ERIC BAILEY

ADVENT OF CODE 2018

Contents

Day 1: Chronal Calibration

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

5a ⟨*Day01.hs* 5a⟩≡
```
module Day01
  ( partOne
  , partTwo
  ) where
```

⟨*Import functions, operators, and types from other modules.* 7e⟩

⟨*Define data types to model the puzzle input.* 5b⟩

⟨*Define parsers for handling puzzle input.* 6b⟩

⟨*Solve parts one and two.* 6d⟩

Root chunk (not used in this document).

Data Types

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

5b ⟨*Define data types to model the puzzle input.* 5b⟩≡
```
newtype FrequencyChange = FrequencyChange
  { unFrequencyChange :: Sum Integer}
  deriving (Eq, Show)
```

Defines:
  `FrequencyChange`, used in chunks 5 and 6.
This definition is continued in chunks 5c and 6a.
This code is used in chunk 5a.

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

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

Describe these instances

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

5c ⟨*Define data types to model the puzzle input.* 5b⟩+≡
```
instance Hashable FrequencyChange where
    hashWithSalt salt = hashWithSalt salt . getSum . unFrequencyChange
```

Uses `FrequencyChange` 5b.

6a ⟨*Define data types to model the puzzle input.* 5b⟩+≡

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

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

Uses FrequencyChange 5b.

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}]
```

6b ⟨*Define parsers for handling puzzle input.* 6b⟩≡

```
frequencyChanges :: Parser [FrequencyChange]
frequencyChanges = many (FrequencyChange . Sum <$> integer)
```

Defines:
frequencyChanges, used in chunk 6c.
Uses FrequencyChange 5b.
This code is used in chunk 5a.

In practice, we'll use **ByteString**s and the helper function `maybeParseByteString :: Parser a -> ByteString -> Maybe a`, to try to parse the puzzle input.

6c ⟨*Try to parse the input* 6c⟩≡

```
maybeParseByteString frequencyChanges
```

Uses frequencyChanges 6b and maybeParseByteString 14e.
This code is used in chunks 6d and 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.

6d ⟨*Solve parts one and two.* 6d⟩≡

```
partOne :: ByteString -> Maybe Integer
partOne = fmap (getSum . unFrequencyChange . mconcat) .
          ⟨Try to parse the input 6c⟩
```

This definition is continued in chunk 7a.
This code is used in chunk 5a.

Part Two

7a ⟨*Solve parts one and two.* 6d⟩+≡
```
partTwo :: ByteString → Maybe Integer
partTwo =
    ⟨Try to parse the input 6c⟩ >=>
    ⟨Compute the list of frequencies reached 7b⟩ >>>
    ⟨Find the first duplicate 7c⟩ >>>
    ⟨Unbox the result 7d⟩
```

7b ⟨*Compute the list of frequencies reached* 7b⟩≡
```
scan . cycle
```
Uses scan 16b.
This code is used in chunk 7a.

7c ⟨*Find the first duplicate* 7c⟩≡
```
findFirstDup
```
Uses findFirstDup 15d.
This code is used in chunk 7a.

7d ⟨*Unbox the result* 7d⟩≡
```
fmap (getSum . unFrequencyChange)
```
This code is used in chunk 7a.

Imports

7e ⟨*Import functions, operators, and types from other modules.* 7e⟩≡
```
import           Control.Category ((>>>))
```
This definition is continued in chunk 7.
This code is used in chunk 5a.

7f ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Control.Monad    ((>=>))
```

7g ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Data.ByteString  (ByteString)
```

7h ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Data.Hashable    (Hashable (..))
```

7i ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Data.Monoid      (Sum (..))
```

7j ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Text.Trifecta    (Parser, integer, many)
```

7k ⟨*Import functions, operators, and types from other modules.* 7e⟩+≡
```
import           Util             (findFirstDup, maybeParseByteString, scan)
```
Uses findFirstDup 15d, maybeParseByteString 14e, and scan 16b.

Day 2: Inventory Management System

9a ⟨*Day02.hs* 9a⟩≡

```
module Day02
  ( partOne
  , partTwo
  ) where

⟨Imports 11⟩

⟨Types and parsers 9b⟩

⟨Part One 10a⟩

⟨Part Two 10c⟩
```

Uses `partOne` 10b and `partTwo` 10d.
Root chunk (not used in this document).

Type aliases and parsers

9b ⟨*Types and parsers* 9b⟩≡

```
type BoxID = String

boxID :: Parser BoxID
boxID = some letter

boxIDs :: Parser [BoxID]
boxIDs = boxID `sepEndBy` newline

type Checksum = Integer
```

Defines:
`BoxID`, used in chunk 10.
`boxIDs`, used in chunk 10.
This code is used in chunk 9a.

Part One

10a ⟨*Part One* 10a⟩≡

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

Defines:
  checksum, used in chunk 10b.
Uses BoxID 9b.
This definition is continued in chunk 10b.
This code is used in chunk 9a.

10b ⟨*Part One* 10a⟩+≡

```
partOne :: ByteString -> Maybe Checksum
partOne = maybeParseByteString boxIDs >=> pure . checksum
```

Defines:
  partOne, used in chunk 9a.
Uses boxIDs 9b, checksum 10a, and maybeParseByteString 14e.

Part Two

10c ⟨*Part Two* 10c⟩≡

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

Defines:
  correctBoxIDs, used in chunk 10d.
Uses BoxID 9b and hammingSimilar 15c.
This definition is continued in chunk 10d.
This code is used in chunk 9a.

10d ⟨*Part Two* 10c⟩+≡

```
partTwo :: ByteString -> Maybe String
partTwo = maybeParseByteString boxIDs >=>
          correctBoxIDs >=>
          uncurry commonElems
```

Defines:
  partTwo, used in chunk 9a.
Uses boxIDs 9b, commonElems 15a, correctBoxIDs 10c, and maybeParseByteString 14e.

Imports

11 ⟨*Imports* 11⟩≡

```
import           Control.Arrow   ((&&&), (***), (>>>))
import           Control.Monad   ((>=>))
import           Data.ByteString (ByteString)
import           Data.List       (find, tails)
import           Data.Maybe      (listToMaybe, mapMaybe)
import           Text.Trifecta   (Parser, letter, newline, sepEndBy, some)
import           Util            (commonElems, frequencies, hammingSimilar,
                                  maybeParseByteString)
```

Uses commonElems 15a, hammingSimilar 15c, and maybeParseByteString 14e.
This code is used in chunk 9a.

Common Utilities

Language extensions

LambdaCase is one of my favorite extensions.

Add link re: LambdaCase

13a ⟨*Util.hs* 13a⟩≡

```
{-# LANGUAGE LambdaCase #-}
```

This definition is continued in chunk 13b.
Root chunk (not used in this document).

Module outline

Consider some prose here

13b ⟨*Util.hs* 13a⟩+≡

```
module Util
  ( Frequencies, frequencies
  , maybeParseByteString
  , commonElems
  , findFirstDup
  , hammingDistance, hammingSimilar
  , scan
  ) where
```

⟨*Import functions, operators, and types from other modules.* 16c⟩

⟨*Computing frequencies* 14a⟩

⟨*Parsing puzzle input* 14e⟩

⟨*Manipulating lists* 15a⟩

Uses Frequencies 14a, commonElems 15a, findFirstDup 15d, hammingDistance 15b, hammingSimilar 15c, maybeParseByteString 14e, and scan 16b.

Computing frequencies of elements in a list

Describe the Frequencies type alias

14a ⟨*Computing frequencies* 14a⟩≡
```
type Frequencies a = HM.HashMap a Integer
```
Defines:
Frequencies, used in chunks 13 and 14.
This definition is continued in chunk 14.
This code is used in chunk 13b.

Define a function `frequencies` to compute the **`Frequencies`** of elements in a given list.

14b ⟨*Computing frequencies* 14a⟩+≡
```
frequencies :: (Eq a, Hashable a) ⇒ [a] → Frequencies a
```
Uses Frequencies 14a.

Starting with the empty map, perform a right-associative fold of the list, using the binary operator `go`.

14c ⟨*Computing frequencies* 14a⟩+≡
```
frequencies = foldr go HM.empty
  where
    go :: (Eq a, Hashable a) ⇒ a → Frequencies a → Frequencies a
```
Uses Frequencies 14a.

Given a key `k` and map of known frequencies, increment the associated frequency count by 1, or set it to 1 if no such mapping exists.

14d ⟨*Computing frequencies* 14a⟩+≡
```
    go k = HM.insertWith (+) k 1
```

Parsing puzzle input

Describe the general parsing strategy

14e ⟨*Parsing puzzle input* 14e⟩≡
```
maybeParseByteString :: Parser a → ByteString → Maybe a
maybeParseByteString p = parseByteString p mempty >>> \case
  Failure _   → Nothing
  Success res → Just res
```
Defines:
maybeParseByteString, used in chunks 6c, 7k, 10, 11, and 13b.
This code is used in chunk 13b.

Manipulating lists

Describe commonElems

15a ⟨*Manipulating lists* 15a⟩≡
```
commonElems :: (Eq a) ⇒ [a] → [a] → Maybe [a]
commonElems (x:xs) (y:ys) | x == y    = Just [x] <> recur
                          | otherwise = recur
  where recur = commonElems xs ys
commonElems _ _                       = Nothing
```
Defines:
commonElems, used in chunks 10d, 11, and 13b.
This definition is continued in chunks 15 and 16.
This code is used in chunk 13b.

Describe hammingDistance, incl. design choices

15b ⟨*Manipulating lists* 15a⟩+≡
```
hammingDistance :: Eq a ⇒ [a] → [a] → Maybe Integer
hammingDistance (x:xs) (y:ys) | x /= y    = (+1) <$> recur
                              | otherwise = recur
  where recur = hammingDistance xs ys
hammingDistance [] []                     = Just 0
hammingDistance _ _                       = Nothing
```
Defines:
hammingDistance, used in chunks 13b and 15c.

Describe hammingSimilar

15c ⟨*Manipulating lists* 15a⟩+≡
```
hammingSimilar :: Eq a ⇒ Integer → [a] → [a] → Bool
hammingSimilar n xs = maybe False (≤ n) . hammingDistance xs
```
Defines:
hammingSimilar, used in chunks 10c, 11, and 13b.
Uses hammingDistance 15b.

Define a function to find the first duplicated element of a list, if such an element exists.

15d ⟨*Manipulating lists* 15a⟩+≡
```
findFirstDup :: (Eq a, Hashable a) ⇒ [a] → Maybe a
```
Defines:
findFirstDup, used in chunks 7, 13b, and 15e.

Recurse over the list until either the end or a duplicate is found.

15e ⟨*Manipulating lists* 15a⟩+≡
```
findFirstDup = go HS.empty
  where
```
Uses findFirstDup 15d.

If the list is empty, we've found **Nothing**.

15f ⟨*Manipulating lists* 15a⟩+≡
```
    go _ []       = Nothing
```

If we've seen x before, we've **Just** found a duplicate.

15g ⟨*Manipulating lists* 15a⟩+≡
```
    go seen (x:xs) | x `HS.member` seen = Just x
```

Otherwise, insert `x` into the set of elements we've `seen` and carry on searching the rest of the list.

16a ⟨*Manipulating lists* 15a⟩+≡

```
                | otherwise          = go (HS.insert x seen) xs
```

Compute a list of successive reduced values, using the monodial operation, from the left, starting with the monoidal idendity.

$$(b_k)_{k=0}^{|a|},\ b_0 = e \text{ and } b_{k+1} = b_k a_k$$

Improve this. Consider group theory notation.

16b ⟨*Manipulating lists* 15a⟩+≡

```
scan :: Monoid m => [m] -> [m]
scan = scanl mappend mempty
```

Defines:
scan, used in chunks 7 and 13b.

Imports

16c ⟨*Import functions, operators, and types from other modules.* 16c⟩≡

```
import           Control.Category    ((>>>))
import           Data.ByteString     (ByteString)
import           Data.Hashable       (Hashable (..))
import qualified Data.HashMap.Strict as HM
import qualified Data.HashSet        as HS
import           Text.Trifecta       (Parser, Result (..), parseByteString)
```

This code is used in chunk 13b.

Chunks

Index

To-Do