a couple of helpers:

```
eqDoorColor :: DoorColor -> DoorColor -> Bool
oneOf :: DoorColor -> [DoorColor] -> Bool
```

1.4 Press buttons

Now complete the door mechanics by making buttons work! To make it work you will have to keep track of all pressed buttons.

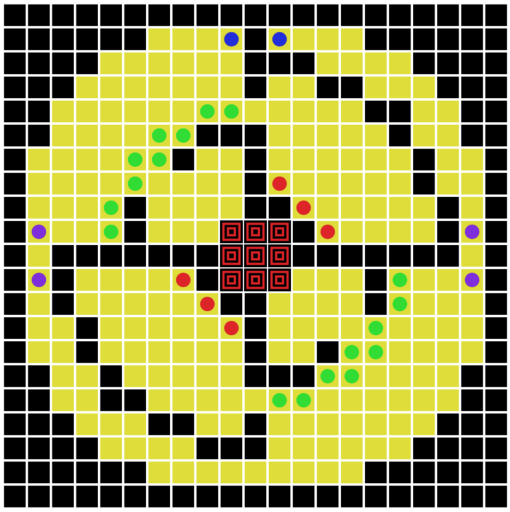

Figure 3: Sample map with opened doors.

Start by thinking of the type of state (`world` in `interactionOf`):

```
-- | State of the game (player coordinates and open doors).
data State = ...
```

Then update all the functions correspondingly!

Define `solution4` as a playable escape the room with complete door mechanics based on your own level. There are several possible mechanics that you may want to implement:

1. Pressing a button opens all doors of the corresponding color;
2. Pressing a button toggles (opens if closed, closes if open) all doors of the corresponding color;
3. Pressing a button happens automatically when character moves onto the button tile;
4. Pressing a button happens manually when player uses special key (e.g. `Enter`);
5. There can be hidden buttons/doors;
6. Button color can be irrelevant for the door it opens;
7. Some buttons can be "single-use" (e.g. you cannot press them again);
8. Something else may happen!