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` 10e.
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` 12b.
This code is used in chunk 7a.

7c ⟨*Find the first duplicate* 7c⟩≡
```
findFirstDup
```
Uses `findFirstDup` 11d.
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` 11d, `maybeParseByteString` 10e, and `scan` 12b.

Common Utilities

Language extensions

LambdaCase is one of my favorite extensions.

Add link re: LambdaCase

9a ⟨*Util.hs* 9a⟩≡

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

This definition is continued in chunk 9b.
Root chunk (not used in this document).

Module outline

Consider some prose here

9b ⟨*Util.hs* 9a⟩+≡

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

⟨Import functions, operators, and types from other modules. 12c⟩

⟨Computing frequencies 10a⟩

⟨Parsing puzzle input 10e⟩

⟨Manipulating lists 11a⟩
```

Uses `Frequencies` 10a, `commonElems` 11a, `findFirstDup` 11d, `hammingDistance` 11b, `hammingSimilar` 11c, `maybeParseByteString` 10e, and `scan` 12b.

Computing frequencies of elements in a list

Describe the Frequencies type alias

10a ⟨*Computing frequencies* 10a⟩≡
```
type Frequencies a = HM.HashMap a Integer
```
Defines:
Frequencies, used in chunks 9 and 10.
This definition is continued in chunk 10.
This code is used in chunk 9b.

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

10b ⟨*Computing frequencies* 10a⟩+≡
```
frequencies :: (Eq a, Hashable a) ⇒ [a] → Frequencies a
```
Uses Frequencies 10a.

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

10c ⟨*Computing frequencies* 10a⟩+≡
```
frequencies = foldr go HM.empty
  where
    go :: (Eq a, Hashable a) ⇒ a → Frequencies a → Frequencies a
```
Uses Frequencies 10a.

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.

10d ⟨*Computing frequencies* 10a⟩+≡
```
    go k = HM.insertWith (+) k 1
```

Parsing puzzle input

Describe the general parsing strategy

10e ⟨*Parsing puzzle input* 10e⟩≡
```
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, and 9b.
This code is used in chunk 9b.

Manipulating lists

Describe commonElems

11a ⟨*Manipulating lists* 11a⟩≡
```
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 chunk 9b.
This definition is continued in chunks 11 and 12.
This code is used in chunk 9b.

Describe hammingDistance, incl. design choices

11b ⟨*Manipulating lists* 11a⟩+≡
```
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 9b and 11c.

Describe hammingSimilar

11c ⟨*Manipulating lists* 11a⟩+≡
```
hammingSimilar :: Eq a => Integer -> [a] -> [a] -> Bool
hammingSimilar n xs = maybe False (<= n) . hammingDistance xs
```
Defines:
hammingSimilar, used in chunk 9b.
Uses hammingDistance 11b.

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

11d ⟨*Manipulating lists* 11a⟩+≡
```
findFirstDup :: (Eq a, Hashable a) => [a] -> Maybe a
```
Defines:
findFirstDup, used in chunks 7, 9b, and 11e.

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

11e ⟨*Manipulating lists* 11a⟩+≡
```
findFirstDup = go HS.empty
  where
```
Uses findFirstDup 11d.

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

11f ⟨*Manipulating lists* 11a⟩+≡
```
    go _ []         = Nothing
```

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

11g ⟨*Manipulating lists* 11a⟩+≡
```
    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.

12a ⟨*Manipulating lists* 11a⟩+≡
```
                    | 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.

12b ⟨*Manipulating lists* 11a⟩+≡
```
scan :: Monoid m => [m] -> [m]
scan = scanl mappend mempty
```
Defines:
`scan`, used in chunks 7 and 9b.

Imports

12c ⟨*Import functions, operators, and types from other modules.* 12c⟩≡
```
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 9b.

Chunks

Index

To-Do