ERIC BAILEY

ADVENT OF CODE

Contents

Day 1: The Tyranny of the Rocket Equation

Copy description

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

GAP Solution

fuel := mass\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 2: 1202 Program Alarm

Copy description

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

Haskell Solution

⟨*Day02.hs* 7a⟩≡

```
module Data.AOC19.Day02 where

import           Control.Arrow       (first, (»>))
import           Data.List           (find)
import           Data.Vector         (Vector, fromList, modify, toList, (!))
import qualified Data.Vector         as V
import           Data.Vector.Mutable (write)
import qualified Data.Vector.Mutable as MV
import           Text.Trifecta       (Parser, Result (..), comma, natural,
                                      parseFromFile, parseString, sepBy)
```

This definition is continued in chunks 7 and 8.
Root chunk (not used in this document).

⟨*Day02.hs* 7a⟩+≡

```
program :: Parser (Vector Int)
program = fromList . map fromInteger <$> (natural ‘sepBy‘ comma)
```

⟨*Day02.hs* 7a⟩+≡

```
partOne :: IO Int
partOne =
    do res ← parseFromFile program "../../../input/day02.txt"
       case res of
         Nothing    → error "No parse"
         Just state → pure (V.head (runProgram (restoreGravityAssist state)))
```

⟨*Day02.hs* 7a⟩+≡
```
partTwo :: IO Int
partTwo =
    do res ← parseFromFile program "../../../input/day02.txt"
       case res of
         Nothing    → error "No parse"
         Just state →
           do let n = V.length state - 1
              pure . maybe (error "Fail") (first (*100) ⋙ uncurry (+)) $
                find (go state) (concatMap (zip [0..n] . repeat) [0..n])
  where
    go state (noun, verb) =
        19690720 ≡ V.head (runProgram (restoreGravityAssist' noun verb state))
```

⟨*Day02.hs* 7a⟩+≡
```
restoreGravityAssist :: Vector Int → Vector Int
restoreGravityAssist = restoreGravityAssist' 12 2
```

⟨*Day02.hs* 7a⟩+≡
```
restoreGravityAssist' :: Int → Int → Vector Int → Vector Int
restoreGravityAssist' noun verb =
    modify (\v → write v 1 noun *> write v 2 verb)
```

⟨*Day02.hs* 7a⟩+≡
```
runProgram :: Vector Int → Vector Int
runProgram = go 0
  where
    go n state
      | state ! n ≡ 99 = state
      | otherwise      = go (n + 4) $ step (toList (V.slice n 4 state))
      where
        step [1, x, y, dst] = modify (runOp (+) x y dst) state
        step [2, x, y, dst] = modify (runOp (*) x y dst) state
        step _              = state

    runOp f x y dst v = write v dst =≪ f <$> MV.read v x <*> MV.read v y
```

⟨*Day02.hs* 7a⟩+≡
```
example1 :: Vector Int
example1 =
    case parseString program mempty "1,9,10,3,2,3,11,0,99,30,40,50" of
      Success prog   → prog
      Failure reason → error (show reason)
```

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* 9a⟩≡
```haskell
input :: [[Int]]
input = digits 10 <$> [236491 .. 713787]
```
This code is used in chunk 10b.

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* 9b⟩≡
```haskell
any ((≥ 2) . length) . group
```
This code is used in chunk 9c.

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

⟨*Part One* 9c⟩≡
```haskell
partOne :: Int
partOne = length $ filter isPossiblePassword input
  where
    isPossiblePassword :: [Int] → Bool
    isPossiblePassword = liftM2 (&&) isSorted hasDouble

    hasDouble :: Eq a ⇒ [a] → Bool
    hasDouble = ⟨has a double 9b⟩
```
This definition is continued in chunk 12.
This code is used in chunks 10b and 14.

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* 9d⟩≡
```haskell
any ((== 2) . length) . group
```
This code is used in chunk 10a.

⟨*Part Two* 10a⟩≡
```
partTwo :: Int
partTwo = length $ filter isPossiblePassword input
  where
   isPossiblePassword :: [Int] -> Bool
   isPossiblePassword = liftM2 (&&) isSorted hasDouble

   hasDouble :: Eq a => [a] -> Bool
   hasDouble = ⟨has a strict double 9d⟩
```
This definition is continued in chunk 13a.
This code is used in chunks 10b and 14.

Full Solution

⟨*Day04.hs* 10b⟩≡
```
module Data.AOC19.Day04 where

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


⟨Input 9a⟩


⟨Part One 9c⟩


⟨Part Two 10a⟩
```
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* 11a⟩≡
```
data Pixel
    = Black
    | White
    | Transparent
  deriving (Enum, Eq)
```
This code is used in chunk 14.

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

⟨*Implement* ***Show*** *for* `Pixel` 11b⟩≡
```
instance Show Pixel where
    show Black       = " "
    show White       = "#"
    show Transparent = "."
```
This code is used in chunk 14.

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* 11c⟩≡
```
type Image = [Layer]
type Layer = [Row]
type Row   = [Pixel]
```
This code is used in chunk 14.

Parsers

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

⟨*Parse an image* 12a⟩≡
```
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 14.

Parse an encoded black, white, or transparent pixel.

⟨*Parse a pixel* 12b⟩≡
```
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 14.

Part One

⟨*Part One* 9c⟩+≡
```
partOne :: FilePath → IO ()
partOne fname =
    do ⟨Parse a 25 × 6 image from the input 12d⟩
```
This code is used in chunks 10b and 14.

Better/safer binding

⟨*Parse a* 25 × 6 `image` *from the input* 12d⟩≡
```
Just layers ← parseFromFile (image 25 6) fname
```
This code is used in chunk 12c.

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

sp?

⟨*Part One* 9c⟩+≡
```
       let layer = head $ sortBy (compare `on` numberOf Black) layers
```
This code is used in chunks 10b and 14.

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

⟨*Part One* 9c⟩+≡
```
       let ones = numberOf White layer
       let twos = numberOf Transparent layer
       print $ ones * twos
```
This code is used in chunks 10b and 14.

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* 9c⟩+≡
```
    where
      numberOf :: Eq a ⇒ a → [[a]] → Int
      numberOf x = sum . fmap (length . filter (== x))
```
This code is used in chunks 10b and 14.

Part Two

⟨*Part Two* 10a⟩+≡
```
partTwo :: FilePath → IO ()
partTwo fname =
    do Just layers ← parseFromFile (image 25 6) fname
       putStrLn $
         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 10b and 14.

Miscellaneous

⟨*A transparent layer* 13b⟩≡
```
transparentLayer :: Int → Int → Layer
transparentLayer width height = replicate height (replicate width Transparent)
```
This code is used in chunk 14.

Pull this out into a utility module

⟨*Handle a single argument as file path to input* 13c⟩≡
```
getInputFilename :: IO FilePath
getInputFilename =
    do args ← getArgs
       case args of
         [fname] → pure fname
         []      → error "Must specify input filename"
         _       → error "Too many args"
```
This code is used in chunk 14.

Full solution

⟨*Day08.hs* 14⟩≡

```
- ——————————————————- [ Day08.hs ]
- TODO: Module doc
- ——————————————————— [ EOH ]

module Data.AOC19.Day08
  ( main
  , partOne, partTwo
  ) where


import           Control.Applicative ((<|>))
import           Data.Function       (on)
import           Data.List           (sortBy)
import           System.Environment  (getArgs)
import           Text.Trifecta       (Parser, char, count, parseFromFile, some,
                                      (<?>))


- ——————————————————— [ Types ]

⟨Define a Pixel data type 11a⟩


⟨Implement Show for Pixel 11b⟩


⟨Define a few convenient type aliases 11c⟩


- ——————————————————— [ Main ]

main :: IO ()
main =
    do putStr "Part One: "
       partOne =<< getInputFilename
       putStrLn "Part Two: "
       partTwo =<< getInputFilename


- ——————————————————- [ Part One ]

⟨Part One 9c⟩


- ——————————————————- [ Part Two ]

⟨Part Two 10a⟩


- ——————————————————- [ Parsers ]

⟨Parse an image 12a⟩
```

⟨*Parse a pixel* 12b⟩

```
-- ----------------------------------------------------- [ Helpers ]
```

⟨*A transparent layer* 13b⟩

⟨*Handle a single argument as file path to input* 13c⟩

```
-- --------------------------------------------------------- [ EOF ]
```

Root chunk (not used in this document).

Chunks

To-Do