ERIC BAILEY

ADVENT OF CODE

Contents

Day 1: The Tyranny of the Rocket Equation

Copy description

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 2: 1202 Program Alarm

Haskell Solution

⟨*Day02.hs* 7a⟩≡

```haskell
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⟩+≡

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

⟨*Day02.hs* 7a⟩+≡

```haskell
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)
```

Chunks

⟨Day01.g 5a⟩   ⟨Day02.hs 7a⟩

To-Do