

# **Advent of Code 2016**

Eric Bailey



Day 1. No Time for a Taxicab	1
1. Module Declaration and Imports	2
2. Data Types	2
3. Parsers	3
4. Logic	4
5. Part One	5
6. Part Two	5
7. Main	6
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 **Directions** 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)
```

□