

# Intro to Monads and Monadic Programming

CS395T - Domain Specific Languages  
Greg Lavender  
Department of Computer Sciences  
The University of Texas at Austin



## Semantics of Programming Languages

- Well-known methods
  - Denotational (Scott Strachey)
  - Axiomatic (Floyd-Hoare Logic)
  - Structured Operational (Plotkin)
- Less well-known methods
  - Action Semantics (Mosses)
  - Algebraic Semantics (Goguen)
  - Categorical Semantics (Moggi)
    - based on Category Theory



## Denotational Semantics

- **Contributions**
  - provides mathematical models of a language
  - documents the meaning of a language
    - so that you can reason about a program's correctness
  - provides insight into the features of the language
    - useful for implementers
- **Limitations**
  - semantics is not modular in the sense that a language extension may require a completely rewritten denotational description
  - descriptions often not reusable in other language contexts
  - does not scale up with the complexity of "real" languages
  - would like a semantics that is modular, so features can be studied in isolation and then combined as needed to form a language, so structure & composition rules become important

3/7/05

3



## Category Theory

- **Invented in 1940s by S. Eilenberg and S. MacLane**
  - Abstract algebraic way of reasoning about mathematical structures in terms of mappings between them
    - focus is on the "morphisms" within and between structures not the type of objects in the structure
    - categorical recasting of: sets, groups, vector spaces, algebraic topology, algebraic geometry, etc.
- **"Functorial Semantics" arose in early 1960s**
  - F. W. Lawvere dissertation
  - Foundations of mathematics
    - categorical logic - study of the abstract structure of logical systems
      - set theory, axiomatic number theory, proof theory, model theory, recursion theory, structure & composition

3/7/05

4



## What Category Theory Offers

- A category is a domain of mathematical discourse framed in a very general way
  - Provides a way of thinking about mathematical/logical *structures* and the *algebra of morphisms* between structures
    - abstracts away from the concrete objects of the structure
    - e.g., instead of thinking about a set of integers and relations between integers (e.g.,  $a < b$ ), we think about sets as structures and morphisms (mappings) between sets
  - category theory is a set of tools for stating results that can be used across a spectrum of mathematical domains
- **Applications in Computer Science**
  - datatype constructors studied as "adjoint functors"
  - lambda calculus studied as a "cartesian closed category"
  - various notions of computation and sequencing of computations studied as a "monad"

3/7/05

5




## Moggi's "Notions of Computation"

- In categorial terms
  - the objects on which we compute are types
  - the morphisms between such objects are programs
    - "programs" as used here are just some kind of computation
- **Kinds of computations**
  - computations with side-effects
    - program: store  $\rightarrow$  (val, store')
  - computations with exceptions
    - program :  $A \rightarrow B + \text{Exception}$
  - partial computations
    - program:  $A \rightarrow B + \perp$
  - non-deterministic computations
    - program:  $A \rightarrow \{B, C, D, \dots\}$
  - interactive Input and Output

3/7/05


6



## Monads Formally

- We are familiar with the functional notation
  - $f: a \rightarrow b$ 
    - where  $f$  takes an argument of type  $a$  and computes a value of type  $b$
- We replace this notion by
  - $f: a \rightarrow M b$ 
    - Where  $f$  takes an argument of type  $a$  and computes a value of type  $b$  with the additional effect captured by  $M$
    - an "effect" may be to act on state, generate an exception, do input/output, etc.
- Operations on type  $M$ 
  - $\text{unit} :: a \rightarrow M a$ 
    - turns a value into a computation
  - $(*) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b$ 
    - Applies a function of type  $a \rightarrow M b$  to a computation of type  $M a$ . We provide the computation argument first

3/7/05 7



## Monads Formally

- A Monad is a triple  $(M, \text{unit}, *)$ 
  - $M$  is a type constructor
  - Unit and  $*$  are operations on the polymorphic types of  $M$
  - An expression  $m * \lambda x.n$  means:
    - perform computation  $m$ , bind  $x$  to the resulting value, then perform computation  $n$
    - $m :: M a$
    - $x :: a$
    - $n :: M b$
    - $\lambda x.n :: a \rightarrow M b$
    - $(m * \lambda x.n) :: M b$
    - $\text{let } x = m \text{ in } n$

3/7/05 8

## Monads in Haskell

- The Monad type class and its combinators

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m
  p >> q = p >>= \ _ -> q
  fail s = error s
```

- The operator `>>=` is called the “bind” operator. It implements sequential composition
  - `return x` means given `x::a` return a monad `m a`
  - `(x >>= f)` means compute `x::m a` to get a result value `r::a` that is then passed to `f::a->m b` which produces a new monad `m b`
  - `(x >> y)` is used to combine actions that do not pass result values so no need for the function `f::a->m b`
    - left/right unit laws: `return a >>= f = f a` and `p >>= return = p`
    - assoc. law: `p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g`

3/7/05

9

## List is a Monad

- List datatype is builtin but is essentially:

```
data [a] = [] | a : [a] deriving (Eq, Ord)
```

```
instance Monad [ ] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  []      >>= f = []
  return x    = [x]
  fail s      = []
```

- note that the bind operator `>>=` is recursively invoked such that the entire list is processed recursively until `[]` and a new list is constructed using `++` (concat)
- return x constructs a singleton list monad

3/7/05

10



## Haskell "do" notation

- Syntactic sugar for `>>=` operator

```
do { x <- e; s } = e >>= \x-> do {s}
do (e; s) = e >> do {s}
do {e} = e

sequence      :: Monad m => [m a] -> m [a]
sequence []   = return []
sequence (c:cs) = do x <- c
                    xs <- sequence cs
                    return (x:xs)

-- alternatives
sequence (c:cs) = do { x<-c; xs<-sequence cs; return(x:xs) }

sequence (c:cs) = c >>= (\x -> cs >>= (\xs -> (sequence cs
>> return (x:xs))))
```

3/7/05

11



## List Comprehension is Monad Sequencing

- Cross product using a list comprehension

```
xprod :: [Int] -> [Int] -> [(Int,Int)]
xprod xs ys = [(x,y) | x<-xs, y<-ys]
```

- Which is syntactic sugar for the do notation

```
xprod :: [Int] -> [Int] -> [(Int,Int)]
xprod xs ys = do { x<-xs; y<-ys; return (x,y) }
```

- Which is syntactic sugar for the monad bind operator `>>=` defined for the List monad instance

```
xprod :: [Int] -> [Int] -> [(Int,Int)]
xprod xs ys = xs >>= (\x -> ys >>=
(\y -> return (x,y)))
```

3/7/05

12

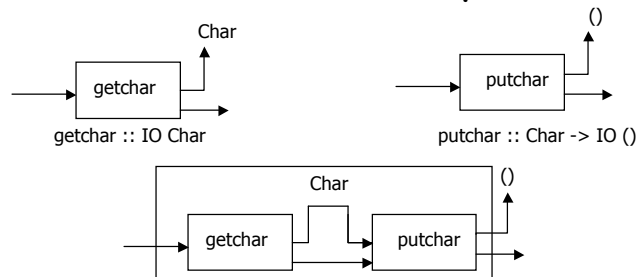
## Monadic IO

- IO presents a real problem for a pure functional language because it is inherently a side-effecting computation
  - monads "saved the day" for functional programming purists since monads provide a functional semantics for how to do IO
  - A value of type "IO a" is an action (computation) that when performed may do some I/O before delivering a value of type a.
  - What does this mean?
    - type IO a = World -> (a, World)
    - intuitively: a value of type IO a is a function that takes the "state of the world" as input and produces a pair consisting of a value of type "a" and a new world
    - This idea is rather like the idea of a continuation and continuation passing style

3/7/05

13

## IO Monad Examples



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

```
echo :: IO ()
echo = getChar >>= putChar
```

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

```
echoDup :: IO ()
echoDup = getChar >>=
  (\c -> (putChar c >> putChar c))
```

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
  getChar >>= \c2 ->
  return (c1,c2)
```

3/7/05

14



## Monadic Parsing

- Monads are used to define flexible recursive descent parsers using combinators instead of relying on a separate parser generator tool
  - this allows you to write the grammar for a language in terms of a family of parsers and combinators in Haskell that can be combined to perform most parsing actions
  - Advantages:
    - Can combine the lexer and the parser into one program
    - parsers can be predictive
    - LL(k) but can deal with ambiguous constructs and  $k > 1$
    - easier to debug
  - Disadvantages
    - May not be as fast as a generated bottom up LR(k) parser

3/7/05



## Monadic Parser

- A polymorphic parser type

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

- a parser takes a string and returns a list of pairs
- an empty list means the parse failed
- a list with one or more pairs means the parse succeeded
  - a list of a single pair implies that a single item was parsed
  - a list of more than one pair implies there was more than one possible way to parse
  - the pairs consist of a parsed value  $v::a$  representing the parsed prefix of the input string and the suffix of the input string that was unparsed

3/7/05

16



## Monadic Parsers

```
-- parse is just a deconstructor function
parse :: Parser a -> String -> [(a, String)]
parse (Parser p) = p

instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f = Parser (\cs -> concat [parse (f a) cs' |
                                   (a,cs') <- parse p cs] )
```

3/7/05

17



## Simple Parser Example

```
-- simple character parser
item :: Parser Char
item = Parser (\xs -> case xs of
  ""      -> []
  (c:cs) -> [(c,cs)])

-- a parser that consumes three characters, returning the
  first and last as a pair

p :: Parser (Char, Char)
p = do { x <- item; item; z <- item; return (x,z) }
```

3/7/05

18

## Parser Combinators

- We need a couple of monad class extensions

```
class Monad m => MonadZero m where
  zero :: m a

class MonadZero m => MonadPlus m where
  (++) :: m a -> m a -> m a
  -- non-deterministic choice operator

instance MonadZero Parser where
  zero = Parser (\cs -> [])

instance MonadPlus Parser where
  p ++ q = Parser (\cs -> parse p cs ++ parse q cs)
  -- applies both parsers p & q to string and returns
  -- their results
```

```
Note: zero ++ p = p; p ++ zero = p
      p ++ (q ++ r) = (p ++ q) ++ r
```

3/7/05

19

## Parser Combinators

- To write parsers, we need a collection of useful parser combinators

```
-- deterministic choice, return first of many
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p ++ q) cs of
                          [] -> []
                          (x::xs) -> [x])

-- conditional parsing
sat :: (Char -> Bool) -> Parser Char
sat p = do { c <- item; if p c then return c else zero }

-- parse a character
char :: Char -> Parser Char
char c = sat (c ==)

-- (recursively) parse a specific string
string :: String -> Parser String
string "" = return ""
string (c:cs) = do { char c; string cs; return (c:cs) }
```

3/7/05

20

## Parser Combinators

- Recursion combinators

```
-- Kleene one & star combinators
many :: Parser a -> Parser [a]
many p = many1 p +++ return []

many1 :: Parser a -> Parser [a]
many1 p = do { a <- p; as <- many p; return (a:as) }

-- parse a sequence tossing the separator
sepby :: Parser a -> Parser b -> Parser [a]
p `sepby` sep = (p `sepby1` sep) +++ return []

sepby1 :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do a <- p
                    as <- many (do {sep; p})
                    return (a:as)
```

3/7/05

21

## Parser Combinators

- Parse repeated applications of a parser *p* separated by applications of a parser *op* whose result is an operator that associates to the left, and combines the results from the *p* parsers

```
chainl :: Parser a -> Parser (a->a->a) -> a -> Parser a
chainl p op a = (p `chainl1` op) +++ return a

chainl1 :: Parser a -> Parser (a->a->a) -> Parser a
p `chainl1` op = do { a<-p; rest a }
                 where
                   rest a = (do {f<-op; b<-p; rest(f a b)} )
                           +++
                           return a
```

3/7/05

22

## Lexical Combinators

- Use lexical combinators to avoid having a separate lexical phase

```
-- parse a string of spaces, tabs and newlines
space :: Parser String
space = many (sat isSpace)

-- parse a token followed by a space
token :: Parser a -> Parser a
token p = do { a <- p; space; return a }

-- parse a specific token symbol
symb :: String -> Parser String
symb cs = token (string cs)

-- apply a parser throwing away leading spaces
apply :: Parser a -> String -> [(a,String)]
apply p = parse (dp {space; p})
```

3/7/05

23

## Putting it all together

- BNF for simple arithmetic expressions

```
expr ::= expr addop term | term
term ::= term mulop factor | factor
factor ::= digit | '(' expr ')'
digit ::= '0' | '1' | ... | '9'
addop ::= '+' | '-'
mulop ::= '*' | '/'

expr :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)
expr = term `chainl1` addop
term = factor `chainl1` mulop
digit = do {x <- token (sat isDigit); return (ord x - ord '0')}
addop = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```

3/7/05

24



## Category Theory Books

- **Mathematical**
  - *Categories for the Working Mathematician*, S. MacLane
  - *Toposes, Triples and Theories*, M. Barr & C. Wells
  - *Practical Foundations of Mathematics*, P. Taylor
- **Logical**
  - *Topoi: A Categorical Analysis of Logic*, R. Goldblatt
  - *Introduction to Higher-Order Categorical Logic*, J. Lambek & P. J. Scott
  - *Elementary Categories, Elementary Toposes*, C. McLarty
  - *Categories for Types*, R. Crole
- **Computer Science**
  - *Basic Category Theory for Computer Scientists*, B. Pierce
  - *Category Theory for Computer Science*, M. Barr & C. Wells
  - *Arrows, Structures & Functors: The Categorical Imperative*, M. Arbib & E. Manes
  - *Computational Category Theory*. D. Rydeard & R. Burstall

3/7/05

25



## Monadic Programming Papers

- **Papers online**
  - *Abstract View of Programming Languages*, E. Moggi
  - *Notions of Computation and Monads*, E. Moggi
  - *Computational Lambda Calculus and Monads*, E. Moggi
  - *Comprehending Monads*, P. Wadler
  - *Monads for Functional Programming*, P. Wadler
  - *The Essence of Functional Programming*, P. Wadler
  - *Tackling the Awkward Squad: monadic I/O, concurrency, exceptions and foreign-language calls in Haskell*, S. Peyton-Jones
  - *Representing Monads*, A. Filinski
  - *Monadic Parsing in Haskell*, G. Hutton & E. Meijer
  - *Parsec, a Fast Combinator Parser*, D. Leijen
  - *Building Interpreters by Composing Monads*, G. Steele

3/7/05

26



## Homework due March 21st

- Read the paper "Monadic Parsing in Haskell"
- Implement the example arithmetic parser
  - I recommend using *GHC* ([www.haskell.org/ghc](http://www.haskell.org/ghc))
  - I recommend using the parsec library
    - See paper on Parsec