ERIC BAILEY

ADVENT OF CODE

Contents

Day 1: The Tyranny of the Rocket Equation

Copy description

https://adventofcode.com/2019/day/1

GAP Solution

$\text{fuel} := \text{mass} \backslash 3 - 2$

⟨*Day01.g* 5a⟩≡

```
FuelRequiredModule := function( mass )
    return Int( Float( mass / 3 ) ) - 2;
end;;
```

This definition is continued in chunks 5 and 6.
Root chunk (not used in this document).

⟨*Day01.g* 5a⟩+≡

```
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* 5a⟩+≡

```
TotalFuelRequiredModule := function( mass )
    local fuel;;
    fuel := FuelRequiredModule( mass );
    if IsPosInt( fuel ) then
        return fuel + TotalFuelRequiredModule( fuel );
    else
        return 0;
    fi;
end;;
```

⟨*Day01.g* 5a⟩+≡
```
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 1: Chronal Calibration

As usual, Day 1 consists of two parts, `partOne` and `partTwo`.

⟨*Day01.hs* 7a⟩≡

```
module AdventOfCode.Year2018.Day01
  ( main,
    partOne,
    partTwo,
  )
where

⟨Import functions, operators, and types from other modules. 9b⟩

⟨Define data types to model the puzzle input. 7b⟩

⟨Define the main function 9a⟩

⟨Define parsers for handling puzzle input. 8b⟩

⟨Solve parts one and two. 8c⟩
```

Root chunk (not used in this document).

Data Types

A frequency change is represented by a (summable) integer.

Figure 1: Computing the end frequency, given a list of frequency changes.

```
endFreq :: [FrequencyChange] -> Integer
endFreq = getSum . unFrequencyChange . mconcat
```

⟨*Define data types to model the puzzle input.* 7b⟩≡

```
newtype FrequencyChange
  = FrequencyChange
      {unFrequencyChange :: Sum Integer}
  deriving (Eq, Show)
```

This definition is continued in chunks 7c and 8a.
This code is used in chunk 7a.

Describe these instances

Since `findFirstDup` uses **`HashSet`**s internally, we need to make sure **`FrequencyChange`** is **`Hashable`**.

⟨*Define data types to model the puzzle input.* 7b⟩+≡

```
instance Hashable FrequencyChange where
  hashWithSalt salt = hashWithSalt salt . getSum . unFrequencyChange
```

This code is used in chunk 7a.

⟨*Define data types to model the puzzle input.* 7b⟩+≡

```
instance Semigroup FrequencyChange where
  (FrequencyChange x) <> (FrequencyChange y) = FrequencyChange (x <> y)

instance Monoid FrequencyChange where
  mempty = FrequencyChange (Sum 0)
```

This code is used in chunk 7a.

Parsing

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}]
```

⟨*Define parsers for handling puzzle input.* 8b⟩≡

```
frequencyChange :: Parser FrequencyChange
frequencyChange = FrequencyChange . Sum <$> integer
```

This code is used in chunk 7a.

Part One

Computing the answer for Part One is also a cinch. We just need to parse the sequence of changes in frequency, then sum them.

⟨*Solve parts one and two.* 8c⟩≡

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

This definition is continued in chunk 8d.
This code is used in chunk 7a.

Part Two

⟨*Solve parts one and two.* 8c⟩+≡

```
partTwo :: [FrequencyChange] → Maybe Integer
partTwo =
  ⟨Compute the list of frequencies reached 8e⟩
    >>> ⟨Find the first duplicate 8f⟩
    >>> ⟨Unbox the result 8g⟩
```

This code is used in chunk 7a.

⟨*Compute the list of frequencies reached* 8e⟩≡

```
scan . cycle
```

This code is used in chunk 8d.

⟨*Find the first duplicate* 8f⟩≡

```
findFirstDup
```

This code is used in chunk 8d.

⟨*Unbox the result* 8g⟩≡

```
fmap (getSum . unFrequencyChange)
```

This code is used in chunk 8d.

Main

⟨*Define the main function* 9a⟩≡

```
main :: IO ()
main = do
  input <- parseInput (some frequencyChange) $(inputFilePath)
  putStr "Part One: "
  print (partOne input)
  putStr "Part Two: "
  putStrLn $ maybe "failed!" show (partTwo input)
```

This definition is continued in chunk 14d.
This code is used in chunks 7a and 13a.

Imports

⟨*Import functions, operators, and types from other modules.* 9b⟩≡

```
import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (inputFilePath)
import AdventOfCode.Util (findFirstDup, scan)
import Control.Category ((>>>))
import Data.Hashable (Hashable (..))
import Data.Monoid (Sum (..))
import Text.Trifecta (Parser, integer, some)
```

This code is used in chunk 7a.

Day 2: 1202 Program Alarm

Copy description

https://adventofcode.com/2019/day/2

Day 2: Inventory Management System

⟨*Day02.hs* 13a⟩≡

```
module AdventOfCode.Year2018.Day02
  ( main,
    partOne,
    partTwo,
  )
where

⟨Imports 14e⟩

⟨Types and parsers 13b⟩

⟨Part One 13c⟩

⟨Part Two 14b⟩

⟨Define the main function 9a⟩
```

Root chunk (not used in this document).

Type aliases and parsers

⟨*Types and parsers* 13b⟩≡

```
type BoxID = String

boxID :: Parser BoxID
boxID = some letter

type Checksum = Integer
```

This code is used in chunk 13a.

Part One

⟨*Part One* 13c⟩≡

```
checksum :: [BoxID] -> Checksum
checksum =
  fmap frequencies
    >>> filter (elem 2) &&& filter (elem 3)
    >>> length *** length
    >>> product
    >>> fromIntegral
```

This definition is continued in chunks 14a, 15c, and 18.
This code is used in chunks 13a, 16b, and 20.

⟨*Part One* 13c⟩+≡

```
partOne :: [BoxID] -> Checksum
partOne = checksum
```

This code is used in chunks 13a, 16b, and 20.

Part Two

⟨*Part Two* 14b⟩≡

```
correctBoxIDs :: [BoxID] -> Maybe (BoxID, BoxID)
correctBoxIDs = listToMaybe . mapMaybe go . tails
  where
    go (x : xs@(_ : _)) = (,) <$> pure x <*> find (hammingSimilar 1 x) xs
    go _ = Nothing
```

This definition is continued in chunks 14c, 16a, and 19a.
This code is used in chunks 13a, 16b, and 20.

⟨*Part Two* 14b⟩+≡

```
partTwo :: [BoxID] -> Maybe String
partTwo = fmap (uncurry intersect) . correctBoxIDs
```

This code is used in chunks 13a, 16b, and 20.

Main

⟨*Define the main function* 9a⟩+≡

```
main :: IO ()
main = do
  input <- parseInput (boxID ‘sepEndBy‘ newline) $(inputFilePath)
  putStr ”Part One: ”
  print (partOne input)
  putStr ”Part Two: ”
  putStrLn (fromMaybe ”failed!” (partTwo input))
```

This code is used in chunks 7a and 13a.

Imports

⟨*Imports* 14e⟩≡

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

This code is used in chunk 13a.

Day 4: Secure Container

Copy description

https://adventofcode.com/2019/day/4

Haskell Solution

Input

My puzzle input was the range 236491-713787, which I converted into a list of lists of digits.

⟨*Input* 15a⟩≡

```haskell
input :: [[Int]]
input = digits 10 <$> [236491 .. 713787]
```

This code is used in chunk 16b.

Part One

For part one, there must be two adjacent digits that are the same, i.e. there exists at least one group of length ≥ 2.

⟨*has a double* 15b⟩≡

```haskell
any ((≥ 2) . length) . group
```

Root chunk (not used in this document).

It must also be the case that the digits never decrease, i.e. the password isSorted.

⟨*Part One* 13c⟩+≡

```haskell
partOne :: Int
partOne = length $ filter isPossiblePassword input
  where
    isPossiblePassword :: [Int] → Bool
    isPossiblePassword = liftM2 (&&) isSorted hasDouble
    hasDouble :: Eq a ⇒ [a] → Bool
    hasDouble = any ((≥ 2) . length) . group
```

This code is used in chunks 13a, 16b, and 20.

Part Two

For part two, the password still isSorted, but must also have a strict double, i.e. at least one group of length ≡ 2.

⟨*has a strict double* 15d⟩≡

```haskell
any ((≡ 2) . length) . group
```

Root chunk (not used in this document).

⟨*Part Two* 14b⟩+≡

```
partTwo :: Int
partTwo = length $ filter isPossiblePassword input
  where
    isPossiblePassword :: [Int] -> Bool
    isPossiblePassword = liftM2 (&&) isSorted hasDouble
    hasDouble :: Eq a => [a] -> Bool
    hasDouble = any ((== 2) . length) . group
```

This code is used in chunks 13a, 16b, and 20.

Full Solution

⟨*Day04.hs* 16b⟩≡

```
module AdventOfCode.Year2019.Day04 where

import Control.Monad (liftM2)
import Data.Digits (digits)
import Data.List (group)
import Data.List.Ordered (isSorted)

⟨Input 15a⟩

⟨Part One 13c⟩

⟨Part Two 14b⟩
```

Root chunk (not used in this document).

Day 8:

Add missing title

Copy description

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

Haskell solution

Pixels

A pixel can be black, white, or transparent.

⟨*Define a Pixel data type* 17a⟩≡

```
data Pixel
  = Black
  | White
  | Transparent
  deriving (Enum, Eq)
```

This code is used in chunk 20.

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

⟨*Implement* ***Show*** *for* `Pixel` 17b⟩≡

```
instance Show Pixel where
  show Black = " "
  show White = "#"
  show Transparent = "."
```

This code is used in chunk 20.

Type aliases

Define a `Layer` as a list of `Row`s, and a `Row` as a list of `Pixel`s.

⟨*Define a few convenient type aliases* 17c⟩≡

```
type Image = [Layer]

type Layer = [Row]

type Row = [Pixel]
```

This code is used in chunk 20.

Parsers

Parse an `Image`, i.e. one or more `Layer`s comprised of `height` `Row`s of `width` `Pixel`s.

⟨*Parse an image* 18a⟩≡

```
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 20.

Parse an encoded black, white, or transparent pixel.

⟨*Parse a pixel* 18b⟩≡

```
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 20.

Part One

⟨*Part One* 13c⟩+≡

```
partOne :: IO Int
partOne =
  do
    ⟨Parse a 25 × 6 image from the input 18d⟩
```

This code is used in chunks 13a, 16b, and 20.

⟨*Parse a* 25 × 6 `image` *from the input* 18d⟩≡

```
layers <- parseInput (image 25 6) "input/2019/day08.txt"
```

This code is used in chunk 18c.

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

sp?

⟨*Part One* 13c⟩+≡

```
    let layer = head $ sortBy (compare `on` numberOf Black) layers
```

This code is used in chunks 13a, 16b, and 20.

Return the product of the number of ones (`White` pixels) and the number of twos (`Transparent` pixels) in that `layer`.

⟨*Part One* 13c⟩+≡

```
    let ones = numberOf White layer
    let twos = numberOf Transparent layer
    pure $ ones * twos
```

This code is used in chunks 13a, 16b, and 20.

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 `Pixel`s of a given color in a given `Layer`.

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

⟨*Part One* 13c⟩+≡

```
  where
    numberOf :: Eq a => a -> [[a]] -> Int
    numberOf x = sum . fmap (length . filter (== x))
```

This code is used in chunks 13a, 16b, and 20.

Part Two

⟨*Part Two* 14b⟩+≡

```
partTwo :: IO String
partTwo =
  do
    layers ← parseInput (image 25 6) ”input/2019/day08.txt”
    pure
      $ 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 13a, 16b, and 20.

Miscellaneous

⟨*A transparent layer* 19b⟩≡

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

This code is used in chunk 20.

Full solution

```
⟨Day08.hs 20⟩≡
  module AdventOfCode.Year2019.Day08
    ( main,
      partOne,
      partTwo,
    )
  where

  import AdventOfCode.Util (parseInput)
  import Control.Applicative ((<|>))
  import Data.Function (on)
  import Data.List (sortBy)
  import Text.Trifecta ((<?>), Parser, char, count, some)

  ⟨Define a Pixel data type 17a⟩

  ⟨Implement Show for Pixel 17b⟩

  ⟨Define a few convenient type aliases 17c⟩

  main :: IO ()
  main =
    do
      putStrLn ”[2019] Day 8: Space Image Format”
      putStr ”Part One: ”
      print =<< partOne
      putStrLn ”Part Two: ”
      putStrLn =<< partTwo

  ⟨Part One 13c⟩

  ⟨Part Two 14b⟩

  ⟨Parse an image 18a⟩

  ⟨Parse a pixel 18b⟩

  ⟨A transparent layer 19b⟩
```

Root chunk (not used in this document).

Chunks

To-Do