

ERIC BAILEY

ADVENT OF CODE

Contents

2018 5

Day 1: Chronal Calibration 6

Haskell Solution 6

Day 2: Inventory Management System 8

Haskell solution 8

2019 11

Day 1: The Tyranny of the Rocket Equation 12

GAP Solution 12

Day 4: Secure Container 13

Haskell Solution 13

Day 8: 15

Haskell solution 15

2021 19

Day 2: Dive! 20

Haskell solution 22

2018

Day 1: Chronal Calibration

Copy description

<https://adventofcode.com/2018/day/1>*Haskell Solution*

A FREQUENCY CHANGE is represented by a summable integer.

<Define data types to model the puzzle input. 6a>≡

```
newtype FrequencyChange = FrequencyChange
    {unFrequencyChange :: Integer}
    deriving stock
    (Eq, Ord, Show)
    deriving
    (Semigroup, Monoid)
    via (Sum Integer)
```

This code is used in chunk 7.

PARSING THE PUZZLE INPUT for Day 1 is easy. The frequency changes are represented by signed integers, e.g.

```
parseString frequencyChanges mempty "+1\n-2\n+3" =
Success [Sum {getSum = 1},Sum {getSum = -2},Sum {getSum = 3}]
```

<Parse the input. 6b>≡

```
getInput :: IO [FrequencyChange]
getInput = parseInput (some (FrequencyChange <$> integer)) $(inputFilePath)
```

This code is used in chunk 7.

COMPUTING THE ANSWER FOR PART ONE is also a cinch: just sum the changes in frequency.

<Solve parts one and two. 6c>≡

```
partOne :: [FrequencyChange] → Maybe Integer
partOne = Just . unFrequencyChange . mconcat
```

This definition is continued in chunk 6d.

This code is used in chunk 7.

TO SOLVE PART TWO, compute the list of frequencies reached and find the first duplicate.

<Solve parts one and two. 6c>+≡

```
partTwo :: [FrequencyChange] → Maybe Integer
partTwo =
    fmap unFrequencyChange
    . findFirstDup
    . scan
    . cycle
```

This code is used in chunk 7.

BRING IT all together.

```

<Day01.hs 7>≡
{-# LANGUAGE DerivingVia #-}

module AdventOfCode.Year2018.Day01 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMainMaybe, inputFilePath)
import AdventOfCode.Util (findFirstDup, scan)
import Data.Monoid (Sum (..))
import Text.Trifecta (integer, some)

<Define data types to model the puzzle input. 6a>

main :: IO ()
main = $(defaultMainMaybe)

<Parse the input. 6b>

<Solve parts one and two. 6c>
Root chunk (not used in this document).

```

Day 2: Inventory Management System

Copy description

<https://adventofcode.com/2018/day/2>*Haskell solution*

DEFINE SOME CONVENIENT type aliases.

A **BoxID** is just a **String**, and a **Checksum** is just an **Integer**.

(Type aliases 8a)≡

```
type BoxID = String
```

```
type Checksum = Integer
```

This code is used in chunk 9.

TO SOLVE PART ONE, **Just** compute the **checksum**.¹

¹ See what I did there?

(Compute the checksum. 8b)≡

```
checksum :: [BoxID] → Checksum
```

```
checksum =
```

```
  fmap frequencies
```

```
    >> filter (elem 2) &&& filter (elem 3)
```

```
    >> length *** length
```

```
    >> product
```

```
    >> fromIntegral
```

This code is used in chunk 9.

(Part One 8c)≡

```
partOne :: [BoxID] → Maybe Checksum
```

```
partOne = Just . checksum
```

This code is used in chunk 9.

SOLVE Part Two.

(Correct the box IDs. 8d)≡

```
correctBoxIDs :: [BoxID] → Maybe (BoxID, BoxID)
```

```
correctBoxIDs = listToMaybe . mapMaybe go . tails
```

```
  where
```

```
    go (x : xs@( _ : _)) = (x,) <$> find (hammingSimilar 1 x) xs
```

```
    go _ = Nothing
```

This code is used in chunk 9.

(Part Two 8e)≡

```
partTwo :: [BoxID] → Maybe String
```

```
partTwo = fmap (uncurry intersect) . correctBoxIDs
```

This code is used in chunk 9.

BRING IT all together.

```

<Day02.hs 9>≡
{-# LANGUAGE TupleSections #-}

module AdventOfCode.Year2018.Day02 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMainMaybe, inputFilePath)
import AdventOfCode.Util (frequencies, hammingSimilar)
import Control.Arrow ((&&&), (**), (»>))
import Data.List (find, intersect, tails)
import Data.Maybe (listToMaybe, mapMaybe)
import Text.Trifecta (letter, newline, sepEndBy, some)

<Type aliases 8a>

main :: IO ()
main = $(defaultMainMaybe)

getInput :: IO [BoxID]
getInput = parseInput (some letter `sepEndBy` newline) $(inputFilePath)

<Part One 8c>

<Part Two 8e>

<Compute the checksum. 8b>

<Correct the box IDs. 8d>
Root chunk (not used in this document).

```


2019

Day 1: The Tyranny of the Rocket Equation

Copy description

<https://adventofcode.com/2019/day/1>*GAP Solution*

fuel := mass\3 - 2

```

<Day01.g 12a>≡
  FuelRequiredModule := function( mass )
    return Int( Float( mass / 3 ) ) - 2;
  end;;

```

This definition is continued in chunk 12.
 Root chunk (not used in this document).

```

<Day01.g 12a>+≡
  PartOne := function( )
    local input, line, mass, sum;;
    sum := 0;
    input := InputTextFile ( "./input/day01.txt" );
    line := ReadLine( input );
    repeat
      mass := Int( Chomp( line ) );
      sum := sum + FuelRequiredModule( mass );
      line := ReadLine( input );
    until line = fail or IsEndOfStream( input );
    return sum;
  end;;

```

```

<Day01.g 12a>+≡
  TotalFuelRequiredModule := function( mass )
    local fuel;;
    fuel := FuelRequiredModule( mass );
    if IsPosInt( fuel ) then
      return fuel + TotalFuelRequiredModule( fuel );
    else
      return 0;
    fi;
  end;;

```

```

<Day01.g 12a>+≡
  PartTwo := function( )
    local input, line, mass, sum;;
    sum := 0;
    input := InputTextFile ( "./input/day01.txt" );
    line := ReadLine( input );
    repeat
      mass := Int( Chomp( line ) );
      sum := sum + TotalFuelRequiredModule( mass );
      line := ReadLine( input );
    until line = fail or IsEndOfStream( input );
    return sum;
  end;;

```

Day 4: Secure Container

Copy description

<https://adventofcode.com/2019/day/4>*Haskell Solution*

MY PUZZLE INPUT was the range 236491-713787, which I converted into a list of lists of `digits`.

```
<Input 13a>≡
  getInput :: IO [[Int]]
  getInput = pure $ reverse . digits 10 <$> [236491 .. 713787]
```

This code is used in chunk 14.

SPOILER: Parts One and Two vary only in the strictness of the definition of a double, so a generic solver can be parameterized by the binary operation to compare the number of adjacent digits that are the same with 2. In both parts of the puzzle, it must also be the case that the digits never decrease, i.e. the password `isSorted`.

```
<Generic solver 13b>≡
  solve :: (Int → Int → Bool) → [[Int]] → Int
  solve = count . (isSorted <&&>) . hasDouble
  where
    hasDouble cmp = any (('cmp' 2) . length) . group
```

This code is used in chunk 14.

FOR PART ONE, there must be a double, i.e. at least two adjacent digits that are the same.

```
<Part One 13c>≡
  partOne :: [[Int]] → Int
  partOne = solve (≥)
```

This definition is continued in chunk 16.

This code is used in chunks 14 and 17.

FOR PART TWO, the password must have a strict double.

```
<Part Two 13d>≡
  partTwo :: [[Int]] → Int
  partTwo = solve (==)
```

This definition is continued in chunk 16e.

This code is used in chunks 14 and 17.

BRING IT all together.

```

<Day04.hs 14>≡
  module AdventOfCode.Year2019.Day04 where

  import AdventOfCode.TH (defaultMain)
  import AdventOfCode.Util (count, (<&&>))
  import Data.FastDigits (digits)
  import Data.List (group)
  import Data.List.Ordered (isSorted)

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

```

<Input 13a>

<Part One 13c>

<Part Two 13d>

<Generic solver 13b>

Root chunk (not used in this document).

*Day 8:**Haskell solution*

Add missing title

Copy description

<https://adventofcode.com/2019/day/8>

A `PIXEL` can be black, white, or transparent.

```
<Define a Pixel data type 15a>≡
data Pixel
  = Black
  | White
  | Transparent
  deriving (Enum, Eq)
```

This code is used in chunk 17.

Show black pixels as spaces, white ones as hashes, and transparent as dots.

```
<Implement Show for Pixel 15b>≡
instance Show Pixel where
  show Black = " "
  show White = "#"
  show Transparent = "."
```

This code is used in chunk 17.

DEFINE A `Layer` as a list of `Rows`, and a `Row` as a list of `Pixels`.

```
<Define a few convenient type aliases 15c>≡
type Image = [Layer]

type Layer = [Row]

type Row = [Pixel]
```

This code is used in chunk 17.

PARSE AN `Image`, i.e. one or more `Layers` comprised of `height Rows` of `width Pixels`.

```
<Parse an image 15d>≡
image :: Int → Int → Parser Image
image width height = some layer
  where
    layer :: Parser Layer
    layer = count height row
    row :: Parser Row
    row = count width pixel
```

This code is used in chunk 17.

Parse an encoded black, white, or transparent pixel.

```
<Parse a pixel 15e>≡
pixel :: Parser Pixel
pixel =
  (char '0' *> pure Black <?> "A black pixel")
  <|> (char '1' *> pure White <?> "A white pixel")
  <|> (char '2' *> pure Transparent <?> "A transparent pixel")
```

This code is used in chunk 17.

SOLVE Part One.

(Part One 13c)+≡

```
partOne :: Image → Int
```

This code is used in chunks 14 and 17.

Return the product of the number of ones (**White** pixels) and the number of twos (**Transparent** pixels) in the **layer** with the fewest **Black** pixels.

(Part One 13c)+≡

```
partOne layers = numberOf White layer * numberOf Transparent layer
  where
```

This code is used in chunks 14 and 17.

Find the **layer** with the fewest zeros, i.e. **Black** pixels.

sp?

(Part One 13c)+≡

```
layer = minimumBy (compare 'on' numberOf Black) layers
```

This code is used in chunks 14 and 17.

Return the number of elements equivalent to a given one, in a given list of lists of elements of the same type. More specifically, return the number of **Pixels** of a given color in a given **Layer**.

There's gotta be a Data.List function for this..

(Part One 13c)+≡

```
numberOf :: Eq a ⇒ a → [[a]] → Int
numberOf x = sum . fmap (length . filter (== x))
```

This code is used in chunks 14 and 17.

SOLVE Part Two.

(Part Two 13d)+≡

```
partTwo :: Image → String
partTwo layers =
  unlines . map (concatMap show) $
    foldl decodeLayer (transparentLayer 25 6) layers
  where
    decodeLayer :: Layer → Layer → Layer
    decodeLayer = zipWith (zipWith decodePixel)
    decodePixel :: Pixel → Pixel → Pixel
    decodePixel Transparent below = below
    decodePixel above _ = above
```

This code is used in chunks 14 and 17.

DEFINE A HELPER FUNCTION to create a transparent layer.

(A transparent layer 16f)+≡

```
transparentLayer :: Int → Int → Layer
transparentLayer width height = replicate height (replicate width Transparent)
```

This code is used in chunk 17.

Add some prose here.

```

<Day08.hs 17>≡
  module AdventOfCode.Year2019.Day08 where

  import AdventOfCode.Input (parseInput)
  import AdventOfCode.TH (defaultMain, inputFilePath)
  import Control.Applicative ((<|>))
  import Data.Function (on)
  import Data.List (minimumBy)
  import Text.Trifecta (Parser, char, count, some, (<?>))

  <Define a Pixel data type 15a>

  <Implement Show for Pixel 15b>

  <Define a few convenient type aliases 15c>

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

  getInput :: IO Image
  getInput = parseInput (image 25 6) $(inputFilePath)

  <Part One 13c>

  <Part Two 13d>

  <Parse an image 15d>

  <Parse a pixel 15e>

  <A transparent layer 16f>

```

Root chunk (not used in this document).

2021

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 vector², monoidal under addition³.

```
<Define some data types 22a>≡
newtype Direction = Direction {unDirection :: V2 Int}
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
  via (Sum (V2 Int))
```

This definition is continued in chunk 23.

This code is used in chunk 25.

The *<known directions 22b>* are **forward**, **down**, and **up**.

```
<known directions 22b>≡
forward, down, up :: Int → Direction
```

This definition is continued in chunk 22.

This code is used in chunk 22f.

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

```
<known directions 22b>+≡
forward = Direction . flip V2 0
```

This code is used in chunk 22f.

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

```
<known directions 22b>+≡
down = Direction . V2 0
```

This code is used in chunk 22f.

up **x** decreases the depth by **x** units, i.e. **down** with a negated **x**.

```
<known directions 22b>+≡
up = down . negate
```

This code is used in chunk 22f.

Define a **Direction** parser using the *<known directions 22b>*.

```
<Define a Direction parser 22f>≡
direction :: Parser Direction
direction = dir <*> (fromInteger <$> natural)
  where
    dir =
      symbol "forward" $> forward
      <|> symbol "down" $> down
      <|> symbol "up" $> up
```

<known directions 22b>

This code is used in chunk 25.

The puzzle input is a list of **Directions**.

```
<Parse the input 22g>≡
getInput :: IO [Direction]
getInput = parseInput (some direction) $(inputFilePath)
```

Root chunk (not used in this document).

² <https://hackage.haskell.org/package/linear/docs/Linear-V2.html#t:V2>

³ <https://hackage.haskell.org/package/base/docs/Data-Monoid.html#t:Sum>

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 23a>≡
  solve :: Monoid m => (m -> V2 Int) -> [m] -> Int
  solve extract = product . extract . mconcat
```

This code is used in chunk 25.

FOR PART ONE, the additive monoid is **Direction**.

```
<Solve Part One 23b>≡
  partOne :: [Direction] -> Int
  partOne = solve unDirection
```

This code is used in chunk 25.

FOR PART TWO, the additive monoid is **Aim**, i.e. an integer.

```
<Define some data types 22a>+≡
  newtype Aim = Aim Int
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
  via (Sum Int)
```

This code is used in chunk 25.

forward **x** increases the horizontal position by **x** units and increases the depth by the aim multiplied by **x**, forming a **semi-direct product**⁴ of **Direction** (the sub-monoid) and **Aim** (the quotient monoid).

Define how **Aim** acts on **Direction**.

```
<Define some data types 22a>+≡
  instance Action Aim Direction where
    act (Aim a) (Direction (V2 x y)) = Direction (V2 x (y + a * x))
```

This code is used in chunk 25.

Use the **Action** to construct the semi-direct product **Direction** \rtimes **Aim**.

```
<Define the semi-direct product 23e>≡
  lift :: Direction -> Semi Direction Aim
```

This definition is continued in chunks 23 and 24a.

This code is used in chunk 24b.

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

```
<Define the semi-direct product 23e>+≡
  lift dir@(Direction (V2 _ 0)) = inject dir
```

This code is used in chunk 24b.

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 23e>+≡
  lift (Direction (V2 0 y)) = embed (Aim y)
```

This code is used in chunk 24b.

⁴ <https://hackage.haskell.org/package/monoid-extras/docs/Data-Monoid-SemiDirectProduct.html#t:Semi>

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 23e>+≡
lift _ = error "Invalid direction"
```

This code is used in chunk 24b.

To solve Part Two, `lift` each `Direction` in the input to `Direction × Aim`, forgetting the `Aim` tag to extract the final position.

```
<Solve Part Two 24b>≡
partTwo :: [Direction] → Int
partTwo = solve (unDirection . untag) . map lift
  where
    <Define the semi-direct product 23e>
```

This code is used in chunk 25.

BRING IT all together.

```

<Day02.hs 25>≡
{-# 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 22a>

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 23b>

<Solve Part Two 24b>

<Solve the puzzle 23a>

<Define a Direction parser 22f>

```

Root chunk (not used in this document).

Chunks

⟨Compute the checksum. 8b⟩	⟨Generic solver 13b⟩
⟨Correct the box IDs. 8d⟩	⟨Implement Show for Pixel 15b⟩
⟨Day01.hs 7⟩	⟨Input 13a⟩
⟨Day02.hs 9⟩	⟨Parse a pixel 15e⟩
⟨Define data types to model the puzzle input. 6a⟩	⟨Parse an image 15d⟩
⟨Parse the input. 6b⟩	⟨Part One 13c⟩
⟨Part One 8c⟩	⟨Part Two 13d⟩
⟨Part Two 8e⟩	⟨Day02.hs 25⟩
⟨Solve parts one and two. 6c⟩	⟨Define a <i>Direction</i> parser 22f⟩
⟨Type aliases 8a⟩	⟨Define some data types 22a⟩
⟨A transparent layer 16f⟩	⟨Define the semi-direct product 23e⟩
⟨Day01.g 12a⟩	⟨known directions 22b⟩
⟨Day04.hs 14⟩	⟨Parse the input 22g⟩
⟨Day08.hs 17⟩	⟨Solve Part One 23b⟩
⟨Define a few convenient type aliases 15c⟩	⟨Solve Part Two 24b⟩
⟨Define a <i>Pixel</i> data type 15a⟩	⟨Solve the puzzle 23a⟩

To-Do

■ Copy description	6
■ Copy description	8
■ Copy description	12
■ Copy description	13
■ Add missing title	15
■ Copy description	15
■ sp?	16
■ There's gotta be a Data.List function for this..	16
■ Add some prose here.	17