Part I

The Mind-Body Problem
The Mind-Body Problem
Part II

Commands
Print a character

putChar :: Char -> IO ()

For instance,

        putChar '!

denotes the command that, *if it is ever performed*, will print an exclamation mark.
Combine two commands

\[(\gg\gg) :: \text{IO} () \to \text{IO} () \to \text{IO} ()\]

For instance,

\[\text{putChar } '?' \gg \text{putChar } '!'\]

denotes the command that, \textit{if it is ever performed}, prints a question mark followed by an exclamation mark.
Do nothing

\[
done :: IO ()
\]

The term \( \text{done} \) denotes the command that, \textit{if it is ever performed}, won’t do anything.

That is not quite the same as doing nothing! Compare thinking about doing nothing to actually doing nothing: they are distinct enterprises.
Print a string

\[
\begin{align*}
\text{putStr} & \:: \text{String} \rightarrow \text{IO ()} \\
\text{putStr} \ [x] & = \text{done} \\
\text{putStr} \ (x:xs) & = \text{putChar} \ x \ >> \ \text{putStr} \ xs
\end{align*}
\]

So

\[
\text{putStr} \ "?!"
\]

is equivalent to

\[
\text{putChar} \ '?' \ >> \ (\text{putChar} \ '!') \ >> \ \text{done}
\]

and both denote a command that, \textit{if it is ever performed}, prints a question mark followed by an exclamation mark.
Higher-order functions

More compactly, we can define `putStr` as follows.

\[
\text{putStr} :: \text{String} \rightarrow \text{IO} ()
\]

\[
\text{putStr} = \text{foldr} (\gg\gg) \text{done} \cdot \text{map} \text{putChar}
\]

The operator \( \gg \gg \) has identity \text{done} and is associative.

\[
\begin{align*}
m \gg \gg \text{done} &= m \\
\text{done} \gg \gg m &= m \\
(m \gg \gg n) \gg \gg o &= m \gg \gg (n \gg \gg o)
\end{align*}
\]
Main

By now you may be desperate to know *how is a command ever performed?* Here is the file `Confused.hs`:

```haskell
module Confused where
main :: IO ()
main = putStrLn "?!"
```

Running this program prints an indicator of perplexity:

```
$ runghc Confused.hs
?!$
```

Thus `main` is the link from Haskell’s mind to Haskell’s body — the analogue of Descartes’s pineal gland.
Print a string followed by a newline

```haskell
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

Here is the file `ConfusedLn.hs`:

```haskell
module ConfusedLn where
main :: IO ()
main = putStrLn "?!"
```

This prints its result more neatly:

```
$ runghc ConfusedLn.hs
?!
$
```
Part III

Equational reasoning
Equational reasoning lost

In languages with side effects, this program prints “haha” as a side effect.

```plaintext
print "ha"; print "ha"
```

But this program only prints “ha” as a side effect.

```plaintext
let x = print "ha" in x; x
```

This program again prints “haha” as a side effect.

```plaintext
let f () = print "ha" in f (); f ()
```
Equational reasoning regained

In Haskell, the term
\[(1+2) \ast (1+2)\]

and the term
\[
\textbf{let } x = 1+2 \textbf{ in } x \ast x
\]

are equivalent (and both evaluate to 9).

In Haskell, the term
\[
\text{putStr "ha" >> putStr "ha"}
\]

and the term
\[
\textbf{let } m = \text{putStr "ha" in } m \ast m
\]

are also entirely equivalent (and both print "haha").
Part IV

Commands with values
Read a character

Previously, we wrote \texttt{IO ( )} for the type of commands that yield no value.

Here, ( ) is the trivial type that contains just one value, which is also written ( ).

We write \texttt{IO Char} for the type of commands that yield a value of type \texttt{Char}.

Here is a command to read a character.

\begin{verbatim}
getchar :: IO Char
\end{verbatim}

Performing the command \texttt{getChar} when the input contains "abc" yields the value 'a' and remaining input "bc".
Do nothing and return a value

More generally, we write \( \text{IO } a \) for commands that return a value of type \( a \).

The command

\[
\text{return} :: a \rightarrow \text{IO } a
\]

is similar to \text{done}, in that it does nothing, but it also returns the given value.

Performing the command

\[
\text{return } [] :: \text{IO } \text{String}
\]

when the input contains "bc" yields the value \([\ ]\) and an unchanged input "bc".
Combining commands with values

We combine command with an operator written $\gg=$ and pronounced “bind”.

\[
(\gg=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\]

For example, performing the command

\[
\text{getChar} \gg= \ x \rightarrow \text{putChar } (\text{toUpperCase } x)
\]

when the input is "abc" produces the output "A", and the remaining input is "bc".
The “bind” operator in detail

\[(\gg\gg) \:: IO\ a \to (a \to IO\ b) \to IO\ b\]

If

\[m :: IO\ a\]

is a command yielding a value of type \(a\), and

\[k :: a \to IO\ b\]

is a function from a value of type \(a\) to a command yielding a value of type \(b\), then

\[m \gg\gg k :: IO\ b\]

is the command that, \textit{if it is ever performed}, behaves as follows:

1. first perform command \(m\) yielding a value \(x\) of type \(a\);
2. then perform command \(k \ x\) yielding a value \(y\) of type \(b\);
3. then yield the final value \(y\).
Reading a line

Here is a program to read the input until a newline is encountered, and to return a list of the values read.

```haskell
getLine :: IO String
getLine = getChar >>= \x ->
    if x == '\n' then
      return []
    else
      getline >>= \xs ->
      return (x:xs)
```

For example, given the input "abc\ndef" This returns the string "abc" and the remaining input is "def".
Commands as a special case

The general operations on commands are:

\[
\begin{align*}
\text{return} & : a \rightarrow \text{IO } a \\
(\gg\gg=) & : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\end{align*}
\]

The command \texttt{done} is a special case of \texttt{return}, and the operator \texttt{>>} is a special case of \texttt{gggg=}.

\[
\begin{align*}
\text{done} & : \text{IO } () \\
\text{done} & = \text{return } () \\
(\gg) & : \text{IO } () \rightarrow \text{IO } () \rightarrow \text{IO } () \\
\text{m >> n} & = \text{m ggggg } () \rightarrow n
\end{align*}
\]
Echoing input to output

This program echoes its input to its output, putting everything in upper case, until an empty line is entered.

```haskell
echo :: IO ()
echo = getLine >>= \line ->
  if line == "" then
    return ()
  else
    putStrLn (map toUpper line) >>
    echo

main :: IO ()
main = echo
```

Testing it out

$ runghc Echo.hs
This is a test.
THIS IS A TEST.
It is only a test.
IT IS ONLY A TEST.
Were this a real emergency, you'd be dead now.
WERE THIS A REAL EMERGENCY, YOU'D BE DEAD NOW.

$
Part V

“Do” notation
Reading a line in “do” notation

```haskell
getLine :: IO String
getLine = getChar >>= \x ->
    if x == '\n' then
        return []
    else
        getLine >>= \xs ->
        return (x:xs)
```

is equivalent to

```haskell
getLine :: IO String
getLine = do {
    x <- getChar;
    if x == '\n' then
        return []
    else do {
        xs <- getLine;
        return (x:xs)
    }
}
```
Echoing in “do” notation

echo :: IO ()
echo = getLine >>= \line ->
  if line == "" then
    return ()
  else
    putStrLn (map toUpper line) >>
    echo

is equivalent to

echo :: IO ()
echo = do {
  line <- getLine;
  if line == "" then
    return ()
  else do {
    putStrLn (map toUpper line);
    echo
  }
}
“Do” notation in general

Each line \( x \leftarrow e; \ldots \) becomes \( e >>= \backslash x \rightarrow \ldots \)
Each line \( e; \ldots \) becomes \( e >> \ldots \)

For example,

\[
\begin{align*}
do \; \{ & x1 \leftarrow e1; \\
& x2 \leftarrow e2; \\
& e3; \\
& x4 \leftarrow e4; \\
& e5; \\
& e6 \; \} 
\end{align*}
\]
is equivalent to

\[
\begin{align*}
e1 >>= \backslash x1 \rightarrow & \\
e2 >>= \backslash x2 \rightarrow & \\
e3 & \\
e4 >>= \backslash x4 \rightarrow \\
e5 & \\
e6 
\end{align*}
\]
Part VI

Monads
Substitution

We write \( n[x := v] \) to stand for

\[
\text{term } n \text{ with variable } x \text{ replaced by value } v.
\]

For example, if \( n \) is \( x \ast x \) and \( x \) is \( x \) and \( v \) is \( 3 \),

\[
(x \ast x) [x := 3] = 3 \ast 3
\]

The beta law, which substitutes an actual parameter for a formal parameter, is

\[
(\lambda x \to n) v = n [x := v]
\]

For instance,

\[
(\lambda x \to x \ast x) 3 = (x \ast x) [x := 3] = 3 \ast 3
\]
Monoids

A *monoid* is a pair of an operator ($\oplus$) and a value $u$, where the operator has the value as identity and is associative.

\[
\begin{align*}
    u \oplus x &= x \\
x \oplus u &= x \\
(x \oplus y) \oplus z &= x \oplus (y \oplus z)
\end{align*}
\]

Examples of monoids:

- $(+)$ and 0
- $(\ast)$ and 1
- $(||)$ and False
- $(\&\&)$ and True
- $(++)$ and []
- $(>>)$ and done

The Chicken McNugget monoid
Monads

We know that $(\gg\gg)$ and `done` satisfy the laws of a *monoid*.

\[
\begin{align*}
\text{done} \gg\gg m & = m \\
m \gg\gg \text{done} & = m \\
(m \gg\gg n) \gg\gg o & = m \gg\gg (n \gg\gg o)
\end{align*}
\]

Similarly, $(\gg\gg=)$ and `return` satisfy the laws of a *monad*.

\[
\begin{align*}
\text{return } v \gg\gg= \lambda x \rightarrow m & = m \ [x := v] \\
m \gg\gg= \lambda x \rightarrow \text{return } x & = m \\
(m \gg\gg= \lambda x \rightarrow n) \gg\gg= \lambda y \rightarrow o & = m \gg\gg= \lambda x \rightarrow (n \gg\gg= \lambda y \rightarrow o)
\end{align*}
\]
Part VII

The monad of lists
The monad of lists

In the standard prelude:

```haskell
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

instance Monad [] where
  return :: a -> [a]
  return x = [ x ]

  (>>=) :: [a] -> (a -> [b]) -> [b]
  m >>= k = [ y | x <- m, y <- k x ]
```

Equivalently, we can define:

```haskell
[] >>= k = []
(x:xs) >>= k = (k x) ++ (xs >>= k)
```

or

```haskell
m >>= k = concat (map k m)
```
‘Do’ notation and the monad of lists

\[
\text{pairs} :: \text{Int} \rightarrow [(\text{Int}, \text{Int})]
\]
\[
\text{pairs} \ n = [ (i,j) \mid i \leftarrow [1..n], j \leftarrow [(i+1)..n] ]
\]
is equivalent to

\[
\text{pairs'} :: \text{Int} \rightarrow [(\text{Int}, \text{Int})]
\]
\[
\text{pairs'} \ n = \text{do} \ {i \leftarrow [1..n]; \ j \leftarrow [(i+1)..n]; \ \text{return} \ (i,j)}
\]

For example,

\[
$\text{ghci Pairs}$
\]
\[
\text{GHCi, version 8.0.2: http://www.haskell.org/ghc/} \quad \text{?? for help}
\]
\[
\text{Pairs} > \text{pairs} 4
\]
\[
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
\]
\[
\text{Pairs} > \text{pairs'} 4
\]
\[
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
\]
Monads with plus

In the standard prelude:

```haskell
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a

instance MonadPlus [] where

    mzero :: [a]
    mzero = []

    mplus :: [a] -> [a] -> [a]
    mplus = (++)

    guard :: MonadPlus m => Bool -> m ()
    guard False = mzero
    guard True  = return ()

    msum :: MonadPlus m => [m a] -> m a
    msum = foldr mplus mzero
```
Using guards

```
pairs'' :: Int -> [(Int, Int)]
pairs'' n = [ (i,j) | i <- [1..n], j <- [1..n], i < j ]
```
is equivalent to

```
pairs''' :: Int -> [(Int, Int)]
pairs''' n = do {
    i <- [1..n];
    j <- [1..n];
    guard (i < j);
    return (i,j)
}
```

For example,

```
$ ghci Pairs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Pairs> pairs'' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
Pairs> pairs''' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```
Part VIII

Parsers
Parser type

First attempt:

\[
\text{type } \text{Parser } a = \text{String } \to a
\]

Second attempt:

\[
\text{type } \text{Parser } a = \text{String } \to (a, \text{String})
\]

Third attempt:

\[
\text{type } \text{Parser } a = \text{String } \to [(a, \text{String})]
\]

A parser for things
is a function from strings
to lists of pairs
Of things and strings

—Graham Hutton
module Parser (Parser, apply, parse, char, spot, token, star, plus, parseInt) where

import Char
import Monad

-- The type of parsers
data Parser a = Parser (String -> [(a, String)])

-- Apply a parser
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) s = f s

-- Return parsed value, assuming at least one successful parse
parse :: Parser a -> String -> a
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
The Monad type class

```haskell
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

Parser is a Monad

-- Parsers form a monad

-- class Monad m where
--     return :: a -> m a
--     (>>=) :: m a -> (a -> m b) -> m b

instance Monad Parser where
    return x = Parser (\s -> [(x,s)])
    m >>= k = Parser (\s ->
        [ (y, u) |
            (x, t) <- apply m s,
            (y, u) <- apply (k x) t ])

Parser is a Monad with Plus

-- Some monads have additional structure

-- class MonadPlus m where
--     mzero :: m a
--     mplus :: m a -> m a -> m a

instance MonadPlus Parser where
    mzero = Parser (\s -> [])
    mplus m n = Parser (\s -> apply m s ++ apply n s)
Parsing characters

-- Parse a single character
char :: Parser Char
char = Parser f
  where
    f [] = []
    f (c:s) = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = Parser f
  where
    f [] = []
    f (c:s) | p c = [(c,s)]
             | otherwise = []

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)
Parsing characters

-- Parse a single character
char ::(Parser Char
char = Parser f

where
  f [] = []
  f (c:s) = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot ::(Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)
Parsing a given string

\[
\text{match} :: \text{String} \rightarrow \text{Parser String}
\]
\[
\text{match } [] \quad = \quad \text{return } []
\]
\[
\text{match } (x:xs) \quad = \quad \text{do}
\quad \quad y \leftarrow \text{token } x;
\quad \quad ys \leftarrow \text{match } xs;
\quad \quad \text{return } (y:ys)
\]
Parsing a sequence

-- match zero or more occurrences
star :: Parser a -> Parser [a]
star p = plus p `mplus` return []

-- match one or more occurrences
plus :: Parser a -> Parser [a]
plus p = do { x <- p;
              xs <- star p;
              return (x:xs) }
Parsing an integer

-- match a natural number
parseNat :: Parser Int
parseNat = do { s <- plus (spot isDigit);
              return (read s) }

-- match a negative number
parseNeg :: Parser Int
parseNeg = do { token '-';
               n <- parseNat
               return (-n) }

-- match an integer
parseInt :: Parser Int
parseInt = parseNat `mplus` parseNeg
Module Exp

module Exp where

import Monad
import Parser

data Exp = Lit Int
  | Exp :+: Exp
  | Exp : * : Exp
    deriving (Eq, Show)

evalExp :: Exp -> Int
evalExp (Lit n) = n
evalExp (e :+: f) = evalExp e + evalExp f
evalExp (e : * : f) = evalExp e * evalExp f
Parsing an expression

parseExp :: Parser Exp
parseExp = parseLit `mplus` parseAdd `mplus` parseMul

where
parseLit = do { n <- parseInt;
  return (Lit n) }

parseAdd = do { token '(';
  d <- parseExp;
  token '+';
  e <- parseExp;
  token ')';
  return (d :+: e) }

parseMul = do { token '(';
  d <- parseExp;
  token ' * '; 
  e <- parseExp;
  token ')';
  return (d :*: e) }
Testing the parser

```
$ ghci Exp.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Parser      ( Parser.hs, interpreted )
[2 of 2] Compiling Exp         ( Exp.hs, interpreted )
Ok, modules loaded: Parser, Exp.
> parse parseExp "(1+(2*3))"
Lit 1 :+: (Lit 2 :*: Lit 3)
> evalExp (parse parseExp "(1+(2*3))")
7
> parse parseExp "((1+2)*3)"
(Lit 1 :+: Lit 2) :*: Lit 3
> evalExp (parse parseExp "((1+2)*3)")
9
```