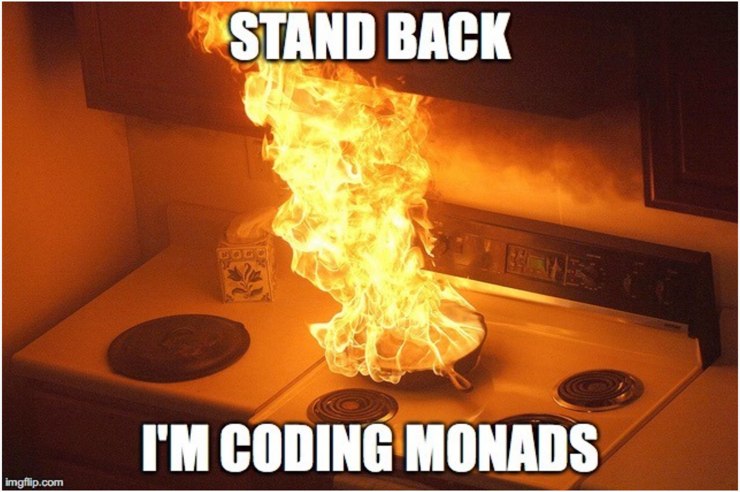
Monadic Parsing: An Introduction

Advanced Programming 2016

Oleks

DIKU

September 20, 2016



— **DIKUMemes** (2016-09-16)

Assignments FAQ

Assignments FAQ

- Due **Sunday 20.00**.

  Q: But can't I just ...

  A: Submit what you have.
  Resubmission is granted on a *descent* attempt.

Assignments FAQ

- Due **Sunday 20.00**.

  Q: But can't I just …

  A: Submit what you have.
  Resubmission is granted on a *descent* attempt.

- Extent of solution

  Q: Am I ***** if my monad doesn't work?

  A: TA's are (programmed by) people.
  You'll be fine. Worst case, resubmission.
  Also, it doesn't have to be perfect.

Assignments FAQ

- Due **Sunday 20.00**.

  Q: But can't I just …

  A: Submit what you have.
  Resubmission is granted on a *descent* attempt.

- Extent of solution

  Q: Am I ***** if my monad doesn't work?

  A: TA's are (programmed by) people.
  You'll be fine. Worst case, resubmission.
  Also, it doesn't have to be perfect.

- Feedback?

  A: We will work harder to meet the Thursday deadline.
  You'll have 7 days *after* feedback to resubmit.
  Oh, and there is OnlineTA…

The good-old `Functor`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The good-old `Maybe Functor`

```haskell
instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

The good-old `Applicative Functor`

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The good-old `Maybe` `Applicative` `Functor`

```
instance Applicative Maybe where
  -- pure :: a -> Maybe a
  pure = Just

  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  Nothing  <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

The good-old `Applicative Functor Monad`

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

  fail   :: String -> m a
  fail = error -- Who put this here!?
```

The good-old `Applicative` `Functor` `Monad` `Maybe`

```haskell
instance Monad Maybe where
  -- return :: a -> Maybe a
  return a        = Just a

  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Just a  >>= f   = f a
  Nothing >>= _   = Nothing

  -- fail :: String -> Maybe a
  fail _          = Nothing
```

Gotta Go Fast? Copy This! (1/2)

```
module M where -- Search and replace " M ".

data M a = M -- Change this.

instance Functor M where
  fmap f m = m >>= \a -> return (f a)

instance Applicative M where
  pure = return
  df <*> dx = df >>= \f -> dx >>= return . f
```

Gotta Go Fast? Copy This! (2/2)

```haskell
instance Monad M where
  -- return :: a -> M a
  return = undefined -- This.

  -- (>>=) :: M a -> (a -> M b) -> M b
  (>>=) = undefined -- And, most notoriously, this.

  -- fail :: String -> M a
  -- fail _ = -- Also this, if it makes sense.
```

The Reader Flavour

Read some data (`d`) and produce some value (`a`).

```haskell
newtype Reader d a = Reader { runRd :: d -> a }

-- Functor, Applicative, ...

instance Monad (Reader d) where
  return a = Reader $ \ _ -> a
  m >>= f = Reader $ \ d ->
    let a = runRd m d
    in runRd (f a) d

getData :: Reader d d
getData = Reader $ \ d -> d

withData :: d -> Reader d a -> Reader d a
withData d m = Reader $ \ _ -> runRd m d
```

The Writer Flavour

Produce some data (s) as a side-effect.

```
newtype Writer s a = Writer { runWr :: (a, s) }

-- Functor, Applicative, ...

instance Monoid s => Monad (Writer s) where
  return a = Writer $ (a, mempty)
  m >>= f = Writer $
    let (a, s1) = runWr m
        (b, s2) = runWr (f a)
    in (b, s1 `mappend` s2)

-- no data to get, so just write:
write :: s -> Writer s ()
write s = Writer ((), s)
```

The (Reader + Writer =) State Flavour (1/2)

Read state (`s`) and produce a value (`a`) and a new state (`s`).

```
newtype State s a = State { runSt :: s -> (a, s) }

-- Functor, Applicative, ...

instance Monad (State s) where
  return a = State $ \ s -> (a, s)
  m >>= f = State $ \ s ->
    let (a, s1) = runSt m s
        (b, s2) = runSt (f a) s1
    in (b, s2)
```

The (Reader + Writer =) State Flavour (2/2)

Read state (`s`) and produce a value (`a`) and a new state (`s`).

```
get :: State s s
get = State $ \ s -> (s, s)

set :: s -> State s ()
set s = State $ \ _ -> ((), s)

with :: s -> State s a -> State s a
with s m = State $ \ _ -> runSt m s
```

Parsing?

Parsing?

`String` $\rightsquigarrow$ `Ast`

Part 1: Matching Strings

Parsing Theory

Languages can be regular, context-free, context-sensitive, etc.

Parsing Theory

Languages can be regular, context-free, context-sensitive, etc.

- Use `lex` for the regular ones.

Parsing Theory

Languages can be regular, context-free, context-sensitive, etc.

- Use `lex` for the regular ones.
- Use `yacc` for the context-free ones.

Parsing Theory

Languages can be regular, context-free, context-sensitive, etc.

- Use `lex` for the regular ones.
- Use `yacc` for the context-free ones.
- Use (??) for the context-sensitive ones.

Parsing Theory

Languages can be regular, context-free, context-sensitive, etc.

- Use `lex` for the regular ones.
- Use `yacc` for the context-free ones.
- Use (??) for the context-sensitive ones.

$$\text{Regular} \subset \text{Context-Free} \subset \text{Context-Sensitive}$$


MONADS
SET YOU FREE
imgflip.com

Monadic Parsing Theory

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

Monadic Parsing Theory

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

Monadic laws:

1. `return a >>= f == f a`
2. `m >>= return == m`
3. `(m >>= f) >>= g == m >>= \ a -> f a >>= g`

Regular Languages

(Strings that can be matched by a regular expression.)

Regular Expressions

<table>
<tr><th>Name</th><th>Example</th><th>Matches</th><th>Doesn’t Match</th></tr>
<tr><td>Chars</td><td>/a/</td><td>"a","ab"</td><td>"b",""</td></tr>
<tr><td>String</td><td>/ab/</td><td>"ab","abc"</td><td>"bc",""</td></tr>
<tr><td>Character Class</td><td>/[a-b]/</td><td>"a","b"</td><td>"c",""</td></tr>
<tr><td>Wildcard</td><td>/./</td><td>"a","b","c"</td><td>""</td></tr>
<tr><td>Choice</td><td>/a|b/</td><td>"a","b"</td><td>"c",""</td></tr>
<tr><td>Option</td><td>/a?/</td><td>"a","aa",""</td><td></td></tr>
<tr><td>Many</td><td>/.*/</td><td>"abc",".*@#",""</td><td></td></tr>
<tr><td>Some</td><td>/.+/</td><td>"abc",".*@#"</td><td>""</td></tr>
</table>

`MatchParser` (1/3)

```
newtype MatchParser a = MatchParser {
  runParser :: String -> Maybe (a, String)
}
```

A State monad with a Maybe flavour.

MatchParser (2/3)

```
instance Monad MatchParser where
  -- return :: a -> MatchParser a
  return a = MatchParser $ \ s -> Just (a, s)

  -- (>>=) :: MatchParser a
  --          -> (a -> MatchParser b)
  --          -> MatchParser b
  m >>= f = MatchParser $ \s -> do
    (a, s’) <- runParser m s
    runParser (f a) s’

  -- fail :: String -> MatchParser a
  fail _ = MatchParser $ \ _ -> Nothing
```

MatchParser (3/3)

```
getc :: MatchParser Char
getc = MatchParser getc’
  where getc’ "" = Nothing
        getc’ (x:xs) = Just (x, xs)
```

MatchParser (3/3)

```
getc :: MatchParser Char
getc = MatchParser getc'
  where getc' "" = Nothing
        getc' (x:xs) = Just (x, xs)

reject :: MatchParser a
reject = MatchParser $ \ _ -> Nothing
```

Parsing

```haskell
parse :: MatchParser a -> String -> Maybe (a, String)
parse = runParser
```

Verbatim Chars

```
/a/
```

```
char :: Char -> MatchParser Char
char c = do
  c' <- getc
  if c == c' then return c else reject
```

Verbatim Strings

`/ab/`

```
string :: String -> MatchParser String
string "" = return ""
string (c:cs) = do
  void $ char c
  void $ string cs
  return (c:cs)
```

Character Classes

/[a-b]/

```
chars :: [Char] -> MatchParser Char
chars cs = do
  c <- getc
  if c `elem` cs then return c else reject
```

Wildcard, First Attempt

```
/./
```

```
wild :: MatchParser ()
wild = do
  _ <- get
  return ()
```

Wildcard, First Attempt

```
/./
```

```haskell
wild :: MatchParser ()
wild = do
  _ <- get
  return ()
```

Works, but awfully verbose…

Wildcard, First Attempt

`/./`

```
wild :: MatchParser ()
wild = do
  _ <- get
  return ()
```

Works, but awfully verbose…

`get :: MatchParser Char`, but we want `MatchParser ()`

Wildcard, First Attempt

`/./`

```
wild :: MatchParser ()
wild = do
  _ <- get
  return ()
```

Works, but awfully verbose…

`get :: MatchParser Char`, but we want `MatchParser ()`

`MatchParser` is an `Applicative` `Functor` `Monad`…

Wildcard, First Attempt

`/./`

```
wild :: MatchParser ()
wild = do
  _ <- get
  return ()
```

Works, but awfully verbose…

`get :: MatchParser Char`, but we want `MatchParser ()`

`MatchParser` is an `Applicative Functor Monad`…

`https://www.haskell.org/hoogle/`

Wildcard

```
/./
```

```haskell
import Control.Monad ( void )

wild :: MatchParser ()
wild = void $ getc
```

Alternative

```haskell
-- From Control.Applicative.
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a
    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a
```

Alternative

```haskell
-- From Control.Applicative.
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a
    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a

    -- | One or more.
    some :: f a -> f [a]
    some v = -- Free!
    -- | Zero or more.
    many :: f a -> f [a]
    many v = -- Free!
```

A Parser with Alternatives

```haskell
import Control.Applicative
  ( Alternative((<|>), empty, many, some) )

reject :: MatchParser a

parse :: MatchParser a -> String -> Maybe (a, String)

instance Alternative MatchParser where
  -- empty :: a -> MatchParser a
  empty = reject

  -- (<|>) :: MatchParser a -> MatchParser a
  --          -> MatchParser a
  p <|> q = MatchParser $ \cs ->
    parse p cs <|> parse q cs
```

Beyond Regex: Rejection

```
keywords :: [String]
keywords = ["if"]

varName :: MatchParser String
varName = do
  cs <- some $ chars ['a'..'z']
  if cs `elem` keywords then reject else return cs
```

Beyond Regex: Strings Of User-Defined Length

```
lenString :: MatchParser String
lenString = do
  cn <- chars ['1'..'9']
  cns <- many $ chars ['0'..'9']
  let n = read (cn:cns) -- See report.pdf
  ntimes n (chars ['a'..'z'])
```

Testing

To claim that your code is correct in your assessment, as a minimum, you should do some sort of testing.

Your test results should be easily reproducible.

Tasty

Tasty is a wrapper around other testing frameworks.

```
$ stack install tasty
$ stack install tasty-hunit
$ stack install tasty-quickcheck
```

Tasty

Tasty is a wrapper around other testing frameworks.

```
$ stack install tasty
$ stack install tasty-hunit
$ stack install tasty-quickcheck
```

Best to keep tests in a separate module.

```haskell
module Tests where

import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck as QC
```

Unit Testing

Test small units of functionality.

Compare to an expected set of results.

`https://wiki.haskell.org/HUnit_1.0_User's_Guide`

Unit Testing with Tasty

```
unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "The reason char returns Char" $
      (parse (many $ chars "ab") "abab") @?=
        (Just ("abab", ""))
  ]
```

Property-Based Testing

State a desired property.

Let your computer generate test-cases
in attempt to falsify this property.

Property-Based Testing in Tasty

```
properties :: TestTree
properties = testGroup "QuickCheck"
  [ QC.testProperty "Fancy property" $
      \(s, v) -> parse (chars s) v ==
        parse (foldl ((<|>)) empty (map char s)) v
  ]
```

Running Tasty Tests

```haskell
tests :: TestTree
tests = testGroup "Tests" [properties, unitTests]

main :: IO ()
main = defaultMain tests
```

Running Tasty Tests

```haskell
tests :: TestTree
tests = testGroup "Tests" [properties, unitTests]

main :: IO ()
main = defaultMain tests
```

```
$ stack exec runhaskell -- -Wall Tests.hs
```

- Add `--quickcheck-verbose` to see the data actually generated as input by QuickCheck.

Next Time

- Parsing grammars
- Spoiler: `[]` is also an instance of `Alternative`
- More about property-based testing

What Now?

- Code, slides, and some exercises with `MatchParser` and Tasty will be published shortly after the lecture.
- Do these at your leisure or at the exercise session.
- Read "Functional Pearls: Monadic Parsing in Haskell" by Graham Hutton and Erik Mejer.
  (Called `pearl.pdf` on Absalon.)
- Read "Grammars and parsing with Haskell Using Parser Combinators" by Peter Sestoft and Ken Friis Larsen.
  (Called `parsernotes.pdf` on Absalon.)