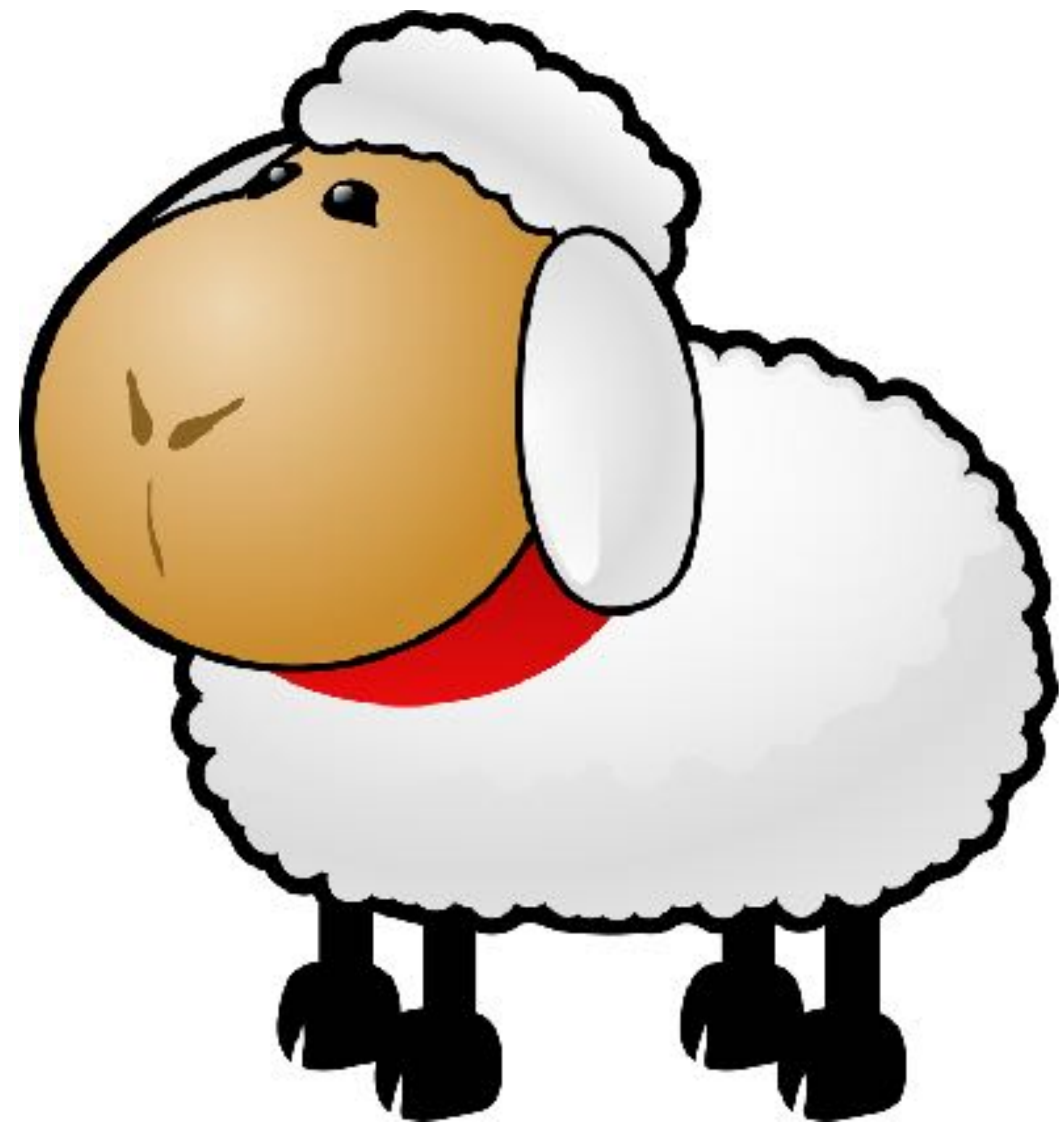
Design Patterns for Functional Programming

Christophe Scholliers

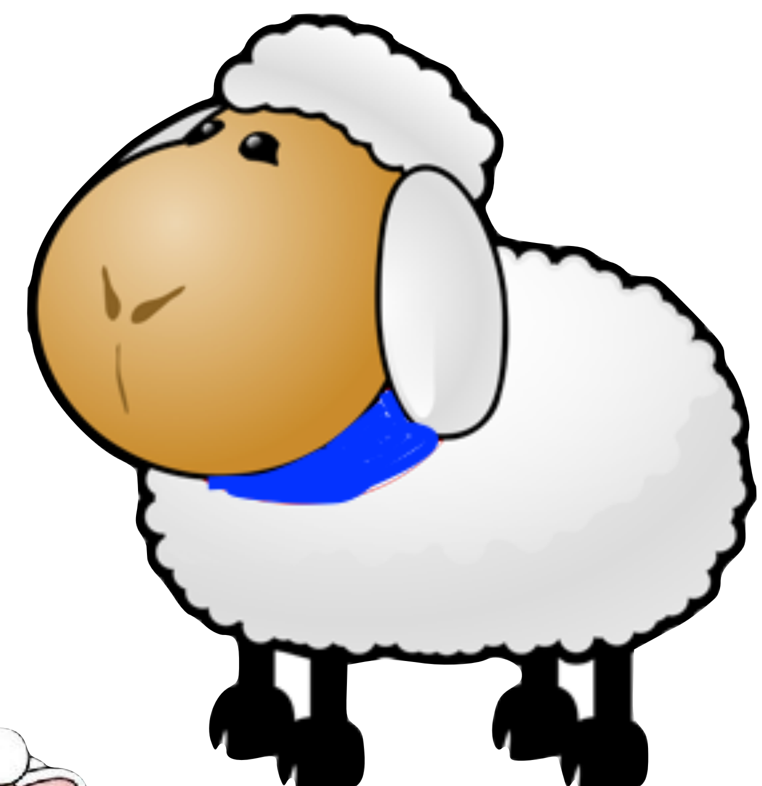
Slides adopted from Don Sannella University of Edinburgh
Informatics 1 Functional Programming Lecture 15
and
https://wiki.haskell.org/All_About_Monads

Cloning Sheep

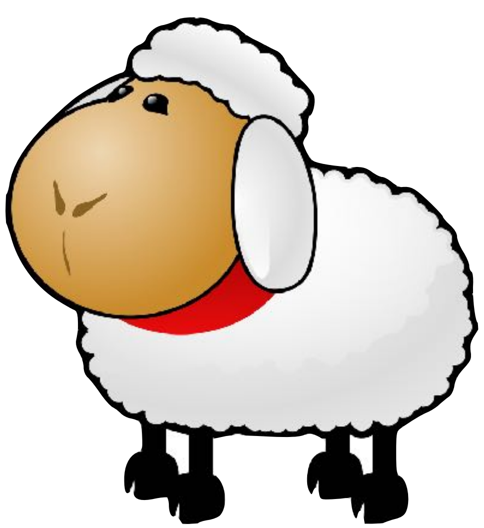


Sheep is an algebraic datatype

```haskell
-- everything you need to know about sheep
data Sheep = Sheep { name::String,
                     mother::Maybe Sheep,
                     father::Maybe Sheep }
```



`mother::Maybe Sheep`



Operations over Sheep

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
                          Nothing -> Nothing
                          Just m  -> father m


mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m  -> case (father m) of
                                              Nothing -> Nothing
                                              Just gf -> father gf
```

The Computer Science way

```haskell
-- comb is a combinator for sequencing operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b

comb Nothing  _ = Nothing
comb (Just x) f = f x
```

Let’s make a combinator

```haskell
-- comb is a combinator for sequencing operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x
```

Let’s use our operator

Wrap

```
-- now we can use `comb` to build complicated sequences
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = (Just s) `comb` mother `comb` father

fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = (Just s) `comb` father `comb` mother `comb` mother

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = (Just s) `comb` mother `comb` father `comb` father
```

Doing it with class

```haskell
class Chain m where
    comb' :: m a -> (a -> m b) -> m b
    wrap  :: a -> m a

instance Chain Maybe where
    comb' Nothing _    = Nothing
    comb' (Just x)  f  = f x
    wrap  a            = Just a
```

```haskell
mothersPaternalGrandfather' :: Sheep -> Maybe Sheep
mothersPaternalGrandfather' s = (wrap s) `comb'` mother `comb'` father `comb'` father
```

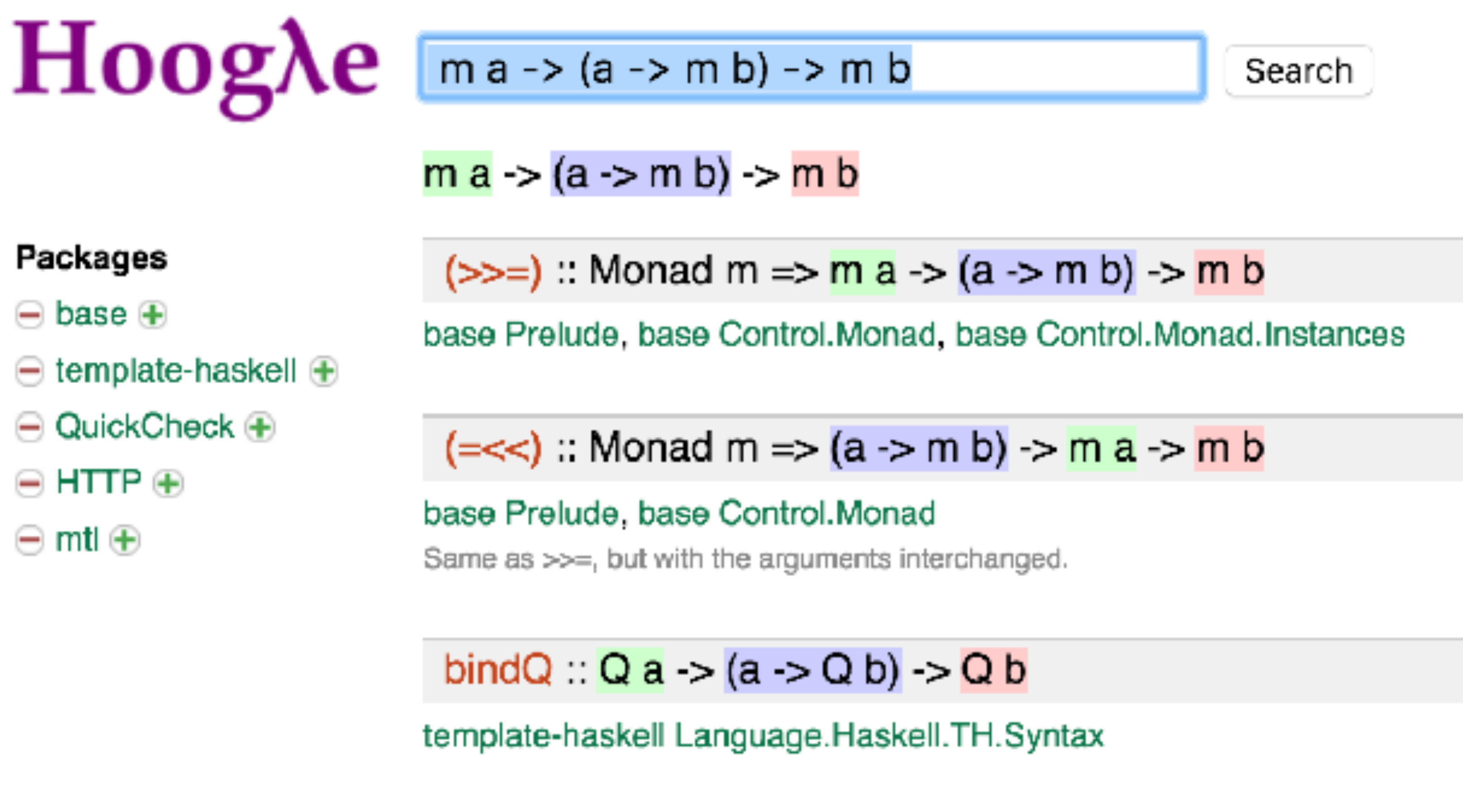
A Type Class for our Pattern

```
class Chain m where
    comb' :: m a -> (a -> m b) -> m b
    wrap  :: a -> m a

instance Chain Maybe where
    comb' Nothing _     = Nothing
    comb' (Just x)  f   = f x
    wrap  a             = Just a
```

```
mothersPaternalGrandfather' :: Sheep -> Maybe Sheep
mothersPaternalGrandfather' s = (wrap s) `comb'` mother `comb'` father `comb'` father
```

Did we invent something new ?



The monad type class

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



-- Defined in ‘GHC.Base’
instance Monad (Either e) -- Defined in ‘Data.Either’
instance Monad [] -- Defined in ‘GHC.Base’
instance Monad Maybe -- Defined in ‘GHC.Base’
instance Monad IO -- Defined in ‘GHC.Base’
instance Monad ((->) r) -- Defined in ‘GHC.Base’
```

Using the maybe monad

```haskell
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

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

```haskell
-- we can use monadic operations to build complicated sequences
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = (return s) >>= mother >>= father

fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = (return s) >>= father >>= mother >>= mother
```

Do notation

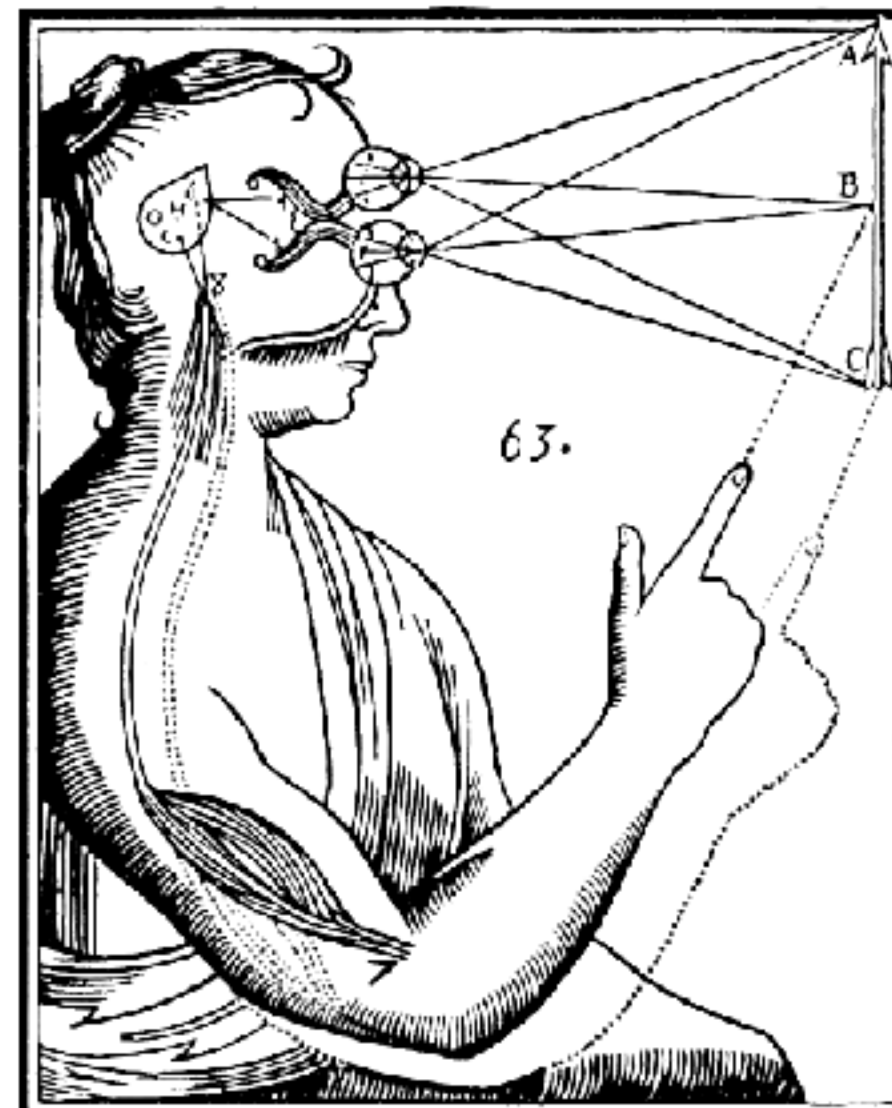
```
fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = do f  <- father s
                                  gm <- mother f
                                  mother gm

                            ~~

mothersPaternalGrandfather s = father s >>= (\f ->
                               mother f >>= (\gm ->
                               mother gm))
```

Commands

The mind body-problem



“pijnappelklier”

Printing a character

command !

```
putChar :: Char -> IO ()

-- denotes the command that,
if it is ever performed,
will print an exclamation mark.
putChar '!'
```

Combining Commands

Then

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

--denotes the command that, if it is ever
performed, prints a question mark followed by an
exclamation mark.
putChar '?' >> putChar '!'
```

Do nothing

command !

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

The term done doesn't actually do nothing; it just specifies the command that, *if it is ever performed*, won't do anything. (Compare thinking about doing nothing to actually doing nothing: they are distinct enterprises.)

Printing a string

command !

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


putStr “?!"       = putChar ’?’ >> (putChar ’!’ >> done)
```

More compactly

```haskell
f (g x) = (f . g) x
```

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

Then >>

```
--Properties
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?*

```haskell
main :: IO ()
main =  putStr "?!"
```

Running this program prints an indicator of perplexity:

```
xtofs$ runghc main.hs
?! xtofs$
```

Thus main is the link from Haskell's mind to Haskell's body

Print with a newline

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

main :: IO ()
main =  putStrLn “?!"
```

Equational Reasoning

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

```
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").
```

Commands with a value

Reading Characters

```haskell
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

```
return :: a -> IO a

return [] :: IO String
```

`when the input contains "bc" yields the value []`
`and an unchanged input "bc"`

Combining commands with values

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

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

In detail

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

```
m :: IO a
k :: a -> IO b
m >>= k :: IO b
```

if it is ever performed

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

```
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".

Done and then revised

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

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

Echo

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

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

```
xtofs$ runghc echo.hs
Knowledge is knowing a tomato is a fruit; wisdom is not putting it in a fruit salad.
KNOWLEDGE IS KNOWING A TOMATO IS A FRUIT; WISDOM IS NOT PUTTING IT IN A FRUIT SALAD.
```

Do notation

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

getLine :: IO String
getLine = do {
               x <- getChar;
               if x == ’\n’ then
                  return []
               else do {
                  xs <- getLine;
                  return (x:xs)
                }
             }
```

Do notation

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


echo :: IO ()
echo = do {
             line <- getLine;
             if line == "" then
                 return ()
             else do {
                putStrLn (map toUpper line);
                echo
             }
           }
```

Do revised

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

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

Monoids & Monads

Monoids

A *monoid* is a pair of an operator (@@) and a value u,

where the operator has the value as **identity** and is **associative**.

```
u@@x         = x
x@@u         = x
(x@@y)@@z = x@@(y@@z)
```

(+) and 0

(∗) and 1

(||) and False

(&&) and True

(++) and []

(>>) and done

Monad Laws


```
return a >>= f    ≡ f a
m >>= return      ≡ m
(m >>= f) >>= g   ≡ m >>= (\x -> f x >>= g)
```

Left identity


```
do { x' <- return x;
     f x'
   }
```

≡

```
do { f x }
```

Right Identity


```
do { x <- m;
     return x
   }
```

≡

```
do { m }
```

Associativity


```
do { y <- do { x <- m;
               f x
             }
     g y
   }
```

≡

```
do { x <- m;
     do { y <- f x;
          g y
        }
   }
```

≡

```
do { x <- m;
     y <- f x;
     g y
   }
```

Interlude Modules

Defining Modules

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Using Modules

```haskell
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Abstract Data Types

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

data Tree a              = Leaf a | Branch (Tree a) (Tree a)

leaf                     = Leaf
branch                   = Branch
cell   (Leaf a)          = a
left   (Branch l r)      = l
right  (Branch l r)      = r
isLeaf    (Leaf _)       = True
isLeaf    _              = False
```

Making our own IO Monad

Our own IO monad

```
module MyIO(MyIO, myPutChar, myGetChar, convert) where

type Input     = String
type Remainder = String
type Output    = String

data MyIO a    =  MyIO (Input -> (a, Remainder, Output))

apply :: MyIO a -> Input -> (a, Remainder, Output)
apply (MyIO f) inp  =  f inp
```

Our own IO monad

```
myPutChar :: Char -> MyIO ()
myPutChar ch  =  MyIO (\inp -> ((), inp, [ch]))

myGetChar :: MyIO Char
myGetChar =  MyIO (\(ch:rem) -> (ch, rem, []))
```

```
apply myGetChar "abc"   ==  ('a', "bc", "")
apply myGetChar "bc"    ==  ('b', "c", “")

apply (myPutChar 'A') "def"   ==  ((), "def", "A")
apply (myPutChar 'B') "def"   ==  ((), "def", "B")
```

Our own IO monad

```
instance Monad MyIO where
  return x  =  MyIO (\inp -> (x, inp, ""))
  m >>= k   =  MyIO (\inp ->
                 let (x, rem1, out1) = apply m inp in
                 let (y, rem2, out2) = apply (k x) rem1 in
                 (y, rem2, out1++out2))
```

```
apply (myGetChar >>= \x -> myGetChar >>= \y -> return [x,y])  "abc"
==
("ab", "c", "")

apply (myPutChar 'A' >> myPutChar 'B')  "def"
==
((), "def", "AB")

apply (myGetChar >>= \x -> myPutChar (toUpper x)) "abc"
==
((), "bc", "A")
```

Our own IO monad

```
convert :: MyIO () -> IO ()
convert m  =  interact (\inp ->
                let (x, rem, out) = apply m inp in
                out)
```

```
interact :: (String -> String) -> IO ()
```

Using MyIO

```haskell
import Data.Char
import MyIO

myPutStr :: String -> MyIO ()
myPutStr =  foldr (>>) (return ()) . map myPutChar

myPutStrLn :: String -> MyIO ()
myPutStrLn s   =  myPutStr s >> myPutChar '\n'
```

Using MyIO

```
myGetLine :: MyIO String
myGetLine =  do {
               x <- myGetChar;
               if x == '\n' then
                 return []
               else do {
                 xs <- myGetLine;
                 return (x:xs)
               }
             }
```

Using MyIO

```
myEcho :: MyIO ()
myEcho =  do {
            line <- myGetLine;
            if line == "" then
              return ()
            else do {
              myPutStrLn (map toUpper line);
              myEcho
            }
          }
```

Using MyIO

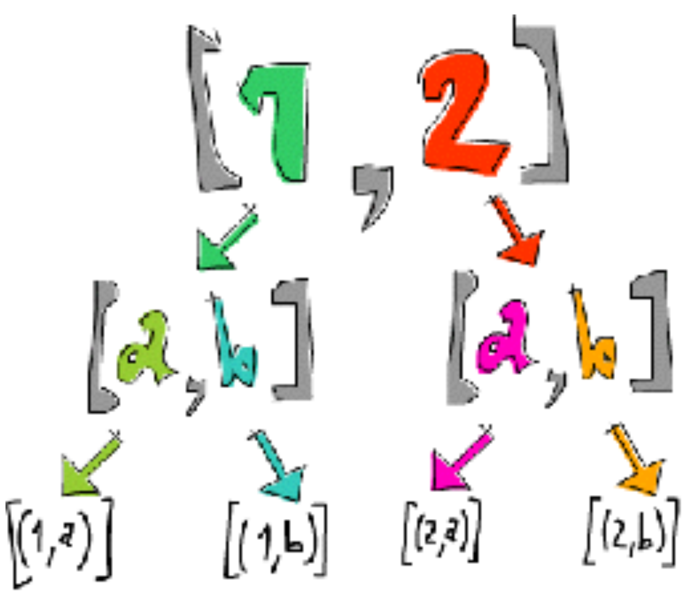
```haskell
main :: IO ()
main =  convert myEcho
```

```
xtofs$ runghc MyEcho.hs
lots of the people  were  mean,  and most of them were miserable, even the ones with digital watches
LOTS OF THE PEOPLE  WERE  MEAN,  AND MOST OF THEM WERE MISERABLE, EVEN THE ONES WITH DIGITAL WATCHES
```

Using MyIO

```
myEcho :: MyIO ()
myEcho =  do {
            line <- myGetLine;
            if line == "" then
              return ()
            else do {
              myPutStrLn (map toUpper line);
              myEcho
            }
          }
```

List Monad



List Monad

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

instance Monad [] where
  return x = [ x ]
  m >>= k  = [ y | x <- m, y <- k x]

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

Using our List Monad

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

pairs' n = [1..n] >>= \i ->  [i+1..n]  >>= \j -> return (i,j)

pairs'' n =  do i <- [1   .. n]
                j <- [i+1 .. n]
                return (i,j)
```

```
Ok, modules loaded: Main.
*Main> pairs 5
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
*Main> pairs' 5
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
*Main> pairs'' 5
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

Monad Plus

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


instance MonadPlus [] where
  mzero = []
  mplus = (++)


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

Monads Plus

```
pairsG  n = [(i,j) | i<-[1..n], j <- [1 .. n ] , i < j ]

pairsG' n = [1..n] >>= \i -> [i+1..n]  >>= \j ->
            guard (i<j) >>= \_ -> return (i,j)

pairsG’' n = do i <- [1 .. n]
                j <- [1 .. n]
                guard (i<j)
                return (i,j)
```

Parsing

Parsing

```haskell
-- First attempt
type Parser a = String -> a
-- Second attempt
type Parser a = String -> (a, String)
-- Final 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

Parsing

```haskell
-- The type of parsers
newtype 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  =  one [ x | (x,t) <- apply m s, t == "" ]
  where
  one []                 =  error "no parse"
  one [x]                =  x
  one xs | length xs > 1  =  error "ambiguous parse"
```

Parsing is a monad

```
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 ])
```

Parsing is a MonadPlus

```haskell
--   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)
```

Characters

```haskell
-- Parse one 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 }

-- Match a given character
token :: Char -> Parser Char
token c  =  spot (== c)
```

Characters

```
apply (spot isDigit) "123ab" = [('1', "23ab")]
apply (spot isDigit) "ab"    = []
apply (token 'a') "ab"       = [('a', "b")]
```

Parsing a string

```
-- match a given string (defined two ways)
match :: String -> Parser String
match []        =  return []
match (x:xs)    =  do {
                     y <- token x;
                     ys <- match xs;
                     return (y:ys)
                   }

match' :: String -> Parser String
match' xs   =  sequence (map token xs)
```

Matching 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)
```

Matching Numbers

```
-- 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
```

Matching Numbers

```haskell
-- 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
```

Parsing Expressions

```haskell
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 Expressions

```
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) }
```

Using our parser

```
parse parseExp "(1+(2*3))" == (Lit 1 :+: (Lit 2 :*: Lit 3))
parse parseExp "((1+2)*3)" == ((Lit 1 :+: Lit 2) :*: Lit 3)
```

```
*Exp> parse parseExp "(1+(2*3))"
Lit 1 :+: (Lit 2 :*: Lit 3)
*Exp> evalExp (parse parseExp "(1+(2*3))")
7
*Exp> parse parseExp "((1+2)*3)"
(Lit 1 :+: Lit 2) :*: Lit 3
*Exp> evalExp (parse parseExp "((1+2)*3)")
9
```