

# Creating an Emacs Diary Parser with Haskell and Parsec

Toby Tripp

March 2018

# Contents

<b>1</b>	<b>“Don’t Fear the Monad”</b>	<b>2</b>
1.1	Outline (7:50) . . . . .	2
1.2	Notation (8:25) . . . . .	2
1.2.1	Composition . . . . .	3
1.3	Monoids (20:40) . . . . .	3
1.4	Monads (30:39) . . . . .	3
<b>2</b>	<b>The Maybe Monad</b>	<b>5</b>
2.1	As Described by Computerphile . . . . .	5
<b>3</b>	<b>Other Monads</b>	<b>6</b>
3.1	The Identity Monad . . . . .	6
3.2	Definition of Maybe . . . . .	6
3.3	The List <i>Monad</i> . . . . .	6
<b>4</b>	<b>Using Parsec</b>	<b>8</b>
4.1	Parsing CSV . . . . .	8
4.2	sepBy and endBy Combinators . . . . .	9
<b>5</b>	<b>Using HUnit to test the Parser</b>	<b>11</b>
5.1	Helper Functions . . . . .	11
5.1.1	Date Helpers . . . . .	11
5.1.2	Parser Testing Helpers . . . . .	11
5.1.3	Writing the Tests . . . . .	12
5.1.4	The Preamble . . . . .	12
5.2	Testing the Parser . . . . .	13
<b>6</b>	<b>The Emacs Diary Parser</b>	<b>16</b>
6.1	Parsing Dates and Times . . . . .	16
6.1.1	Date . . . . .	16
6.1.2	Time . . . . .	17

# Chapter 1

## “Don’t Fear the Monad”

Dr. Beckman, astrophysicist and senior software engineer, begins with a basic introduction to functional programming as a concept [beckman:monad]. Most notably, he focuses on the concept of functions as being replaceable by table-lookups.

### 1.1 Outline (7:50)

1. Functions
2. Monoids
3. Functions
4. Monads

### 1.2 Notation (8:25)

FROM “IMPERATIVE” TO FUNCTIONAL NOTATION :

- $int\ x = x \in int$
- `x :: int`
- $int\ f(int\ x)$
- `f :: int → int`

GIVEN TYPE VARIABLE  $a : \forall a$

- `A x`
- `x :: a`
- `static A f<A>(A x)`
- `f :: a → a`

### 1.2.1 Composition

Given:

```
x :: a
f :: a → a
g :: a → a
```

in imperative style function composition might appear as: `f(g(a))` or in reverse: `g(f(a))`.

In functional style, function application appears as: `g a` and composition can be shown as: `f(g a)`. Parenthesis are necessary due to partial application being left associative. For example, `f h g` is applied as though `(f h) g`.

It is also possible to use a composition operator, `o`, to imply composition: `(f o g) a`. So, given the above 1.2.1, we can deduce:

```
h = (f o g) a = f o g
h :: a → a
```

This does confuse the concepts of `a` as argument and `a` as type, but the point remains clear, I think.

## 1.3 Monoids (20:40)

In abstract algebra, a branch of mathematics, a monoid is an algebraic structure with a single associative binary operation and an identity element.

Monoids are studied in semigroup theory, because they are semigroups with identity.  
<sup>a</sup>:

---

<sup>a</sup>Wikipedia

A *Monoid* is a *Set* with:

1. an associative binary operator (generally composition)
2. an identity value

The operator need not be commutative.

In a programming context, a *Monoid* guarantees type-consistency over function composition.

## 1.4 Monads (30:39)

Given:

```
x :: a
f :: a → M a
g :: a → M a
g :: a → M a
```

`M` is described as a “Type Constructor.”

Again, Dr. Beckman is using `a` to represent both a value of type `a` as well as the type itself `a`. Here he introduced the *Monad* “bind” operator: `>>=`, which he likes to call “shove”:

```
f :: a → M a
g :: a → M a
-- the right hand side is g, but written with
-- a lambda to preserve symmetry
λa → (f a) >>= λa → (g a)
```

The reason to preserve symmetry in the above expression is that the desired expression is “bracketed” as:  $\lambda a \rightarrow [(fa) >>= \lambda a \rightarrow (ga)]$  because the bind operator has type:

```
 $\langle >>= \rangle :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ 
```

That is,  $>>=$  accepts a *Monad* ( $M\ a$ ) and returns a function from  $a \rightarrow M\ a$ .

The functions  $f$ , and  $g$  live in a *Monoid*.  $M\ a$  (the *data*) lives in a *Monad*.

$\langle >>= \rangle$  is the analog of function composition and, therefore, obeys the rules of a *Monoid*. Including associativity and identity.

In a *Monad*, identity is—in Haskell—written as:

```
 $\text{return} :: \text{Monad } m \Rightarrow a \rightarrow m\ a$ 
```

Extended to non-uniform types:

```
 $g :: a \rightarrow Mb$   
 $f :: b \rightarrow Mc$   
 $\lambda a \rightarrow (g\ a) >>= \lambda b \rightarrow (f\ b) :: a \rightarrow Mc$   
 $g >>= \lambda b \rightarrow (f\ b)$ 
```

## Chapter 2

# The Maybe Monad

### 2.1 As Described by Computerphile

[hutton:computerphile]

```
-- Type and 2 type constructors: Val and Div
data Expr = Val Int | Div Expr Expr

-- Val 1
-- Div (Val 6) (Val 2)

unsafe_eval :: Expr → Int
unsafe_eval (Val n) = n
unsafe_eval (Div x y) = div (eval x) (eval y)

-- eval (Div (Val 6) (Val 2))
```

What if `Div` is passed zero? The program will crash. So, error-checking is necessary.

```
safediv :: Int → Int → Maybe Int
safediv n m = if m == 0 then Nothing else Just (div n m)

eval :: Expr → Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval' x of
    Nothing → Nothing
    Just n → case eval' y of
        Nothing → Nothing
        Just m → safediv n m
```

Abstracting the pattern of `case` checking `Maybe` values can be represented as:

Without 'do' notation	With 'do' notation
<pre>eval' :: Expr → Maybe Int eval' (Val n) = return n eval' (Div x y) =     eval' x &gt;&gt;= (λn →         eval' y &gt;&gt;= (λm →             safediv n m))</pre>	<pre>eval'' :: Expr → Maybe Int eval'' (Val n) = return n eval'' (Div x y) = do     n ← eval'' x     m ← eval'' y     safediv n m</pre>

## Chapter 3

# Other Monads

The *Monad* type [thompson:99]:

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

*Monad* comes with some default definitions:

```
m >> k = m >>= λ_ → k
fail s = error s
```

From this definition it can be seen that `>>` acts like `>>=`, except that the value returned by the first argument is discarded rather than being passed to the second argument. [thompson:99]

### 3.1 The Identity Monad

The identity monad [thompson:99] takes a type to itself with definitions:

```
m >>= f = f m
return = id
```

### 3.2 Definition of Maybe

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  return      = Just
  fail s      = Nothing
```

### 3.3 The List *Monad*

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

Lists are, in fact, themselves instances of *Monad*.

```
fmap :: Functor f => (a -> b) -> f a -> f b
(>>=) :: Monad m => m a -> (a -> m b) -> m b
map   :: (a -> b) -> [a] -> [b]
```



# Chapter 4

## Using Parsec

### 4.1 Parsing CSV

```
import Text.ParserCombinators.Parsec
```

[osullivan:rwh]

Input type is a sequence of characters, i.e., a Haskell *String*. *String* is the same as *[Char]*. The return value is *[[String]]*; a list of a list of Strings. We'll ignore *st* for now.

The *do* block implies a *Monad*. *GenParser* is a parsing monad.

*many* is a higher-order function that passes input repeatedly to the function passed as its argument. It collects the return values and treturns them in a list.

```
csvFile :: GenParser Char st [[String]]
csvFile =
  do result <- many line
  eof
  return result
```

A *line* is a list of *cells* followed by *eol*.

```
line :: GenParser Char st [String]
line =
  do result <- cells
  eol
  return result
```

```
cells :: GenParser Char st [String]
cells =
  do first <- cellContent
  next <- remainingCells
  return (first : next)
```

The choice operator, (*<|>*), tries the parser on the left and tries The parser on the right if the left consumes no input.

```
remainingCells :: GenParser Char st [String]
remainingCells =
  (char ',' >> cells) <|> (return [])
```

```
cellContent :: GenParser Char st String
cellContent =
  many (noneOf ",\n")
```

```
eol :: GenParser Char st Char
eol = char '\n'
```

```
parseCSV :: String → Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

## 4.2 sepBy and endBy Combinators

```
import Text.ParserCombinators.Parsec
```

```
csvFile = endBy line eol
line    = sepBy cell (char ',')
cell    = quotedCell <|> many (noneOf ",\n\r")
```

A CSV cell may be either a bare cell or a quoted cell. Since a quoted cell may, itself, contain quotes (doubled for escape) a `quotedCell` is `many quotedChar`.

```
quotedCell =
  do char '"'
  \content ← many quotedChar
  \char ← "<?> \"quote_at_end_of_cell\" -- see eol below
  return content
```

The function `quotedChar` begins by consuming any character that is *not* itself a quote. If it is a quoted character, the stream must be checked for a second consecutive quote. If so, a single quote mark is returned to the result string.

Notice that `try` in `quotedChar` on the right side of `<|>`. Recall that I said that `try` only has an effect if it is on the left side of `<|>`. This `try` does occur on the left side of a `<|>`, but on the left of one that must be within the implementation of `many`.

This `try` is important. Let's say we are parsing a quoted cell, and are getting towards the end of it. There will be another cell following. So we will expect to see a quote to end the current cell, followed by a comma. When we hit `quotedChar`, we will fail the `noneOf` test and proceed to the test that looks for two quotes in a row. We'll also fail that one because we'll have a quote, then a comma. If we hadn't used `try`, we'd crash with an error at this point, saying that it was expecting the second quote, because the first quote was already consumed. Since we use `try`, this is properly recognized as not a character that's part of the cell, so it terminates the `many quotedChar` expression as expected. Lookahead has once again proven very useful, and the fact that it is so easy to add makes it a remarkable tool in `Parsec`.

```
quotedChar =
  noneOf "\""
  \<|> try (string "\n\r") >> return ''
```

`Parsec` also includes combinators for error handling and reporting. A first attempt at an `eol` implementation that handles multiple line-ending styles might appear as:

```
eol' = try (string "\n\r")
      <|> try (string "\r\n")
      <|> string "\n"
      <|> string "\r"
```

```
> parseCSV "line1"
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r"
```

The failure above is unclear and requires knowledge of the parser implementation to debug fully. The monad `fail` function can be used to add messaging:

```
eol' = try (string "\n\r")
      <|> try (string "\r\n")
      <|> string "\n"
      <|> string "\r"
      <|> fail "Couldn't find EOL"
```

This adds messaging to the result, but is still noisy and unclear:

```
> parseCSV "line1"
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r"
Couldn't find EOL
```

The Parsec `<?>` operator is designed to help here.

It is similar to `<|>` in that it first tries the parser on its left. Instead of trying another parser in the event of a failure, it presents an error message. Here's how we'd use it:

```
eol = try (string "\n\r")
      <|> try (string "\r\n")
      <|> string "\n"
      <|> string "\r"
      <?> "end of line"
```

This has a more pleasing result:

```
> parseCSV "line1"
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting "," or end of line
```

The general rule of thumb is that you put a human description of what you're looking for to the right of `<?>`.

```
parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

## Chapter 5

# Using HUnit to test the Parser

### 5.1 Helper Functions

```
module SpecHelpers where
import Test.HUnit

import Text.Printf

import Data.Time.Clock
import Data.Time.Calendar
import Data.Time.Format

import Text.Parsec
import Text.Parsec.String
```

#### 5.1.1 Date Helpers

Create a date-parser via partial application of `parseTime*`:

```
acceptOuterWhitespace = True

parseDateF :: String → String → Either String Day
parseDateF fmt s =
  case parseTimeM acceptOuterWhitespace defaultTimeLocale fmt s :: Maybe UTCTime of
    Nothing → Left $ printf "Failed to parse '%s' with '%s'" s fmt
    (Just x) → Right $ utctDay x

parseTimeF :: String → String → Either String UTCTime
parseTimeF fmt s =
  case parseTimeM acceptOuterWhitespace defaultTimeLocale fmt s :: Maybe UTCTime of
    Nothing → Left $ printf "Failed to parse '%s' with '%s'" s fmt
    (Just x) → Right x

showTime :: UTCTime → String
showTime = formatTime defaultTimeLocale "%e %b %Y %R"
```

#### 5.1.2 Parser Testing Helpers

```

nullState = ()
testParse :: Parser a → String → String → Either ParseError a
testParse p src = runParser p nullState src

-- assertParsesTo expected p input = do
--   let result = testParse p input
--   case result of
--     (Right actual) → expected @=? show actual
--     (Left error)   → assertFailure
--     (printf "Failed to parse input '%s': %s" input (show error))

```

A Type for expressing parser assertions

```

data ParserContext a =
  ParserContext { parser    :: Parser a
                 , input    :: String
                 , expected :: String
                 }

parses :: (Show a) ⇒ ParserContext a → Assertion
parses pa =
  case testParse (parser pa) ("expected:␣" ++ (show expected pa)) (input pa) of
    (Right actual) → assertEquals "" (show expected pa) (show actual)
    (Left error)   → assertFailure (show error)

instance (Show a) ⇒ Testable (ParserContext a)
  where
    test = TestCase ∘ parses

assertParser message expected parser input =
  parses pa
  where
    pa = ParserContext parser input expected

```

### 5.1.3 Writing the Tests

Table 5.1: Test Types

<code>recordTests :: Test</code>	
<code>test :: Testable t ⇒ t → Test</code>	Provides a way to convert data into a <i>Test</i> or set of <i>Test</i> .
<code>(~:) :: Testable t ⇒ String → t → Test</code>	Creates a test from the specified <i>Testable</i> , with the specified label attached to it. Since <i>Test</i> is <i>Testable</i> , this can be used as a shorthand way of attaching a <i>STestLabel</i> to one or more tests.
<code>assertEquals :: (Show a, Eq a) ⇒ String → a → a → Assertion</code>	
<code>(@=? :: (Show a, Eq a) ⇒ a → a → Assertion</code>	Asserts that the specified actual value is equal to the expected value (with the actual value on the left-hand side).

### 5.1.4 The Preamble

Declare the test module and export its tests.

```

module EmacsDiary.RecordSpec (recordTests) where

import Test.HUnit
import SpecHelpers

import EmacsDiary.Record

recordTests = test [
  "create_a_DiaryEntry" ~: do
    let (Right d) = parseTimeF "%e_%b_%Y" "7_July_2008"
    "7_July_2008_Chicago,_IL" @=? show (Entry d [Location "Chicago,_IL"])
]

```

## 5.2 Testing the Parser

```

module EmacsDiary.ParserSpec (tests) where

import Test.HUnit
import SpecHelpers

import Text.Parsec (runParser, many)
import Text.Printf (printf)

import EmacsDiary.Record
import qualified EmacsDiary.Interval as I
import EmacsDiary.Parser (diary, record, entry)

assertParsesTo p input expected =
  case runParser p () input of
    (Left e)  → assertFailure $ show e
    (Right actual) → assertEquals ""
      expected
      actual

tests = test [
  "parse_a_solitary_entry" ~: do
    let input = "\n14:30entry\n"
    assertParsesTo entry input
      (Entry (I.timeFromList [14, 30]) [Description "entry"])
  ,
  "parse_multi-element_entry" ~: do
    let input = unlines [
      "\n14:30field1",
      "\n\nfield2",
      "\n\n\nfield3"
    ]
    assertParsesTo entry input
      (Entry (I.timeFromList [14, 30])
        [Description "field1",
         Description "field2",
         Description "field3"])
  ,

```

```

"requires_leading_whitespace_for_entry" ~: do
  let input = unlines [
    "  14:30 field1; field2",
    "OtherText"
  ]
  assertParsesTo entry input
  (Entry (I.timeFromList [14, 30])
   [Description "field1",
    Description "field2"])
,

"entry_with_combinator" ~: do
  let input = unlines [
    "  14:30 field1",
    "  15:30 field2"]
  assertParsesTo (many entry) input
  [(Entry (I.timeFromList [14, 30]) [Description "field1"]),
   (Entry (I.timeFromList [15, 30]) [Description "field2"])]
,

"parse_semicolon_separator" ~: do
  let input = unlines [
    "  14:30 field1; field2",
    "    field3"
  ]
  assertParsesTo entry input
  (Entry (I.timeFromList [14, 30])
   [Description "field1",
    Description "field2",
    Description "field3"])
-- ,

-- "parse date and time with empty description produces error" ~: do
--   let input = "7 July 2008 14:30\n"
--   case runParser diary () input input of
--     (Left e)  → assert True
--     (Right r) → assertFailure
--       (printf "parsing should have failed, but got '%s'" (show r))
-- ,

-- "record parsing returns a Record" ~: do
--   let input = "7 July 2008\n"
--   assertParsesTo diary input [(Record (I.fromDmy 7 7 2008) [])]
-- ,

-- "record parser" ~: do
--   let input = unlines ["7 July 2008 14:30 Work on parsers"]
--   assertParsesTo record input
--     (Record (I.fromDmy 7 7 2008)
--      [Entry (I.timeFromList [14, 30])
--       [Description "Work on parsers"]])
-- ,

-- "parse date and time with description" ~: do
--   let input = unlines ["7 July 2008 14:30 Work on parsers"]
--   assertParsesTo diary input

```

```

--      [(Record (I.fromDmy 7 7 2008)
--              [Entry (I.timeFromList [14, 30])
--                [Description "Work on parsers"]])]
--    ,

-- "parse multi-line entry" ~: do
--   let input = unlines [
--     "7 July 2008 14:30 Work on parsers",
--     "  With gusto!",
--     ""
--   ]
--   case runParser diary 0 input input of
--     (Left e)      → assertFailure $ show e
--     (Right actual) → assertEquals "Multi-line entry"
--       [(Record (I.fromDmy 7 7 2008)
--               [Entry (I.timeFromList [14, 30])
--                 [Description "Work on parsers",
--                   Description "With gusto!"]])]
--       actual
--   ,

-- "parse multiple records" ~: do
--   let input = unlines [
--     "7 July 2008 14:30 Work on parsers",
--     "7 July 2008 15:15 Celebrate"]
--   case runParser diary 0 input input of
--     (Left e)      → assertFailure $ show e
--     (Right actual) → assertEquals "Multiple records"
--       [(Record (I.fromDmy 7 7 2008)
--               [Entry (I.timeFromList [14, 30])
--                 [Description "Work on parsers"]]),
--        (Record (I.fromDmy 7 7 2008)
--               [Entry (I.timeFromList [15, 15])
--                 [Description "Celebrate"]])]
--       actual
-- ]

```



## Chapter 6

# The Emacs Diary Parser

### 6.1 Parsing Dates and Times

```
module EmacsDiary.Parser.Interval where

import qualified EmacsDiary.Parser.Tokens as T
import EmacsDiary.Interval

import Data.Functor.Identity

import Text.Parsec (Column, Parsec, ParsecT, Stream, (<?>),
  many, string, count, spaces, digit, many1, choice, (<|>), try, sepBy1, unexpected)
import Text.Parsec.String (Parser)
import Data.Time.Calendar (fromGregorian)
```

#### 6.1.1 Date

```
--dayP :: Parser Int
dayP = fromIntegral <$> T.numeric

--yearP :: Parser Integer
yearP = T.numeric

months = [
  ("January", 1), ("Jan", 1),
  ("February", 2), ("Feb", 2),
  ("March", 3), ("Mar", 3),
  ("April", 4), ("Apr", 4),
  ("May", 5),
  ("June", 6), ("Jun", 6),
  ("July", 7), ("Jul", 7),
  ("August", 8), ("Aug", 8),
  ("September", 9), ("Sep", 9),
  ("October", 10), ("Oct", 10),
  ("November", 11), ("Nov", 11),
  ("December", 12), ("Dec", 12)
]

monthP :: Parser Int
monthP = T.lexeme $ choice $
```

```

map kvp months
where
  kvp :: (String, Int) → Parser Int
  kvp (mn,n) = try (string mn) >> return n

date :: Parser Date
date = fromDmy <$> dayP <*> monthP <*> yearP <?> "date"

```

### 6.1.2 Time

```

time :: Parser Time
time = timeFromList <$> (T.whitespace *> hms) <?> "time"
where
  -- hms :: Parser [Integer]
  hms = sepBy1 T.numeric (T.symbol ":") <?> "hours:mins:secs"

```