ERIC BAILEY

ADVENT OF CODE

Contents

Day 2: Dive!

https://adventofcode.com/2021/day/2

Now, you need to figure out how to pilot this thing.

It seems like the submarine can take a series of commands like `forward 1`, `down 2`, or `up 3`:

- `forward x` increases the horizontal position by `x` units.
- `down x` increases the depth by `x` units.
- `up x` decreases the depth by `x` units.

Note that since you're on a submarine, `down` and `up` affect your **depth**, and so they have the opposite result of what you might expect.

The submarine seems to already have a planned course (your puzzle input). You should probably figure out where it's going. For example:

```
forward 5
down 5
forward 8
up 3
down 8
forward 2
```

Your horizontal position and depth both start at `0`. The steps above would then modify them as follows:

- `forward 5` adds `5` to your horizontal position, a total of `5`.
- `down 5` adds `5` to your depth, resulting in a value of `5`.
- `forward 8` adds `8` to your horizontal position, a total of `13`.
- `up 3` decreases your depth by `3`, resulting in a value of `2`.
- `down 8` adds `8` to your depth, resulting in a value of `10`.
- `forward 2` adds `2` to your horizontal position, a total of `15`.

After following these instructions, you would have a horizontal position of 15 and a depth of 10. (Multiplying these together produces `150`.)

Calculate the horizontal position and depth you would have after following the planned course. **What do you get if you multiply your final horizontal position by your final depth?**

Part Two

Based on your calculations, the planned course doesn't seem to make any sense. You find the submarine manual and discover that the process is actually slightly more complicated.

In addition to horizontal position and depth, you'll also need to track a third value, **aim**, which also starts at `0`. The commands also mean something entirely different than you first thought:

- `down x` increases your aim by `x` units.
- `up x` decreases your aim by `x` units.
- `forward x` does two things:
  - It increases your horizontal position by `x` units.
  - It increases your depth by your aim **multiplied by** `x`.

Again note that since you're on a submarine, `down` and `up` do the opposite of what you might expect: "down" means aiming in the positive direction.

Now, the above example does something different:

- `forward 5` adds `5` to your horizontal position, a total of `5`. Because your aim is `0`, your depth does not change.
- `down 5` adds `5` to your aim, resulting in a value of `5`.
- `forward 8` adds `8` to your horizontal position, a total of `13`. Because your aim is `5`, your depth increases by $8 * 5 = 40$.
- `up 3` decreases your aim by `3`, resulting in a value of `2`.
- `down 8` adds `8` to your aim, resulting in a value of `10`.
- `forward 2` adds `2` to your horizontal position, a total of `15`. Because your aim is `10`, your depth increases by $2 * 10 = 20$ to a total of `60`.

After following these new instructions, you would have a horizontal position of `15` and a depth of `60`. (Multiplying these produces `900`.)

Using this new interpretation of the commands, calculate the horizontal position and depth you would have after following the planned course. **What do you get if you multiply your final horizontal position by your final depth?**

Haskell solution

A `Direction` is a change in horizontal position and a change in depth, represented by a 2-dimensional vector[1], monoidal under addition[2].

[1] https://hackage.haskell.org/package/linear/docs/Linear-V2.html#t:V2
[2] https://hackage.haskell.org/package/base/docs/Data-Monoid.html#t:Sum

⟨*Define some data types* 7a⟩≡
```
newtype Direction = Direction {unDirection :: V2 Int}
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
    via (Sum (V2 Int))
```

This definition is continued in chunk 8.
This code is used in chunk 10.

The ⟨*known directions* 7b⟩ are `forward`, `down`, and `up`.

⟨*known directions* 7b⟩≡
```
forward, down, up :: Int -> Direction
```
This definition is continued in chunk 7.
This code is used in chunk 7f.

`forward x` increases the horizontal position by `x` units.

⟨*known directions* 7b⟩+≡
```
forward = Direction . flip V2 0
```
This code is used in chunk 7f.

`down x` increases the depth by `x` units.

⟨*known directions* 7b⟩+≡
```
down = Direction . V2 0
```
This code is used in chunk 7f.

`up x` decreases the depth by `x` units, i.e. `down` with a `negate`d `x`.

⟨*known directions* 7b⟩+≡
```
up = down . negate
```
This code is used in chunk 7f.

Define a `Direction` parser using the ⟨*known directions* 7b⟩.

⟨*Define a Direction parser* 7f⟩≡
```
direction :: Parser Direction
direction = dir <*> (fromInteger <$> natural)
  where
    dir =
      symbol "forward" $> forward
        <|> symbol "down" $> down
        <|> symbol "up" $> up
```

⟨*known directions* 7b⟩
This code is used in chunk 10.

The puzzle input is a list of `Direction`s.

⟨*Parse the input* 7g⟩≡
```
getInput :: IO [Direction]
getInput = parseInput (some direction) $(inputFilePath)
```
Root chunk (not used in this document).

General solution

The general solution of the puzzle is to sum a list of additive monoids, extract the final position, and compute the `product` of the horizontal position and depth.

⟨*Solve the puzzle* 8a⟩≡
```
solve :: Monoid m => (m -> V2 Int) -> [m] -> Int
solve extract = product . extract . mconcat
```
This code is used in chunk 10.

Part One

For Part One, the additive monoid is `Direction`.

⟨*Solve Part One* 8b⟩≡
```
partOne :: [Direction] -> Int
partOne = solve unDirection
```
This code is used in chunk 10.

Part Two

For Part Two, the additive monoid is `Aim`, i.e. an integer.

⟨*Define some data types* 7a⟩+≡
```
newtype Aim = Aim Int
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
    via (Sum Int)
```
This code is used in chunk 10.

`forward x` increases the horizontal position by `x` units and increases the depth by the aim multiplied by `x`, forming a semi-direct product[3] of `Direction` (the sub-monoid) and `Aim` (the quotient monoid).

[3] https://hackage.haskell.org/package/monoid-extras/docs/Data-Monoid-SemiDirectProduct.html#t:Semi

Define how `Aim` acts on `Direction`.

⟨*Define some data types* 7a⟩+≡
```
instance Action Aim Direction where
  act (Aim a) (Direction (V2 x y)) = Direction (V2 x (y + a * x))
```
This code is used in chunk 10.

Use the **`Action`** to construct the semi-direct product `Direction` $\rtimes_\phi$ `Aim`.

⟨*Define the semi-direct product* 8e⟩≡
```
phi :: Direction -> Semi Direction Aim
```
This definition is continued in chunks 8 and 9.
This code is used in chunk 9c.

`forward`, i.e. a `Direction` with a depth change of `0`, doesn't affect the aim.

⟨*Define the semi-direct product* 8e⟩+≡
```
phi dir@(Direction (V2 _ 0)) = inject dir
```
This code is used in chunk 9c.

`up` or `down`, i.e. a `Direction` with a horizontal change of `0` and a non-zero depth change `y`, results in an aim change of `y` units.

⟨*Define the semi-direct product* 8e⟩+≡
```
phi (Direction (V2 0 y)) = embed (Aim y)
```
This code is used in chunk 9c.

Since `Direction` is not specific enough to prevent them, add a catch-all clause to handle invalid directions, e.g. `forward` and `up` simultaneously.

⟨*Define the semi-direct product* 8e⟩+≡
```
phi _ = error "Invalid direction"
```
This code is used in chunk 9c.

To solve Part Two, lift each `Direction` in the input to `Direction` $\rtimes_\phi$ `Aim`. To extract the final position, forget the `Aim` tag.

⟨*Solve Part Two* 9c⟩≡
```
partTwo :: [Direction] -> Int
partTwo = solve (unDirection . untag) . map phi
  where
    ⟨Define the semi-direct product 8e⟩
```
This code is used in chunk 10.

Full solution

⟨*Day02.hs* 10⟩≡

```
{-# LANGUAGE DerivingVia #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module AdventOfCode.Year2021.Day02 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMain, inputFilePath)
import Control.Applicative ((<|>))
import Data.Functor (($>))
import Data.Monoid.Action (Action (..))
import Data.Monoid.SemiDirectProduct.Strict (Semi, embed, inject, untag)
import Data.Semigroup (Sum (..))
import Linear (V2 (..))
import Text.Trifecta (Parser, natural, some, symbol)

⟨Define some data types 7a⟩

main :: IO ()
main = $(defaultMain)

getInput :: IO [Direction]
getInput = parseInput (some direction) $(inputFilePath)

example :: [Direction]
example =
  [ forward 5,
    down 5,
    forward 8,
    up 3,
    down 8,
    forward 2
  ]

⟨Solve Part One 8b⟩

⟨Solve Part Two 9c⟩

⟨Solve the puzzle 8a⟩

⟨Define a Direction parser 7f⟩
```

Root chunk (not used in this document).

Chunks

To-Do