Homework assignment #4

Grading will be partially automated. Please ensure that you submit exactly one file named *homework2.zip*, containing a single file named *homework2.hs*. Do not alter the function names provided, and strictly adhere to the specified type signatures. Any deviations will result in automated testing failures and impact your grade.

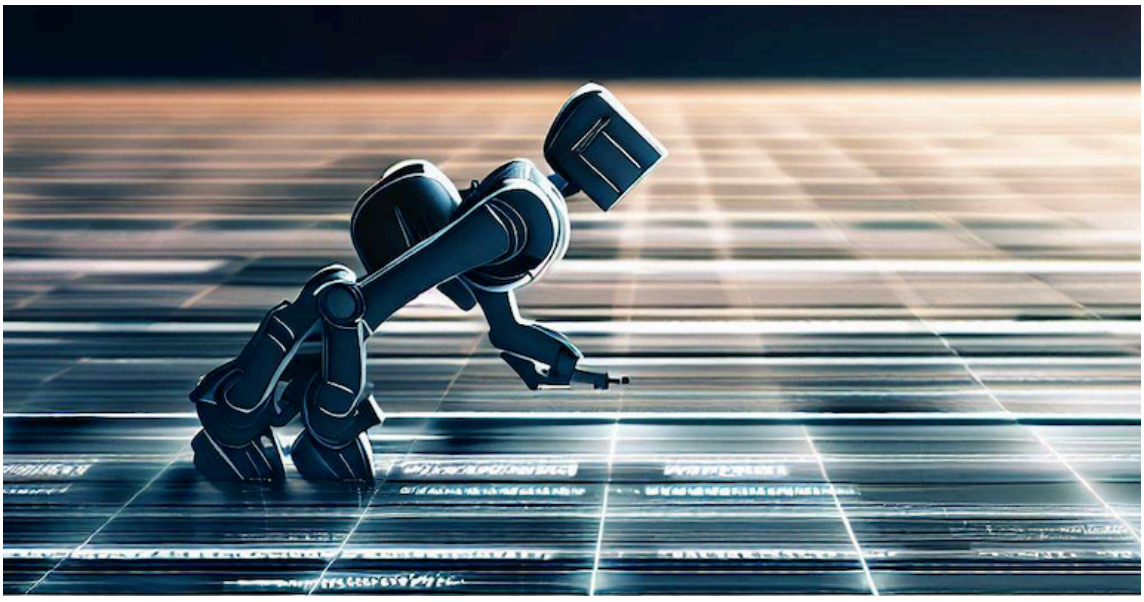
Consider a robot moving a 2D plane, guided by movement orders transmitted through a local antenna. Orders carry specific instructions, consisting of a **length** mesured in units, and an **angle** measured in degrees relative to the X-axis. The robot starts its journey at the origin, position `(0,0)`, where the antenna is located. To keep track of the signal strength, the robot has a **light** that changes colour depending on the connection strength.

There are four light colors:

- Green: Indicates that the distance between the robot and the antenna is less than 20 (<20).
- Yellow: Indicates that the distance between the robot and the antenna falls within the range of 20 to 30 (20 <= distance < 30).
- Orange: Indicates that the distance between the robot and the antenna falls within the range of 30 to 40 (30 <= distance < 40).
- Red: Indicates that the distance between the robot and the antenna is greater than or equal to 40 (>= 40).

When the robot's light turns Red, it finishes executing its current order but becomes unable to accept any further commands.

The goal is to track the robot's final position on the grid.



Task 1

Write function `trackRobotV1` passing around the state as an argument. Assume a Cartesian coordinate system where each position is represented by a pair of `Float` values, and the initial position is `(0.0, 0.0)`. The type `Robot` is as defined below. The list of commands are executed from head to tail.

```
data Light    = Green | Yellow | Orange | Red deriving (Eq, Show)

type Position = (Float, Float)
type Robot    = (Light, Position)
type Command  = (Float, Integer)
type Commands = [Command]

trackRobotV1 :: Commands -> Robot -> Robot
```

Task 2

In Lecture #12, we've created a custom type `ST` for *state transformers* and made it into a monad. We've also seen that this is already implemented in library `Monad.Control.State` with the following type synonym:

```
newtype ST a = S (s -> (a,s))
type State s a = ST a
```

Write a function `trackRobotV2` that behaves similarly to `trackRobotV1` but uses the `State` monad defined in `Monad.Control.State`. Remember, readability is key.

```
import Control.Monad.State

trackRobotV2 :: Commands -> State Robot Robot
```

If you need help understanding the code above, you may want to check the lecture notes again and Graham Hutton book Section 12.3, Part "The State Monad". Another useful link to understand functors, applicatives and monads, in a very graphical way, is the blog post by Aditya Bhargava.

Task 3

A safety protocol has been installed in the robot. As soon as the light turns Orange, the robot halts (without completing the current command) and returns to the origin. The robot remains responsive to subsequent instructions for further movement. Write `trackRobotV3`.

```
trackRobotV3 :: Commands -> State Robot Robot
```

Main template

```
main :: IO ()
main = do
    let commands = [(2, 45), (8, 90), (12, 180)
                   , (25, 90), (5, 0), (10, 90)
                   , (5, 0), (10, 45), (1, 135)]
        init = (Green, (0, 0))
        finalPositionV1 = trackRobotV1 commands init
        finalPositionV2 = evalState (trackRobotV2 commands) init
        finalPositionV3 = evalState (trackRobotV3 commands) init
    putStrLn $ "Final position V1: " ++ show finalPositionV1
    putStrLn $ "Final position V2: " ++ show finalPositionV2
    putStrLn $ "Final position V3: " ++ show finalPositionV3
```