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
```

Import the generic `Day` structure, inspired by Steve Purcell Haskell solution.

```
import public Data.Advent.Day
```

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 record containing a `Direction` and a number of `Blocks`.

```
Blocks : Type
Blocks = Integer

||| A direction and a number of blocks.
record Instruction where
    constructor MkIns
    iDir    : Direction
    iBlocks : Blocks
```

A `Position` is a record containing x and y coordinates in the street grid.

```
||| A pair of coordinates on the street grid.
record Position where
    constructor MkPos
    posX, posY : Blocks

implementation Eq Position where
    (MkPos x1 y1) == (MkPos x2 y2) = x1 == x2 && y1 == y2

implementation Ord Position where
    compare (MkPos x1 y1) (MkPos x2 y2) with (compare x1 x2)
      | EQ = compare y1 y2
      | o  = o
```

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 = [| MkIns 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 <?>
               "a comma-separated list of instructions"
```

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 a given direction.
turn : Heading -> Direction -> Heading
turn h L = turnLeft  h
turn h R = turnRight h

||| Return the new position after moving one block with a given heading.
move : Heading -> (Position -> Position)
move N = record { posY $= (+ 1) }
move E = record { posX $= (+ 1) }
move S = record { posY $= (flip (-) 1) }
move W = record { posX $= (flip (-) 1) }

||| Compute the shortest distance from the initial position and a given one.
shortestDistance : Position -> Blocks
shortestDistance (MkPos x y) = abs x + abs y
```

```
||| Return the list of intermediate positions visited
||| when following a given list of instructions.
followInstructions : List Instruction -> List Position
followInstructions is =
    let directions = drop 1 $ scanl turn N (iDir <$> is)
        distances  = iBlocks <$> is
        moves      = concatMap expand (zip directions distances) in
        scanl (flip move) (MkPos 0 0) moves
  where
    expand : (Heading, Blocks) -> List Heading
    expand (h, n) = replicate (cast n) h
```

5. Part One

How many blocks away is Easter Bunny HQ?

To compute the answer to Part One, 262, follow the instructions, get the last position and compute the shortest distance from the initial position.

```
partOne : List Instruction -> Maybe Blocks
partOne = map shortestDistance . last' . followInstructions
```

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*?

First, define a function `firstDuplicate` to find the first location visited twice in a list of positions. Instead of specializing for `List Position`, generalize the solution to `Ord a => List a`.

```
||| Return `Just` the first duplicate in a given list.
||| If there are none, return `Nothing`.
firstDuplicate : Ord a => List a -> Maybe a
firstDuplicate = go empty
  where
    go : SortedSet a -> List a -> Maybe a
    go _ []          = Nothing
    go seen (x::xs) =
        if contains x seen
           then Just x
           else go (insert x seen) xs

partTwo : List Instruction -> Maybe Blocks
partTwo = map shortestDistance . firstDuplicate . followInstructions
```

7. Main

Run the solutions for Day 1.

```
namespace Main

    partial main : IO ()
    main = runDay $ MkDay 1 instructions
           (pure . maybe "Failed!" show . partOne)
           (pure . maybe "Failed!" show . partTwo)
```

□

DAY 2

Bathroom Security

Link

You arrive at *Easter Bunny Headquarters* under cover of darkness. However, you left in such a rush that you forgot to use the bathroom! Fancy office buildings like this one usually have keypad locks on their bathrooms, so you search the front desk for the code.

"In order to improve security," the document you find says, "bathroom codes will no longer be written down. Instead, please memorize and follow the procedure below to access the bathrooms."

The document goes on to explain that each button to be pressed can be found by starting on the previous button and moving to adjacent buttons on the keypad: `U` moves up, `D` moves down, `L` moves left, and `R` moves right. Each line of instructions corresponds to one button, starting at the previous button (or, for the first line, *the "5" button*); press whatever button you're on at the end of each line. If a move doesn't lead to a button, ignore it.

You can't hold it much longer, so you decide to figure out the code as you walk to the bathroom. You picture a keypad like this:

```
1 2 3
4 5 6
7 8 9
```

Suppose your instructions are:

```
ULL
RRDDD
LURDL
UUUUD
```

- You start at "5" and move up (to "2"), left (to "1"), and left (you can't, and stay on "1"), so the first button is `1`.
- Starting from the previous button ("1"), you move right twice (to "3") and then down three times (stopping at "9" after two moves and ignoring the third), ending up with `9`.
- Continuing from "9", you move left, up, right, down, and left, ending with `8`.
- Finally, you move up four times (stopping at "2"), then down once, ending with `5`.

So, in this example, the bathroom code is 1985.

Your puzzle input is the instructions from the document you found at the front desk.

1. Module Declaration and Imports

```
||| Day 2: Bathroom Security
module Data.Advent.Day02

import public Data.Advent.Day
import public Data.Ix

import Data.Vect

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

2. Data Types

```
%access public export

||| Up, down, left or right.
data Instruction = ||| Up
                   U
                 | ||| Down
                   D
                 | ||| Left
                   L
                 | ||| Right
                   R

||| A single digit, i.e. a number strictly less than ten.
Digit : Type
Digit = Fin 10

implementation Show Digit where
  show = show . finToInteger

implementation [showDigits] Show (List Digit) where
  show = concatMap show

||| A pair of coordinates on the keypad, `(x, y)`.
Coordinates : Type
Coordinates = (Fin 3, Fin 3)
```

3. Parsers

```
%access export

up : Parser Instruction
up = char 'U' *> pure U <?> "up"

down : Parser Instruction
down = char 'D' *> pure D <?> "down"

left : Parser Instruction
left = char 'L' *> pure L <?> "left"

right : Parser Instruction
right = char 'R' *> pure R <?> "right"

instruction : Parser Instruction
instruction = up <|> down <|> left <|> right <?> "up, down, left or right"

partial instructions : Parser (List Instruction)
instructions = some instruction <* (skip endOfLine <|> eof)
```

4. Part One

What is the bathroom code?

```
namespace PartOne

  ||| A keypad like this:
  |||
  ||| ```
  ||| 1 2 3
  ||| 4 5 6
  ||| 7 8 9
  ||| ```
  keypad : Vect 3 (Vect 3 Digit)
  keypad = [ [1, 2, 3],
             [4, 5, 6],
             [7, 8, 9] ]

  move : Coordinates -> Instruction -> Coordinates
  move (x, y) U = (x, pred y)
  move (x, y) D = (x, succ y)
  move (x, y) L = (pred x, y)
  move (x, y) R = (succ x, y)
```

```
  button : Coordinates -> List Instruction -> (Coordinates, Digit)
  button loc@(x, y) [] = (loc, index x (index y keypad))
  button loc (i :: is) = button (move loc i) is

partial partOne : List (List Instruction) -> String
partOne = show @{showDigits} . go ((1,1), [])
  where
    go : (Coordinates, List Digit) -> List (List Instruction) -> List Digit
    go (_, ds) []            = reverse ds
    go (loc, ds) (is :: iis) = let (loc', d) = PartOne.button loc is in
                                   go (loc', d :: ds) iis

namespace PartOne

  ||| ```idris example
  ||| example
  ||| ```
  partial example : String
  example = fromEither $ partOne <$>
            parse (some instructions) "ULL\nRRDDD\nLURDL\nUUUUD"
```

5. Part Two

You finally arrive at the bathroom (it's a several minute walk from the lobby so visitors can behold the many fancy conference rooms and water coolers on this floor) and go to punch in the code. Much to your bladder's dismay, the keypad is not at all like you imagined it. Instead, you are confronted with the result of hundreds of man-hours of bathroom-keypad-design meetings:

```
    1
  2 3 4
5 6 7 8 9
  A B C
    D
```

You still start at "5" and stop when you're at an edge, but given the same instructions as above, the outcome is very different:

- You start at "5" and don't move at all (up and left are both edges), ending at `5`.
- Continuing from "5", you move right twice and down three times (through "6", "7", "B", "D", "D"), ending at `D`.
- Then, from "D", you move five more times (through "D", "B", "C", "C", "B"), ending at `B`.
- Finally, after five more moves, you end at `3`.

So, given the actual keypad layout, the code would be 5DB3.

> Using the same instructions in your puzzle input, what is the correct *bathroom code*?

```
namespace PartTwo

  keypad : Vect 5 (n ** Vect n Char)
  keypad = [ (1 **           ['1'])
           , (3 **      ['2', '3', '4'])
           , (5 ** ['5', '6', '7', '8', '9'])
           , (3 **      ['A', 'B', 'C'])
           , (1 **           ['D'])
           ]

  -- NOTE: This will wrap at the bounds, which might be unexpected.
  partial convert : (n : Nat) -> Fin m -> Fin n
  convert (S j) fm {m} =
      let delta = half $ if S j > m
                         then S j `minus` m
                         else m `minus` S j in
          the (Fin (S j)) $ fromNat $ finToNat fm `f` delta
    where
      f : Nat -> Nat -> Nat
      f = if S j > m then plus else minus
      partial half : Nat -> Nat
      half = flip div 2

  canMoveVertically : (Fin (S k), Fin 5) -> Instruction -> Bool
  canMoveVertically (x, y) i with ((finToNat x, finToNat y))
    canMoveVertically (x, y) U | (col, row) =
        case row of
             Z                  => False
             S Z                => col == 1
             S (S Z)            => inRange (1,3) col
             _                  => True
    canMoveVertically (x, y) D | (col, row) =
        case row of
             S (S Z)            => inRange (1,3) col
             S (S (S Z))        => col == 1
             S (S (S (S Z)))    => False
             _                  => True
    canMoveVertically _ _ | _ = True
```

```
  partial move : (Fin (S k), Fin 5) -> Instruction ->
    ((n ** Fin n), Fin 5)
  move (x, y) U = if canMoveVertically (x, y) U
                  then let n = fst (index (pred y) keypad) in
                           ((n ** convert n x), pred y)
                  else ((_ ** x), y)
  move (x, y) D = if canMoveVertically (x, y) D
                  then let n = fst (index (succ y) keypad) in
                           ((n ** convert n x), succ y)
                  else ((_ ** x), y)
  move (x, y) L = let n = fst (index y keypad) in
                   ((n ** convert n (pred x)), y)
  move (x, y) R = let n = fst (index y keypad) in
                   ((n ** convert n (succ x)), y)

  partial button : (Fin (S k), Fin 5) -> List Instruction ->
    (((n ** Fin n), Fin 5), Char)
  button loc@(x, y) [] =
      let (n ** row) = index y PartTwo.keypad
          xx = convert n x in
          (((n ** xx), y), index xx row)
  button loc (i :: is) =
      let ((S _ ** x), y) = move loc i in
          button (x, y) is

partial partTwo : List (List Instruction) -> String
partTwo = go (((5 ** 0),2), [])
  where
    partial go : (((n ** Fin n), Fin 5), List Char) ->
      List (List Instruction) -> String
    go (_, cs) []            = pack $ reverse cs
    go (loc, cs) (is :: iis) =
        let ((S k ** xx), y) = loc
            (loc', c) = PartTwo.button (xx, y) {k=k} is in
            go (loc', c :: cs) iis

namespace PartTwo

  ||| ```idris example
  ||| PartTwo.example
  ||| ```
  partial example : String
  example = fromEither $ partTwo <$>
            parse (some instructions) "ULL\nRRDDD\nLURDL\nUUUUD"
```

6. Main

```
namespace Main

  partial main : IO ()
  main = runDay $ MkDay 2 (some instructions)
         (pure . partOne)
         (pure . partTwo)
```