

Monads

David Sands

Monads seen so far: IO vs Gen



- Instructions to build a value of type A by interacting with the operating system
- Run by the ghc runtime system
- Instructions to create a random value of type A
- Run by the QuickCheck library functions to perform random tests

Terminology

- A “*monadic value*” is just an expression whose type is an instance of class Monad
- “*t is a monad*” means t is an instance of the class Monad
- We have often called a monadic value an “*instruction*”. This is not standard terminology – but sometimes they are called “*actions*”

Parsing

- So far: how to write

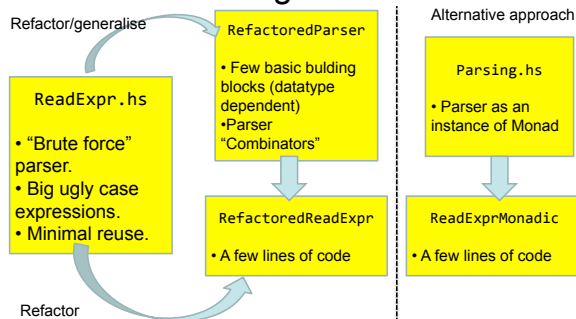
```
readExpr :: String -> Maybe Expr
```

- Key idea:

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

- This lecture: Building Parsers; Parsers as a new type of “instructions” – i.e. a monad.

The Big Picture



Recall some key building blocks

```

succeed :: a -> Parser a
succeed a = P $ \s -> Just(a,s)

sat :: (Char -> Bool) -> Parser Char

(>->) :: Parser a -> Parser b -> Parser b
(>*>) :: Parser a -> (a -> Parser b) -> Parser b
  
```

```

Main> parse (digit >*> \a -> sat (==a)) "22xx"
Just ('2',"xxx")
Main> parse (digit >*> \a -> sat (==a)) "12xx"
Nothing
  
```

The Parser Monad

- Using these building blocks we can make Parser an instance of the class Monad
 - We get a language of "Parsing Instructions"
 - Another way to write Parsers using do notation
 - Deeper understanding of Monads

Monads seen so far: IO vs Gen



- IO A**
 - Instructions to build a value of type A by interacting with the operating system
 - Run by the ghc runtime system
- Gen A**
 - Instructions to create a random value of type A
 - Run by the QuickCheck library functions to perform random tests

Monads = Instructions

- What is the type of doTwice?

```
Main> :i doTwice
doTwice :: Monad a => a b -> a (b,b)
```

Even the *kind of instructions* can vary!
Different kinds of instructions, depending on who obeys them.

Whatever kind of result argument produces, we get a pair of them

IO means operating system.

Monads and do notation

- To be an instance of class Monad you need (as a minimal definition) two operations: **>>=** and **return**

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

  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

Default implementations

Monad

- To be an instance of class Monad you need two operations: **>>=** and **return**

```
instance Monad Parser where
  return = succeed
  (>>=) = (>*>)
  -- (>->) is equivalent to (>>)
```

- Why bother?

•First example of a home-grown monad
•Can understand and use do notation

The truth about Do

- Do syntax is just a shorthand:

```
do act1
  act2 == act1 >> act2 == act1 >>= \_ -> act2
```

```
do v <- act1
  act2 == act1 >>= \v -> act2
```

The truth about Do

Full translation (I)

```
do act1
  ...
  actn == act1 >> do ...
                actn

do v <- act1
  ...
  actn == act1 >>= \v -> do ...
                actn

do actn == actn
```

The truth about Do

Full Translation (II): Let and pattern matching

```
do let p = e == let p = e in
  ...
  actn

do pattern <- act1 == let f pattern = do ...
  ...
  actn                f _ = fail "Error"
  in act1 >>= f

==
```

Example

- recall doTwice

```
doTwice :: Monad m => m a -> m (a,a)
doTwice cmd =
  do a <- cmd
     b <- cmd
     return (a,b)
```

```
Main> parse (doTwice number) "9876"
Just (('9','8'), '76')
```

Example revisited: Parsing Expressions

```
expr :: Parser Expr
expr s1 = case parse num s1 of
  Just (a,s2) -> case s2 of
    '+':s3 -> case parse expr s3 of
      Just (b,s4) -> Just (Add a b, s4)
      Nothing -> Just (a,s2)
    _ -> Just (a,s2)
  Nothing -> Nothing
```

modified to use the new version of Parser type. Otherwise as before

Monadic style abstracts away from implementation of the Parser type

```
expr :: Parser Expr
expr = do a <- num
        do char '+'
          b <- expr
          return (Add a b)
        +++ return a
```

Parser Combinators

```
zeroOrMore, oneOrMore :: Parser a -> Parser [a]
zeroOrMore p = oneOrMore p +++ return []
oneOrMore p = do v <- p
                vs <- zeroOrMore p
                return (v:vs)
```

```
Main> parse (oneOrMore number) "9876+"
Just ("9876", "+")
```

Combinator: a function which take functions as arguments and produces a function as a result

Parser Combinators

```
nat :: Parser Int -- Parses a non negative integer
nat = do xs <- oneOrMore number
        return (read xs)

int :: Parser Int
int = nat +++
      do char '-'
        n <- nat
        return (-n)
```

Chain

```
chain p op f = P $ \s1 ->
  case parse p s1 of
  Just (a,s2) -> case s2 of
    c:s3 | c == op -> case chain p op f s3 of
      Just (b,s4) -> Just (f a b, s4)
      Nothing -> Just (a,s2)
    _ -> Just (a,s2)
  Nothing -> Nothing
```

Old definition (modified to work with the new type)

```
chain p op f = do v <- p
  vs <- zeroOrMore (char op >> p)
  return (foldr1 f (v:vs))
```

Prelude.foldr1 : fold operation for lists with at least one element (no "nil" case)

Factor

```
factor :: Parser Expr
factor ('':s) =
  case expr s of
  Just (a, ':s1) -> Just (a, s1)
  - -> Nothing
```

```
factor s = num s
```

```
factor :: Parser Expr
factor = num +++
  do char '('
  e <- expr
  char ')'
```

Summary

- We can use higher-order functions to build Parsers from other more basic Parsers.
- Parsers can be viewed as an instance of Monad
- We can build our own Monads!
 - A lot of "plumbing" is nicely hidden away
 - The implementation of the Monad is not visible and can thus be changed or extended

IO t

- Instructions for interacting with operating system
- Run by GHC runtime system produce value of type t

Gen t

- Instructions for building random values
- Run by **quickCheck** to generate random values of type t

Parser t

- Instructions for parsing
- Run by **parse** to parse a string and **Maybe** produce a value of type t

Three Monads

Code

- Parsing.hs
 - module containing the parser monad and simple parser combinators.
- ReadExprMonadic.hs
 - A reworking of Read

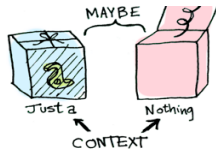
See course home page

A fun blog post about functors, applicatives and monads

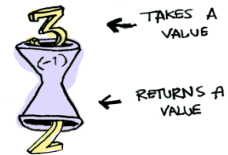
<http://adit.io/posts/>

2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

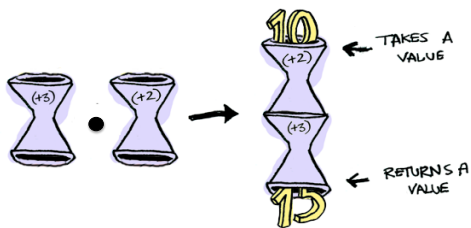
Maybe



Here is a function

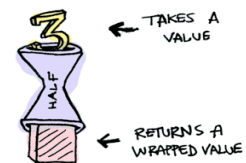


They can be composed



Here is a function

```
half x = if even x  
        then Just (x `div` 2)  
        else Nothing
```



What if we feed it a wrapped value?



We need to use `>>=` to shove our wrapped value into the function

`>>=`



>>=

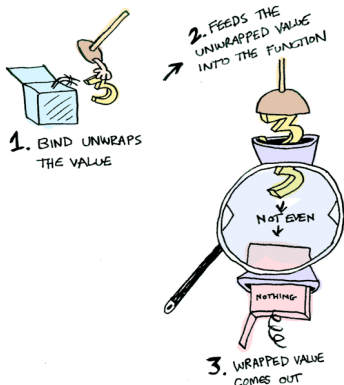
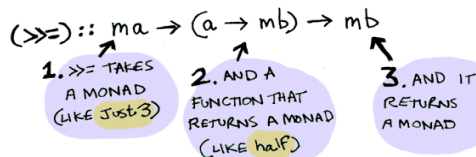
Here's how it works:

```
> Just 3 >>= half
Nothing
> Just 4 >>= half
Just 2
> Nothing >>= half
Nothing
```

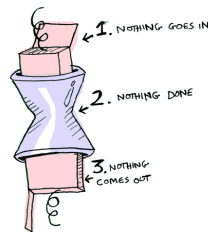
What's happening inside? Monad is another typeclass. Here's a partial definition:

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

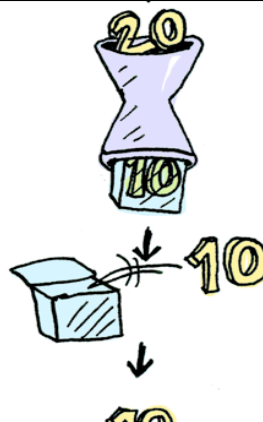
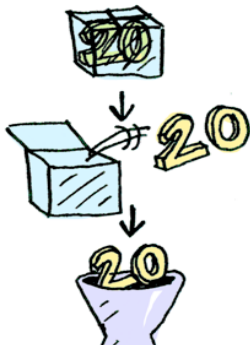
>>=

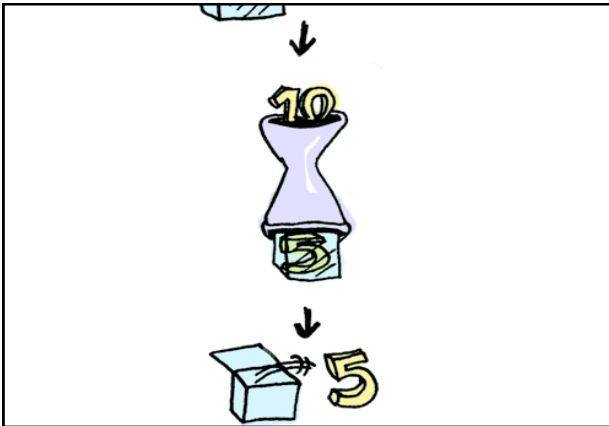


>>=



Just 20 >>= half >>= half >>= half





Maybe another Monad

- Maybe is a very simple monad

```
instance Monad Maybe where
  Just x >>= k = k x
  Nothing >>= _ = Nothing

  return      = Just
  fail s      = Nothing
```

Although simple it can be useful...

Congestion Charge Billing



Congestion Charge Billing

Registration number used to find the Personnummer of the owner

```
carRegister :: [(RegNr,PNr)]
```

Personnummer used to find the name of the owner

```
nameRegister :: [(PNr,Name)]
```

Name used to find the address of the owner

```
addressRegister :: [(Name,Address)]
```

Example: Congestion Charge Billing

```
type CarReg = String ; type PNr = String
type Name = String ; type Address = String

carRegister :: [(CarReg,PNr)]
carRegister =
  [ ("JBD 007", "750408-0909"), ... ]

nameRegister :: [(PNr,Name)]
nameRegister =
  [ ("750408-0909", "Dave"), ... ]

addressRegister :: [(Name,PNr),Address]
addressRegister =
  [ ("Dave", "750408-0909"), "42 Streetgatan\n Askim", ... ]
```

Example: Congestion Charge Billing

With the help of

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

we can return the address of car owners

```
billingAddress :: CarReg -> Maybe (Name, Address)
billingAddress car =
  case lookup car carRegister of
    Nothing -> Nothing
    Just pnr -> case lookup pnr nameRegister of
      Nothing -> Nothing
      Just name ->
        case lookup (name,pnr) addressRegister of
          Nothing -> Nothing
          Just addr -> Just (name,addr)
```

Example: Congestion Charge Billing

Using the fact that Maybe is a member of class Monad we can avoid the spaghetti and write:

```
billingAddress car = do
  pnr <- lookup car carRegister
  name <- lookup pnr nameRegister
  addr <- lookup (name,pnr) addressRegister
  return (name,addr)
```

Example: Congestion Charge Billing

Unrolling one layer of the do syntactic sugar:

```
billingAddress car ==
  lookup car carRegister >>= \pnr ->
  do
    name <- lookup pnr nameRegister
    addr <- lookup (name,pnr) addressRegister
    return (name,addr)
```

- lookup car carRegister gives Nothing then the definition of >>= ensures that the whole result is Nothing
- return is Just

Another Example: A Stack

- A Stack is a stateful object
- Stack operations can push values on, pop values off, add the top elements

```
type Stack = [Int]
newtype StackOp t = StackOp (Stack -> (t,Stack))

-- the type of a stack operation that produces
-- a value of type t
pop :: StackOp Int
push :: Int -> StackOp ()
add :: StackOp ()
```

Running a StackOp

```
type Stack = [Int]
newtype StackOp t = StackOp (Stack -> (t,Stack))

run (StackOp f) = f

-- run (StackOp f) state = f state
```

Operations

```
pop :: StackOp Int
pop = StackOp $ \(x:xs) -> (x,xs) -- can fail

push :: Int -> StackOp ()
push i = StackOp $ \(s) -> ((),i:s)

add :: StackOp ()
add = StackOp $ \(x:y:xs) -> ((),x+y:xs) -- can fail
```

Building a new StackOp...

```
swap :: StackOp ()
swap = StackOp $ \(s) ->
  let (x,s') = run pop s
      (y,s'') = run pop s'
      (_,s''') = run (push x) s''
      (_,s''') = run (push y) s'''
  in ((), s''')
```


StackOp is a Monad

- Stack instructions for producing a value

```
-- (>>=) :: StackOp a -> (a -> StackOp b) -> StackOp b
instance Monad StackOp
  where return n = StackOp $ \s -> (n,s)
        sop >>= f = StackOp $ \s ->
            let (i,s') = run sop s
            in run (f i) s'
```

Stack t

- Stack instructions producing a value of type t
- Run by **run**

Maybe t

- Instructions for either producing a value or nothing
- Run by ?? (not an abstract data type)

Two More Monads

Summary: Parsing

- We can use higher-order functions to build Parsers from other more basic Parsers.
- Parsers can be viewed as an instance of Monad

- We can build our own Monads!
 - A lot of "plumbing" is nicely hidden away
 - A powerful pattern, used widely in Haskell
 - A pattern that can be used in other languages, but syntax support helps
 - F# computation expressions
 - Scala