Rocks

originally Wouter Swierstra, adapted by Andres Löh

Diffculty: easy

For this assignment, you need to use the `wxHaskell` *package. As far as I know, the package is currently not installed on the lab machines, and isn't easy to install there. You can try to install it on your own machine, though.*

In this assignment, we'll complete the 'Haskell Rocks' game, where the player takes control of a spaceship and has to evade several asteroids. A screenshot from the game is shown in Figure 1.

I'm very grateful to Wouter Swierstra, who has designed the exercise on which this assignment is based. Most of the code and a significant part of the explanations are due to him, but all of the potential errors have probably been introduced by me. Also thanks to Daan Leijen, who has used a variant of the game as an example in the article on "wxHaskell: A Portable and Concise GUI Library for Haskell."

Multiple files are distributed. The file `Rocks.hs` contains the main program including all the GUI code. No changes are required in this module. The file `Solution.hs` is where your modifications go. It contains a number of partially implemented functions. A dummy function `tODO` is used to indicate the places where modifications are required.

The file `Solution.hs` can be loaded into `ghci` on its own. It doesn't rely on any other modules or libraries. If you want to play the game, you have to interpret or compile `Rocks.hs`, but this requires a working installation of the wxHaskell library.

Additionally, the distribution contains a couple of images: `rock.ico` contains the image for a single rock, `rocketship.ico` is the spaceship, `explode.ico` is an explosion, and `character.ico` is the portrait that's shown on the statusbar.

Overview

The game is played in a window of fixed height and width. Coordinates in the playing field are represented by the datatype `Point`:

```
data Point = Point PosX PosY
type PosX  = Int
type PosY  = Int
```

Figure 1: A screenshot

Recall the syntax of datatypes: `Point` is a datatype with a single constructor, also called `Point`. The constructor `Point` has two arguments for the x- and the y-coordinate. The coordinates are represented by integers – we define meaningful abbreviations `PosX` and `PosY`.

The origin of the playing field (i.e., `Point 0 0`) is the top-left corner. A larger x-coordinate means further to the right, a larger y-coordinate means further down.

For each object that has a position on the game field (for instance, for the spaceship or the rocks), we store the coordinates of their top-left corner.

An idealized version of the playing field is shown in Figure 2.

In its initial state, the game doesn't do much: you cannot control the spaceship – its position is constant. No rocks are created, so everything is blank. Making the game somewhat fun to play consists of three major tasks: implementing control for the spaceship, implement rock generation and the movement of the rocks, and finally, add the possibility for rocks to collide with the spaceship.

Each of these tasks is now addressed in order, and consists of multiple steps. After completing one task, you should be able to notice the progress when playing the game again. Even though the tasks are designed to be solved in order, you should not spend too much time on a single problem. If you get stuck, try if you can solve other problems first and return to that point later.

Also, don't rely on the GUI for debugging! Rather, devise test cases on your own and run your functions on your test input to see if they have any obvious bugs.

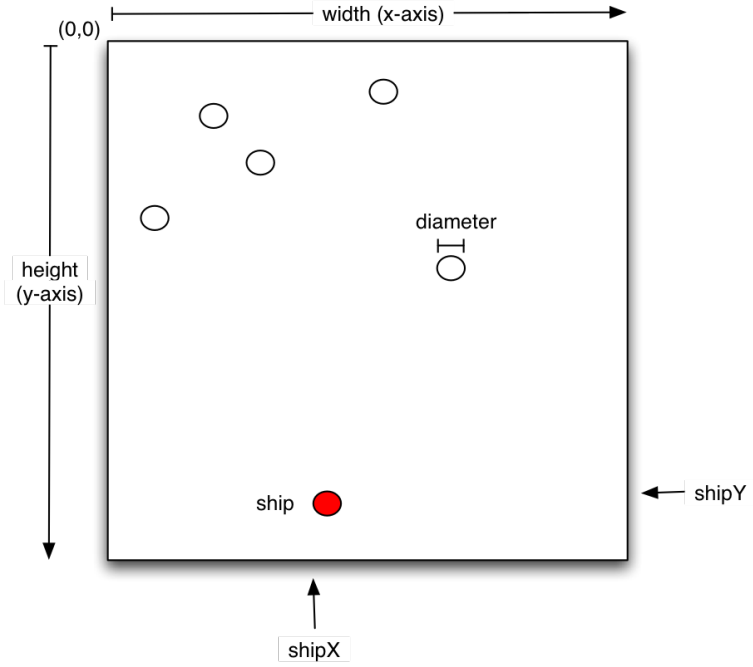

Figure 2: Idealized playing field

Basics

1. Read the skeleton, then load it into `ghci` and try to get familiar with the `Point` type. Try

```
*Main> :t Point
```

to check the type of the constructor. Also check all the predefined constants. Many of these shall be useful in the functions you are about to write.

Spaceship control

2. Implement a pair of functions

```
moveLeft  :: PosX -> PosX
moveRight :: PosX -> PosX
```

Given the x-coordinate of the spaceship, `moveLeft` should compute a new x-coordinate for the spaceship, 5 pixels further to the left. Similarly, `moveRight` should compute the new x-coordinate 5 pixels further to the right.

3 (medium)**.** Modify the functions just written such that the spaceship can never go off screen. Recall that the x-coordinate of the ship is in fact the x-coordinate of the top-left corner of the ship. The width of the ship is given by the constant `iconSize`.

Rock generation and movement

A rock is represented by its *future*: the list of points it will inhabit during its path through the game field. Since the path of rocks cannot be influenced, the complete future can be set into stone once a new rock is created. This motivates the following type abbreviation:

```
type RockFuture = [Point]
```

Here is an example rock that is 'falling' down from the top of the game field at the x-coordinate `10`:

```
exampleRock = [Point 10 0, Point 10 6, Point 10 12, Point 10 18, ...
```

4 (medium)**.** Implement the function

```
track :: PosX -> RockFuture
```

that, given the x-coordinate of the rock, computes the point the rock will cross. The track should only cover the point that are actually on the screen. To solve this problem, you may find it helpful to first write a recursive function

```
multiples :: Int -> [Int]
```

such that `multiples n` is the list of all multiples of `n` (starting at 0) that are less than the height of the playing field. Recall that the height of the playing field is given by the constant `height`. For programming `track`, you can then choose a vertical 'speed' for the rocks, but `6` seems like a good choice.

5. Next, we need to implement the function that adds new asteroids. At any game cycle, a (provided) function produces a new `RockFuture` with a certain probability. The result of this function is of type `Maybe RockFuture`. Recall the type `Maybe`:

```
data Maybe a = Nothing | Just a
```

If the value is `Nothing`, no new rock has been created. If the value is constructed by `Just`, the parameter tells us the future of the new rock. The task is to define a function of type

```
combine :: Maybe RockFuture -> Rocks -> Rocks
```

that adds the potential new rock to the supply of rocks that are currently on screen. The type `Rocks` is just a list of futures:

```
type Rocks = [RockFuture]
```

The combine function should leave the list unchanged if there is no new rock; if there is a new rock, it should be added to the list. Use the standard recipe for defining functions on the `Maybe` type!

6 (difficult)**.** Define the function

```
advanceRocks :: Maybe RockFuture -> Rocks -> Rocks
```

It has the same type as `combine`, but does several things at once and is therefore slightly tricky to define. Don't be afraid to split it up into several smaller functions that you then combine.

Given a potential new rock, and a list of rock futures, the function `advanceRocks` should compute a new list of rocks. This list should be computed in three steps:

- All existing asteroids 'fall' down. For instance,

  ```
  [[Point 1 2,Point 2 3,Point 4 5],
   [Point 6 7,Point 8 9],
   [Point 10 11]]
  ```

  should become

  ```
  [[Point 2 3,Point 4 5],[Point 8 9],[]]
  ```

  Note that the above lists are 'unrealistic' in the sense that they'll not occur in a real game (why?), but still, they serve to show the behaviour: for every list, the first element is removed.

- After all the rocks have been shifted down, any rocks that have disappeared (i.e., have an empty future) should be removed. Our example list then becomes:

  ```
  [[Point 2 3,Point 4 5],[Point 8 9]]
  ```

- Finally, we need to add the new rock to the list, using the `combine` function that we defined above.

While defining the functionality above, try to recall all the higher-order functions on lists we discussed. If you don't succeed using higher-order functions, split the function into several parts and apply the standard recipe for defining functions on lists!

Collision detection

If you have implemented all the tasks above, then the ship moves and asteroids fall down the screen. Unfortunately, nothing happens when they hit your spaceship! In the next few exercises, we're going to change this.

7. To determine the possible collision, we are not so much interested in the *future* of the rocks that are in the game, but in their *current positions*. We therefore introduce the concept of a *snapshot*:

```
type Snapshot = [Point]
```

A snapshot is the list of all the positions that the rocks in the game currently inhabit. Write a function

```
snapshot :: Rocks -> Snapshot
```

that takes a snapshot from the list of all the futures. A value of type `Rocks` is a list of futures. Each future is in turn a list of points. Thus, `Rocks` expands to `[[Point]]`. In every future, the first element indicates the current position. So you will have to extract the first elements out of a list of lists.

After you have done that, ask yourself what happens if one of those futures is empty? Add a comment explaining how you deal with this situation or why you think that you do not have to deal with it. Hint: the only functions 'changing' the state of the current rocks in the game that is called from the main program is `advanceRocks`.

8 (difficult)**.** Implement the function

```
isCollision :: Point -> Point -> Bool
```

Given two points `p1` and `p2`, it should determine whether or not the rock at point `p1` has hit the spaceship at `p2`. To write this function, you may find it helpful to first define a function

```
distance :: Point -> Point -> Double
```

Recall the definition of the `Point` type, and apply the standard recipe for defining a function on `Point` (yes, even if a type has only one constructor). The spaceship will collide with a rock *if the distance from the top-left corner of the spaceship to the top-left corner of the rock is less than the diameter of the rock*. The diameter of a rock is given as the integer constant `diameter`. Use Pythagoras's theorem to determine the distance between the spaceship and the rock. You might find the following functions helpful:

```
fromIntegral :: (Integral a, Num b) => a -> b
```

converts an integral type (such as `Int`) into any numeric type (for instance a floating-point type such as `Double`);

```
sqrt :: (Floating a) => a -> a
```

computes the square root of a (non-negative) floating-point number.

9 (medium)**.** Now we need to put the `isCollision` function to use. Implement the following two functions:

```
collidedRocks   :: Point -> Snapshot -> Snapshot
detectCollision :: Point -> Snapshot -> Bool
```

Both functions take the position of the spaceship and a snapshot as arguments. The function `collidedRocks` now eliminates all the non-colliding rocks, leaving the positions only of those that collide. The function `detectCollision` checks if there is a collision at all.

10 (medium)**.** Finally, define the following function:

```
removeCollisions :: Point -> Rocks -> Rocks
```

Given the current position of the spaceship, and the list of all the rock futures, remove the rocks that have collided with the spaceship. Hint: you might find it helpful to use `snapshot` and `isCollision`. Once more, try to make use of higher-order functions, or use the standard recipe for functions on lists (on the list that is the second argument) as a fallback.

Bonus exercises

These exercises do not influence your grade, but may help you getting more fluent in Haskell and thereby indirectly improve your future grades `;-)`

11 (bonus, difficult)**.** If you've completed the first three exercises, you should have a basic, working Rocks game. Now let your imagination go wild! Add all the features of a real game. Here are a few ideas:

- Allow the ship to move vertically.
- Add bullets that you can fire at rocks to defend yourself.
- Add score that's assigned per second survived and per asteroid destroyed.
- Add a number of lives the player has that is reduced by one everytime the player is hit by a rock. Alternatively, implement regenerating shields that are reduced by a certain amount whenever the ship is hit, and regenerate slowly over time.
- Let rocks no longer always 'fall' down. Instead, give them a direction which has some downwards component.
- Let rocks no longer all have the same speed. Some could move faster, some slower.
- Add larger rocks which split into two (or more) smaller rocks when hit by a bullet, with different direction.
- ...