1. Day one

Here is the overview of our program.

```
module AdventOfCode.Y2015.Day01 where
  ⟨ Imported modules 2 ⟩
  ⟨ Type declarations 3 ⟩
  ⟨ Functions 5 ⟩

  solve :: FilePath → IO ()
  solve ≜ do
    text ← readFileUtf8 path
    let input = Text.unpack text
    print (solve_a input, solve_b input)
```

2. ⟨ Imported modules 2 ⟩ ≡

```
import "text" Data.Text qualified as Text
import AdventOfCode.Lib ( readFileUtf8 )
```

See also chunks 4 and 6.

This code is used in chunk 1.

3. We are told the following concerning the problem facing the protagonist:

- "He starts on the ground floor (floor 0) and then follows the instructions one character at a time."
- "An opening parenthesis, `(`, means he should go up one floor, and a closing parenthesis, `)` means he should go down one floor."
- "The apartment building is very tall, and the basement is very deep; he will never find the top or bottom floors."

We are asked to answer two questions:

1. "To what floor to the instructions take him?"
2. "What is the position of the character that causes [him] to first enter the basement?"

⟨ Type declarations 3 ⟩ ≡

```
data Variables = Variables
  { currentFloor, currentPos :: !Int
  } deriving stock (Generic, Show)
```

This code is used in chunk 1.

4. ⟨ Imported modules 2 ⟩ ≡

```
import "base" GHC.Generics ( Generic )
```

This code is used in chunk 1.

5. To discover to which floor the instructions lead us, we traverse the list of characters adding 1 to *currentFloor* whenever we encounter a `(` and subtracting 1 whenever we encounter a `)`.

```
⟨ Functions 5 ⟩ ≡
  solve_a :: [ Char ] → Int
  solve_a ss ≜ let
    ini = Variables { currentFloor = 0, currentPos = 0 }
    fin = runIdentity $ execStateT (traverse f ss) ini
    in fin ^. #currentFloor
    where
      f :: Char → StateT Variables Identity ()
      f = λcase
        '(' → #currentFloor %= (λx → x + 1)
        ')' → #currentFloor %= (λx → x − 1)
```

See also chunk 7.

This code is used in chunk 1.

6.

```
⟨ Imported modules 2 ⟩ ≡
  import "mtl" Control.Monad.Identity ( Identity, runIdentity )
  import "mtl" Control.Monad.State.Strict
    ( StateT, execStateT, lift )
  import "lens" Control.Lens ( (^.), (%=) )
  import "generic-lens" Data.Generics.Labels
```

This code is used in chunk 1.

7. In order to answer the second question, we use *currentPos* to keep track of our position in the list. Whether we encounter a `(` or a `)`, we increase the value of *currentPos* by 1. We then check *currentFloor* to see what floor we are on. The first time we are in the basement (*currentFloor* $= -1$) we short-circuit the traversal of the list by wrapping the value of *currentPos* with *Left*.

```
⟨ Functions 5 ⟩ ≡
  solve_b :: [ Char ] → Int
  solve_b ss = let
    ini = Variables { currentFloor = 0 , currentPos = 0 }
    Left n = execStateT (traverse f ss) ini
    in n
    where
      f :: Char → StateT Variables (Either Int) ()
      f c = do
        #currentPos %= (+1)
        case c of
          '(' → #currentFloor %= (λx → x + 1)
          ')' → #currentFloor %= (λx → x − 1)
        ⟨ Check if we have found the basement 8 ⟩
```

This code is used in chunk 1.

8. ⟨ Check if we have found the basement 8 ⟩ ≡

```
  currentFloor ← use #currentFloor
  when (currentFloor == negate 1) $ do
    currentPos ← use #currentPos
    lift $ Left currentPos
```

This code is used in chunk 7.

1. Day Two

First, the birds' eye view of our program.

```
module AdventOfCode.Y2015.Day02 where

⟨ Imported modules 2 ⟩
⟨ Type definitions 7 ⟩
⟨ Functions 4 ⟩

solve :: FilePath → IO ()
solve ≜ do
  input ← readFileUtf8 path
  case parse $ Text.unpack input of
    Left err → print err
    Right dims → print (solve_a dims, solve_b dims)
```

2. ⟨ Imported modules 2 ⟩≡

```
import "text" Data.Text qualified as Text
import AdventOfCode.Lib ( readFileUtf8 )
```

See also chunk 8.

This code is used in chunk 1.

2. The elves wish to order paper to wrap their presents. We are given the following:

- Every present is a box (a perfect rectangular prism) and thus is characterised by three dimensions: length (l), width (w) and height (h).
- The surface area of a present is given by

$$2 \times l \times w \ + \ 2 \times w \times h \ + \ 2 \times h \times l$$

- A little extra paper is required for each present viz. the area of the smallest side.

How many total square feet of wrapping paper should they order?

⟨ Functions 2 ⟩ ≡

```
wrapping :: (Int, Int, Int) → Int
wrapping lwh = surfaceArea lwh + (area · shortest_dims) lwh

surface_area :: (Int, Int, Int) → Int
surface_area (l , w, h) = 2 × l × w + 2 × w × h + 2 × h × l

shortest_dims :: (Int , Int , Int) → (Int , Int)
```

```
shortest_dims (a , b , c) = let
  [ x , y , z ] = List.sort [ a , b , c ]
  in (x , y)

area :: (Int , Int) → Int
area = uncurry (×)
```

See also chunks 3, 4 and 5.

This code is used in chunk 1.

3. Given a list of triples representing the length, width and height of a present, we compute the total wrapping paper needed by traversing the list, accumulating the wrapping paper computed for each present.

```
⟨ Functions 2 ⟩ ≡
  solve_a :: [ (Int , Int , Int) ] → Const (Sum Int) [ a ]
  solve_a ≜ traverse f
    where
      f :: (Int , Int , Int) → Const (Sum Int) a
      f ≜ Const · Sum · wrapping
```

This code is used in chunk 1.

4. For the second part of the problem, we are told that each wrapped present must also have a ribbon.

- "The ribbon required to wrap a present is the shortest distance around its size, or the smallest perimeter of any one face."
- "Each present requires a bow made out of ribbon... the feet of ribbon required is equal to the volume of the present."

How many feet of ribbon is needed in total?

```
⟨ Functions 2 ⟩ ≡
  ribbon :: (Int , Int , Int) → Int
  ribbon lwh ≜ volume lwh + (perimeter · shortest_dims) lwh

  volume :: (Int , Int , Int) → Int
  volume (l , w , h) ≜ l × w × h

  perimeter :: (Int , Int) → Int
  perimeter (a , b) ≜ 2 × (a + b)

  solve_b :: [ (Int , Int , Int) ] → Const (Sum Int) [ a ]
```

```
solve_b ≜ traverse f
  where
    f :: (Int , Int , Int) → Const (Sum Int) a
    f ≜ Const · Sum · ribbon
```

This code is used in chunk 1.

5. Our algorithms work by traversing a list of triples. However the input to our program is a text file where each line in the file is comprised of three numbers separated by the ẋ character e.g. `4x23x21` We use the following parsing functions to obtain a list of triples from the input text.

⟨ Functions 2 ⟩ ≡

```
parse :: String
      → Either (ParseErrorBundle String Error) [ (Int , Int , Int) ]
parse ≜ runParser (dimensions `sepBy` newline)

dimensions :: Parser (Int , Int , Int)
dimensions ≜ do
  l ← decimal
  char 'x'
  w ← decimal
  char 'x'
  h ← decimal
  pure (l, w, h)
```

This code is used in chunk 1.

7. ⟨ Type definitions 7 ⟩ ≡

```
type Parser a  ≜  Parsec Error [ Chars ] a
type Error     ≜  Text
```

This code is used in chunk 1.

8. The types and functions for parsing are found in the `megaparsec` package.

⟨ Imported modules 2 ⟩ ≡

```
import "megaparsec" Text.Megaparsec (Parsec, ParseErrorBundle,
    sepBy, runParser)
import "megaparsec" Text.Megaparsec.Char (char, newline)
import "megaparsec" Text.Megaparsec (decimal)
```

This code is used in chunk 1.