Functional Programming with Haskell
Summer 2022
Topic 1. 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*.

Submissions

Submit solution to this homework as a single file or a permanent link to CodeWorld platform. 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 and you don't know how to fix it — 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 link. 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 Traffic Lights

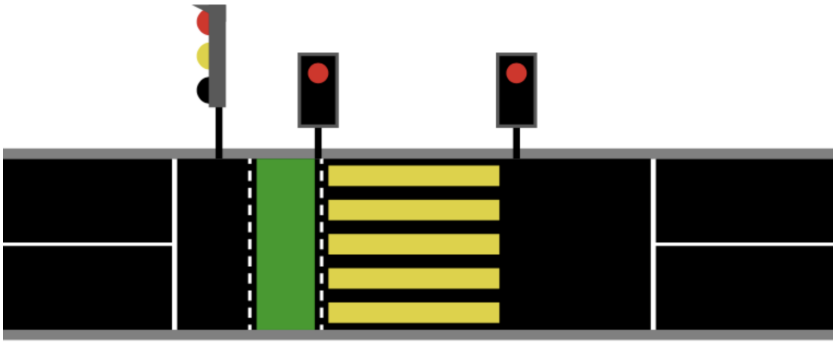

Figure 1: Traffic lights for cars, bicycles, and pedestrians.

In this assignment implement an animation of traffic lights for cars, pedestrians and cyclists.

1.1 Simple traffic system

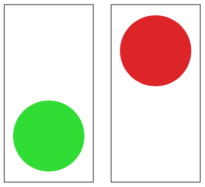

Figure 2: Simple traffic lights system.

In the lecture we came up with this implementation for a simple traffic lights system with two states[1]:

```
{-# LANGUAGE OverloadedStrings #-}
import CodeWorld

-- | Draw a circle for traffic lights given color and Y-offset.
lightCircle :: Color -> Double -> Picture
```

[1] https://code.world/haskell#POklvGdD9Bt1ZxGNVxTU5nA

```haskell
lightCircle c y = translated 0 y (colored c (solidCircle 1))

-- | Green traffic light.
greenLight :: Picture
greenLight = lightCircle green (-1.2)

-- | Red traffic light.
redLight :: Picture
redLight = lightCircle red 1.2

-- | Frame for traffic lights.
frame :: Picture
frame = rectangle 2.5 5

-- | Simple traffic lights with two states.
--
-- * 'True' - green light
-- * 'False' - red light
trafficLights :: Bool -> Picture
trafficLights True  = frame <> greenLight
trafficLights False = frame <> redLight

-- | Traffic lights controller switching lights every 2 seconds.
trafficController :: Double -> Picture
trafficController t
  | floor (t / 2) `mod` 2 == 0 = trafficLights True
  | otherwise                  = trafficLights False

main :: IO ()
main = animationOf trafficController
```

1.2 Complex traffic lights

Based on the code for simple traffic lights system, implement a system with 3 sets of traffic lights operating simultaneously:

- traffic lights for cars with 3 lights and 4 states;
- traffic lights for pedestrians;

- traffic lights for cyclists.

1.2.1 Inactive light circles

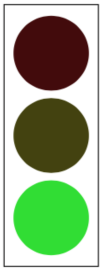

Figure 3: Green light on, others dimmed out.

In the code sample above when a light is inactive it is not drawn. For a more realistic appearance, implement inactive traffic ligths as black, gray or dimmed.

1.2.2 Traffic lights for cars

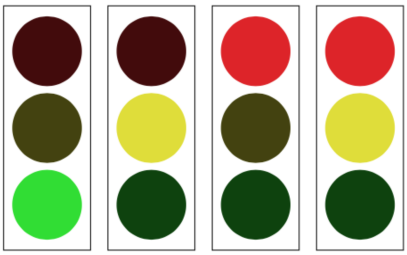

Figure 4: Four states of traffic lights for cars

For cars traffic lights consist of three lights on top of each other:

- red light.
- yellow light;
- green light.

These lights go through a period of four states:

- green light active (go, *3 seconds*);
- yellow light active (complete maneuver, *1 second*).

- red light active (stop, *3 seconds*);
- red and yellow lights active (ready, *1 second*);

1.2.3 Pedestrians and cyclists

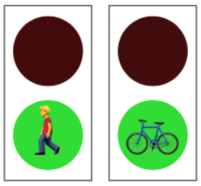

Figure 5: Traffic lights for pedestrians and cyclists.

Traffic lights for pedestrians and cyclists consist of just two lights and have three states:

- green light on (go, *3 seconds*);
- blinking green light (complete crossing, *1 second*);
- red light on (stop, *4 seconds*).

You can add an image of a pedestrian or bicycle using

- a custom function to draw pedestrian and bicycle symbols;
- Unicode symbols (such as `'\x1F6B6'` and `'\x1F6B2'`).

1.2.4 More details

If you have time left, for extra fun, you can add background with the road, the crosswalk, bicycle lane, pavement. You can make some traffic lights turned (i.e. render as seen from the side). You can even add animation for moving cars, bicycles, and pedestrians.

2 Growing tree (+3% extra credit)

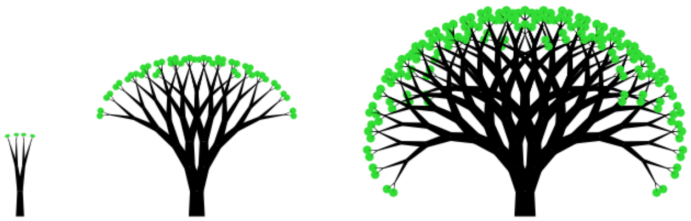

Figure 6: Stages of a growing tree.

In this assignment implement an animation of a growing tree.

2.1 Fractal tree

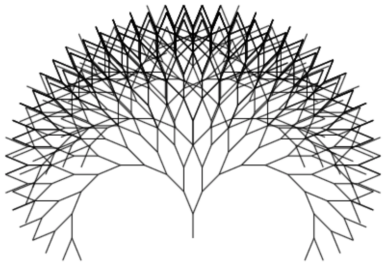

Figure 7: Simple fractal tree.

In the lecture we have come up with this code for a simple fractal tree[2]:

```
{-# LANGUAGE OverloadedStrings #-}
import           CodeWorld

-- | A fractal tree of a given rank.
tree :: Int -> Picture
tree 0 = blank
tree n = segment <>
  translated 0 1 (leftBranch <> rightBranch)
```

[2]https://code.world/haskell#PM695yWbrWCytJU7fuWCd5g

```
  where
    segment = polyline [(0, 0), (0, 1)]

    leftBranch  = rotated (pi/8) (tree (n - 1))
    rightBranch = rotated (-pi/8) (tree (n - 1))

main :: IO ()
main = drawingOf (tree 10)
```

2.2 How a tree grows

2.2.1 Continuous growth

A tree should grow *continuously*, without any sudden discrete jumps in thickness or length. That means there should be many intermediate steps between `tree 8` and `tree 9`. You can achieve this making tree's "age" of type `Double` and adjusting implementation accordingly.

2.2.2 Thick tree

Figure 8: A tree with gradual thickness.

Trees normally get thicker from top to bottom. To achieve this effect each segment should actually become a polygon. Use `solidPolygon` to specify a branch segment that is thick at the bottom and thinner at the top.

Note: adjacent segments should match in thickness where they connect.

Note: final segments have thickness 0 on one end.

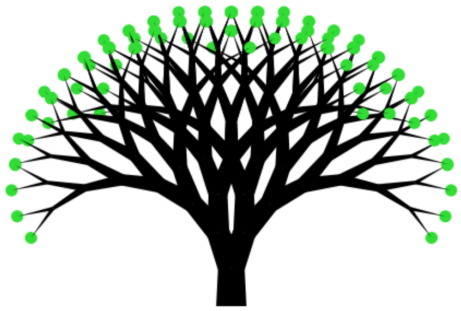

Figure 9: A tree with leaves.

2.2.3 Leaves

Attach leaves to the ends of tree branches. Leaves should grow with the tree (smaller trees have fewer and smaller leaves). You can draw leaves as green circles or anything you want (take inspiration in the nature). You can also add fruit or other decorations.