

Informatics 1
Functional Programming
Lectures 18–19

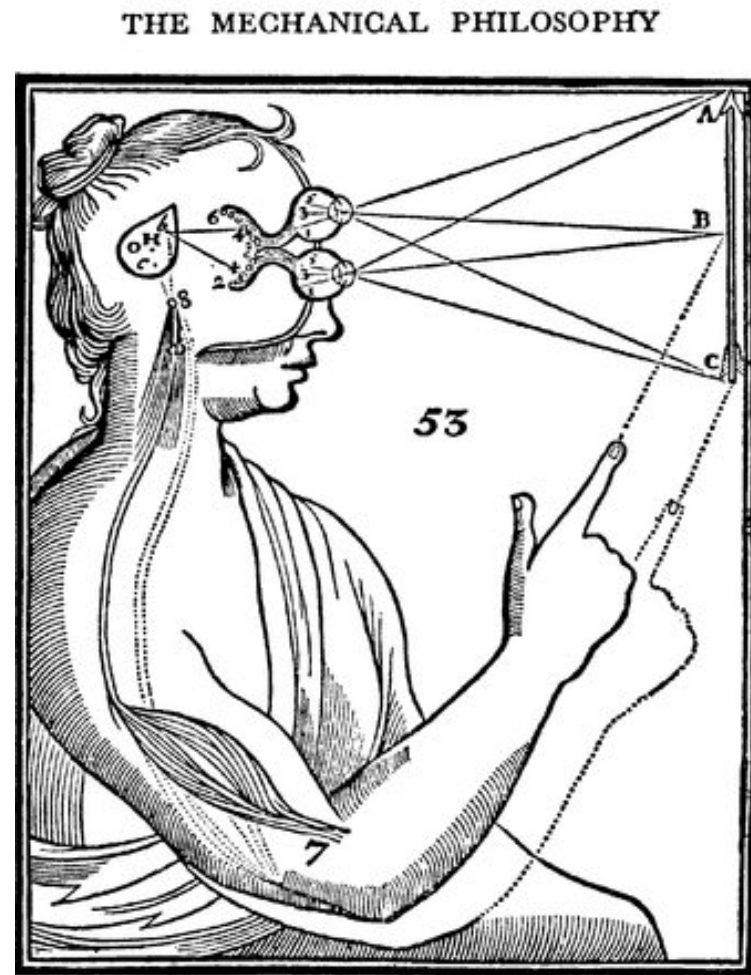
IO and Monads

Don Sannella
University of Edinburgh

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

`(>>) :: IO () -> IO () -> IO ()`

For instance,

`putChar '?' >> putChar '!'`

denotes the command that, *if it is ever performed*, prints a question mark followed by an exclamation mark.

Do nothing

```
done :: IO ()
```

The term

```
done
```

denotes the command that, *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

```
putStr :: String -> IO ()  
putStr []      = done  
putStr (x:xs)  = putChar x >> putStr xs
```

So

```
putStr "?!"
```

is equivalent to

```
putChar '?' >> (putChar '!' >> done)
```

and both denote a command that, *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.

```
putStr  :: String -> IO ()  
putStr = foldr (>>) done . map putChar
```

The operator `>>` has identity `done` and is associative.

```
m >> done      = m  
done >> m       = m  
(m >> n) >> o  = m >> (n >> o)
```

Main

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

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

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

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

Here is the file `ConfusedLn.hs`:

```
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.

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

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

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

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

```
let f () = print "ha" in f (); f ()
```

Equational reasoning regained

In Haskell, the term

$$(1+2) * (1+2)$$

and the term

```
let x = 1+2 in x * x
```

are equivalent (and both evaluate to 9).

In Haskell, the term

```
putStr "ha" >> putStr "ha"
```

and the term

```
let m = putStr "ha" in m >> m
```

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

Part IV

Commands with values

Read a character

Previously, we wrote `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 `IO Char` for the type of commands that yield a value of type `Char`.

Here is a command to read a character.

```
getChar :: IO Char
```

Performing the command `getChar` when the input contains `"abc"` yields the value `'a'` and remaining input `"bc"`.

Do nothing and return a value

More generally, we write `IO a` for commands that return a value of type `a`.

The command

```
return :: a -> IO a
```

is similar to `done`, in that it does nothing, but it also returns the given value.

Performing the command

```
return [] :: IO 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 `>>=` and pronounced “bind”.

$$(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$$

For example, performing the command

```
getChar >>= \x -> putChar (toUpper x)
```

when the input is `"abc"` produces the output `"A"`, and the remaining input is `"bc"`.

The “bind” operator in detail

$$(>>=) \quad :: \quad \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b$$

If

$$m \quad :: \quad \text{IO } a$$

is a command yielding a value of type a , and

$$k \quad :: \quad a \rightarrow \text{IO } b$$

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

$$m \gg= k \quad :: \quad \text{IO } b$$

is the command that, *if it is ever performed*, behaves as follows:

first perform command m yielding a value x of type a ;
then perform command $k \ x$ yielding a value y of type b ;
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.

```
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:

```
return    :: a -> IO a
(>>=)     :: IO a -> (a -> IO b) -> IO b
```

The command `done` is a special case of `return`,
and the operator `>>` is a special case of `>>=`.

```
done      :: IO ()
done      =  return ()
```

```
(>>)       :: IO () -> IO () -> IO ()
m >> n     =  m >>= \() -> n
```

Echoing input to output

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

```
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

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

is equivalent to

```
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 $x \leftarrow e; \dots$ becomes $e \gg= \backslash x \rightarrow \dots$

Each line $e; \dots$ becomes $e \gg \dots$

For example,

```
do { x1 ← e1;  
      x2 ← e2;  
      e3;  
      x4 ← e4;  
      e5;  
      e6 }
```

is equivalent to

```
e1 >>= \x1 ->  
e2 >>= \x2 ->  
e3 >>  
e4 >>= \x4 ->  
e5 >>  
e6
```

Part VI

Monads

Substitution

We write $n[x := v]$ to stand for

term n with variable x replaced by value v .

For example, if n is $x * x$ and x is x and v is 3,

$$(x * x)[x := 3] = 3 * 3$$

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

$$(\lambda x \rightarrow n) v = n[x := v]$$

For instance,

$$(\lambda x \rightarrow x * x) 3 = (x * x)[x := 3] = 3 * 3$$

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.

$$u \oplus x = x$$

$$x \oplus u = x$$

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

Examples of monoids:

(+) and 0

(*) and 1

(||) and False

(&&) and True

(++) and []

(>>) and done

Monads

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

$$\begin{aligned}\text{done} \gg m &= m \\ m \gg \text{done} &= m \\ (m \gg n) \gg o &= m \gg (n \gg o)\end{aligned}$$

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

$$\begin{aligned}\text{return } v \gg= \backslash x \rightarrow m &= m [x := v] \\ m \gg= \backslash x \rightarrow \text{return } x &= m \\ (m \gg= \backslash x \rightarrow n) \gg= \backslash y \rightarrow o &= m \gg= \backslash x \rightarrow (n \gg= \backslash y \rightarrow o)\end{aligned}$$

Part VII

The monad of lists

The monad of lists

In the standard prelude:

```
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:

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

or

```
m >>= k  =  concat (map k m)
```

‘Do’ notation and the monad of lists

```
pairs :: Int -> [(Int, Int)]
pairs n = [ (i,j) | i <- [1..n], j <- [(i+1)..n] ]
```

is equivalent to

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

Monads with plus

In the standard prelude:

```
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:

```
type Parser a = String -> a
```

Second attempt:

```
type Parser a = String -> (a, String)
```

Third attempt:

```
type Parser a = String -> [(a, String)]
```

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

—Graham Hutton

Module Parser

```
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

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


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

```
match :: String -> Parser String
match []      = return []
match (x:xs)  = do
    y <- token x;
    ys <- match xs;
    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
>
```