Advent of Code 2016

Eric Bailey

Contents

DAY 1

No Time for a Taxicab

Link

Santa's sleigh uses a very high-precision clock to guide its movements, and the clock's oscillator is regulated by stars. Unfortunately, the stars have been stolen… by the Easter Bunny. To save Christmas, Santa needs you to retrieve all **fifty stars** by December 25th.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the advent calendar; the second puzzle is unlocked when you complete the first. Each puzzle grants **one star**. Good luck!

You're airdropped near *Easter Bunny Headquarters* in a city somewhere. "Near", unfortunately, is as close as you can get - the instructions on the Easter Bunny Recruiting Document the Elves intercepted start here, and nobody had time to work them out further.

The Document indicates that you should start at the given coordinates (where you just landed) and face North. Then, follow the provided sequence: either turn left (`L`) or right (`R`) 90 degrees, then walk forward the given number of blocks, ending at a new intersection.

There's no time to follow such ridiculous instructions on foot, though, so you take a moment and work out the destination. Given that you can only walk on the street grid of the city, how far is the shortest path to the destination?

For example:

- Following `R2, L3` leaves you `2` blocks East and `3` blocks North, or `5` blocks away.
- `R2, R2, R2` leaves you `2` blocks due South of your starting position, which is `2` blocks away.
- `R5, L5, R5, R3` leaves you `12` blocks away.

1. Module Declaration and Imports

```
||| Day 1: No Time for a Taxicab
module Data.Advent.Day01
```

For no compelling reason, I like to use arrows, so import the requisite modules.

```
import Control.Arrow
import Control.Category
import Data.Morphisms
import Data.SortedSet
```

For parsing, use Lightyear.

```
import public Lightyear
import public Lightyear.Char
import public Lightyear.Strings
```

2. Data Types

Ensure both the type and data constructors are exported for all the following data types.

```
%access public export
```

We'll need to keep track of which cardinal direction we're facing, so define a data type `Heading` to model that. A heading is `N`orth, `E`ast, `S`outh or `W`est.

```
||| A cardinal heading.
data Heading = ||| North
               N
             | ||| East
               E
             | ||| South
               S
             | ||| West
               W
```

As per the puzzle description, the instructions will tell us to go `L`eft or `R`ight, so model `Direction`s accordingly.

```
||| A direction to walk in, left or right.
data Direction = ||| Left
                 L
               | ||| Right
                 R
```

For convenience in the REPL, implement the `Show` interface for `Direction`.

```
implementation Show Direction where
    show L = "Left"
    show R = "Right"
```

An `Instruction`, e.g. `R2`, is comprised of a `Direction` and a number of blocks to walk in that direction.

As such, define an `Instruction` as a pair of a `Direction` and an `Integer`.

```
||| A direction and a number of blocks.
Instruction : Type
Instruction = (Direction, Integer)
```

A `Position` is a pair of a `Heading` and `Coordinates`, such that the `Heading` represents which cardinal direction we're facing and the `Coordinates` describe where we're at in the street grid.

```
||| A pair of coordinates on the street grid, `(x, y)`.
Coordinates : Type
Coordinates = (Integer, Integer)

||| A heading and coordinates.
Position : Type
Position = (Heading, Coordinates)
```

3. Parsers

Ensure the type constructors of the following parsers are exported.

```
%access export
```

A `Direction` is represented in the input by a single character, `'L'` or `'R'`.

```
direction : Parser Direction
direction = (char 'L' *> pure L) <|>|
            (char 'R' *> pure R) <?>
            "a direction"
```

An `Instruction` is represented as a `Direction` followed immediately by an integer.

```
||| Parse an instruction, i.e. a direction and
||| a number of blocks to walk in that direction.
partial instruction : Parser Instruction
instruction = [| MkPair direction integer |] <?> "an instruction"
```

The instructions are given as a comma-separated list.

```
||| Parse a comma-separated list of `Instruction`s.
partial instructions : Parser (List Instruction)
instructions = commaSep instruction <* spaces <* eof <?>
               "a comma-separated instructions (at least one)"
```

4. Logic

```
||| Return the new heading after turning right.
||| @ initial the initial heading
turnRight : (initial : Heading) -> Heading
turnRight N = E
turnRight E = S
turnRight S = W
turnRight W = N

||| Return the new heading after turning left.
||| @ initial the initial heading
turnLeft : (initial : Heading) -> Heading
turnLeft N = W
turnLeft E = N
turnLeft S = E
turnLeft W = S

||| Return the new heading after turning
||| in the direction specifed by a given direction.
turn : Instruction -> (Position ~> Position)
turn (dir,_) = first . Mor $ case dir of { L => turnLeft; R => turnRight }

syntax "(~+" [n] ")" = plusN n

||| Add `n` to a given integer. Shorthand: `(~+ n)`
plusN : Integer -> (Integer ~> Integer)
plusN = Mor . (+)

syntax "(~-" [n] ")" = plusN n

||| Subtract `n` from a given integer. Shorthand: `(~- n)`
minusN : (n : Integer) -> (Integer ~> Integer)
minusN = Mor  . (flip (-))

||| Return a morphism `Position ~> Position` that follows a given instruction,
||| i.e. maps to the new position after moving the specified number of blocks
||| with the current heading from the current coordinates.
move : Instruction -> (Position ~> Position)
move (_,n) = Mor (second . go) &&& Mor id >>> Mor (uncurry applyMor)
  where
    go : Position -> (Coordinates ~> Coordinates)
    go (N,_) = second (~+ n)
    go (E,_) = first  (~+ n)
    go (S,_) = second (~- n)
    go (W,_) = first  (~- n)
```

```
||| Follow a given instruction and return the new position.
||| @ instruction the instruction to follow
||| @ pos the initial position
follow : (instruction : Instruction) -> (pos : Position) -> Position
follow ins = applyMor $ turn ins >>> move ins

private distance' : Coordinates -> Integer
distance' = abs . uncurry (+)

||| Return the distance from coordinates `(0,0)` after following
||| a given list of instructions, starting facing North.
||| @ instructions the list of instructions to follow
distance : (instructions : List Instruction) -> Integer
distance = distance' . snd . foldl (flip follow) (N, (0, 0))
```

Define a generic main function for both parts of the puzzle that takes a function `f : List Instruction -> IO ()` and does as follows:

- Read the input from `input/day01.txt`
- Parse a list of instructions from the input
- Call `f` on the list of instructions
- If anything goes wrong, print the error and return `()`

```
private partial main' : (f : List Instruction -> IO ()) -> IO ()
main' f = do Right str <- readFile "input/day01.txt"
               | Left err => printLn err
             case parse instructions str of
                  Right is => f is
                  Left err => printLn err
```

5. Part One

> *How many blocks away* is Easter Bunny HQ?

To compute the answer to Part One, 262, simply call `main' (printLn . distance)`, i.e. compute the `distance` between the starting and final positions.

```
namespace PartOne

    partial main : IO ()
    main = main' (printLn . distance)
```

6. Part Two

Then, you notice the instructions continue on the back of the Recruiting Document. Easter Bunny HQ is actually at the first location you visit twice.

For example, if your instructions are `R8, R4, R4, R8`, the first location you visit twice is `4` blocks away, due East.

How many blocks away is the *first location you visit twice*?

```
namespace PartTwo
```

Computing the answer to Part Two, `131`, is a bit more complicated.

First, define a recursive function `partTwo` to process a list of instructions and return the distance between the starting location and the first location visited twice.

To do that, maintain a sorted set of seen locations and for each block walked per each instruction, and short circuit if any location has been seen before.

Return `Nothing` if given the empty instruction list. Otherwise, return `Just` the desired distance.

N.B. `partTwo` short circuits at the instruction level rather than block level, so it's possible a little extra work is performed.

```
partTwo : (instructions : List Instruction) ->
          (pos : Position) ->
          (seen : SortedSet Coordinates) ->
          Maybe Integer
partTwo [] _ _                              = Nothing
partTwo ((dir, len) :: is) (h, loc) seen =
    either (Just . distance')
           (partTwo is (follow (dir, len) (h, loc)))
           (foldl go (Right seen) [1..len])
  where
    go : Either Coordinates (SortedSet Coordinates) -> Integer ->
         Either Coordinates (SortedSet Coordinates)
    go dup@(Left _) _  = dup
    go (Right seen') n = let (_, loc') = follow (dir, n) (h, loc) in
                             if contains loc' seen'
                                then Left loc'
                                else Right $ insert loc' seen'

partial main : IO ()
main = main' $ \is => case partTwo is (N, (0, 0)) empty of
                           Nothing     => putStrLn "Failed!"
                           Just answer => printLn answer
```

7. Main

Label and evaluate `PartOne.main` and `PartTwo.main`.

```
namespace Main

    partial main : IO ()
    main = putStr "Part One: " *> PartOne.main *>
           putStr "Part Two: " *> PartTwo.main
```

□