

Creating an Emacs Diary Parser with Haskell and Parsec

Toby Tripp

March 2018

Contents

| | | |
|----------|---|-----------|
| 1 | “Don’t Fear the Monad” | 2 |
| 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 |
| 2 | The Maybe Monad | 5 |
| 2.1 | As Described by Computerphile | 5 |
| 3 | Other Monads | 6 |
| 3.1 | The Identity Monad | 6 |
| 3.2 | Definition of Maybe | 6 |
| 3.3 | The List <i>Monad</i> | 6 |
| 4 | Using Parsec | 8 |
| 4.1 | Parsing CSV | 8 |
| 4.2 | sepBy and endBy Combinators | 9 |
| 5 | Using HUnit to test the Parser | 11 |
| 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 |
| 6 | The Emacs Diary Parser | 17 |
| 6.1 | Parsing Dates and Times | 17 |
| 6.1.1 | Date | 18 |
| 6.1.2 | Time | 19 |
| 6.1.3 | Interval | 19 |
| 6.2 | Testing the Parser | 19 |

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 [1]. 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.
^a:

^aWikipedia

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:

```
(>>=) :: Monad m => m a -> (a -> m b) -> 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*.

`(>>=)` 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:

```
return :: Monad m => a -> m a
```

Extended to non-uniform types:

```
g :: a -> Mb
f :: b -> Mc
λa -> (g a) >>= λb -> (f b) :: a -> Mc
g >>= λb -> (f b)
```

Chapter 2

The Maybe Monad

2.1 As Described by Computerphile

[3]

```
-- 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 >>= (λn → eval' y >>= (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 [4, p. 402]:

```
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. [4, p. 403]

3.1 The Identity Monad

The identity monad [4, p. 404] 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
```

[2, Ch. 16]

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 returns 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 Data.Time.LocalTime

import Text.Parsec
import Text.Parsec.String
import Data.Time.LocalTime (utc)

import qualified EmacsDiary.Interval as Interval
```

5.1.1 Date Helpers

TimeZones

```
cdt = hoursToTimeZone (-5)
```

5.1.2 Parser Testing Helpers

```
assertParsesTo :: (Eq expected, Show expected)
  => Parsec SourceName () expected -- ^ Parser under test
  -> String                        -- ^ input
  -> expected                      -- ^ expected parser output
  -> IO ()

assertParsesTo p input expected =
  case runParser p nullState input input of
    (Left e) -> assertFailure $ show e
    (Right actual) -> assertEquals "" expected actual
  where
    nullState = ()
```

5.1.3 Writing the Tests

Table 5.1: Test Types

| | |
|---|---|
| <code>tests :: Test</code> | Provides a way to convert data into a <code>Test</code> or set of <code>Test</code> . |
| <code>test :: Testable t => t -> Test</code> | Creates a test from the specified <code>Testable</code> , with the specified label attached to it. |
| <code>(~:) :: Testable t => String -> t -> Test</code> | Since <code>Test</code> is <code>Testable</code> , this can be used as a shorthand way of attaching a <code>STestLabel</code> to one or more tests. |
| <code>assertEqual :: (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 (tests) where

import Test.HUnit
import SpecHelpers

import Data.Time.Calendar
import Data.Time.LocalTime

import EmacsDiary.Record
import qualified EmacsDiary.Interval as I
import qualified EmacsDiary.Ics as ICS
```

```
tstamp = ZonedTime local cdt
  where
    local = LocalTime d t
    d = fromGregorian 2008 07 07
    t = TimeOfDay 12 00 00

tests = test [
  "emit_a_diary_record_as_ics" ~: do
    let expected = unlines [
      "BEGIN:VCALENDAR"
      , "VERSION:2.0"
      , "BEGIN:VEVENT"
      , "UID:ED20080707170000"
      , "CREATED:20080707T170000Z"
      , "DTSTART:20080707T170000Z"
      , "DTEND:20080707T170000Z"
      , "SUMMARY:Happy Birthday, Son!"
      , "END:VEVENT"
      , ""
      , "BEGIN:VEVENT"
      , "UID:ED20080707180000"
      , "CREATED:20080707T170000Z"
      , "DTSTART:20080707T180000Z"
      , "DTEND:20080707T190000Z"
```

```

    , "SUMMARY: Eat_Cake."
    , "END:VEVENT"
    , ""
    , ""
    , "END:VCALENDAR"
  ]
let d = I.utcDate tstamp (I.gregorian 2008 7 7)
let e1 = Entry (I.instant d 12 00) [Description "Happy_Birthday,_Son!"]
let e2 = Entry (I.interval (I.makeTime d 13 00)
                      (Just (I.makeTime d 14 00)))
                      [Description "Eat_Cake."]
let input = Diary [
  push e1 $ push e2 (empty d)
]

assertEqual "" expected (ICS.toIcs input)
,

"repeating_events_as_ICS" ~: do
let expected = unlines [
  "BEGIN:VCALENDAR"
  , "VERSION:2.0"
  , "BEGIN:VEVENT"
  , "UID:ED20080709193000"
  , "CREATED:20080707T170000Z"
  , "DTSTART:20080709T193000Z"
  , "DTEND:20080709T193000Z"
  , "RRULE:WEEKLY"
  , "SUMMARY:Coffee"
  , "END:VEVENT"
  , ""
  , ""
  , "END:VCALENDAR"
]
let d = I.DayOfWeek I.Wednesday tstamp
let e = Entry (I.instant d 14 30) [Description "Coffee"]
let input = Diary [push e (empty d)]
assertEqual "" expected (ICS.toIcs input)
]

```

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 Data.Time.LocalTime (hoursToTimeZone)

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

```

Create a parsed-at time-stamp value for use in testing parsers.

```
import Data.Time.Calendar
import Data.Time.LocalTime

tstamp = ZonedTime local cdt
  where
    local = LocalTime d t
    d = fromGregorian 2008 07 07
    t = TimeOfDay 12 0 0
```

A sample *Record*

```
epochRecord = makeRecord (I.gregorian 1970 1 1)
makeRecord = R.empty ◦ (I.utcDate tstamp)
```

Parsers to test:

```
diaryParser = diary tstamp
recordParser = record tstamp
entryParser = entry epochRecord
```

```
tests = test [
  "parse_a_solitary_entry" ~: do
    let input = "\u0014:30\u00entry\u00a"
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["entry"])
  ,
  "parse_multi-element_entry" ~: do
    let input = unlines [
      "\u0014:30\u00field1",
      "\u00\u00field2",
      "\u00\u00field3"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2", "field3"])
  ,
  "requires_leading_whitespace_for_entry" ~: do
    let input = unlines [
      "\u0014:30\u00field1;\u00field2",
      "Other\u00Text"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2"])
  ,
  "entry_with_combinator" ~: do
    let input = unlines [
      "\u0014:30\u00field1",
      "\u0015:30\u00field2"
    ]
    assertParsesTo (many entryParser) input
      [(R.entry (I.instant I.epoch 19 30) ["field1"]),
       (R.entry (I.instant I.epoch 20 30) ["field2"])]
  ,
  "parse_semicolon_separator" ~: do
    let input = unlines [
      "\u0014:30\u00field1;\u00field2",
```

```

        "field3"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2", "field3"])
  ,

  "parse_date_and_time_with_empty_description_produces_error" ~: do
    let input = "7_July_2008_14:30\n"
    case runParser diaryParser () 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 diaryParser input (R.Diary [makeRecord (I.gregorian 2008 7 7)])
  ,

  "record_parser" ~: do
    let input = unlines ["7_July_2008_14:30_Work_on_parsers"]
    let d = I.utcDate tstamp (I.gregorian 2008 7 7)
    assertParsesTo recordParser input
      (R.push (R.entry (I.instant d 14 30) ["Work_on_parsers"])
        (R.empty d))
  ,

  "parse_date_and_time_with_description" ~: do
    let input = unlines ["7_July_2008_14:30_Work_on_parsers"]
    let d = I.utcDate tstamp (I.gregorian 2008 7 7)
    let e = R.entry (I.instant d 14 30) ["Work_on_parsers"]
    assertParsesTo diaryParser input (R.Diary [R.push e (R.empty d)])
  ,

  "parse_multi-line_entry" ~: do
    let input = unlines [
      "7_July_2008_14:30_Work_on_parsers",
      "With_gusto!",
      ""
    ]
    let d = I.utcDate tstamp (I.gregorian 2008 7 7)
    let e = R.entry (I.instant d 14 30) ["Work_on_parsers", "With_gusto!"]
    case runParser diaryParser () input input of
      (Left e)      → assertFailure $ show e
      (Right actual) → assertEquals "Multi-line_entry"
        (R.Diary [R.push e $ makeRecord (I.gregorian 2008 7 7)])
        actual
  ,

```

In the Emacs Diary, weekly repeating events can be specified by using a week-day name instead of a specific date.

```

"weekday_entries" ~: do
  let input = "Wednesday_08:00_Wake_up"
  let e = R.entry (I.instant (I.utcDate tstamp (I.gregorian 2008 07 09)) 8 0) ["Wake_up"]
  assertParsesTo diaryParser input (R.Diary [
    R.push e (R.Record (I.DayOfWeek I.Wednesday tstamp) [])])
  ,

```



```

"parse_entry_with_time-interval" ~: do
  let input = unlines [
    "7_July_2008_13:00-16:00_Gorge_on_Cake"
  ]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let t1 = I.makeTime d 13 0
  let t2 = I.makeTime d 16 0
  let e1 = R.entry (I.interval t1 (Just t2)) ["Gorge_on_Cake"]
  assertParsesTo diaryParser input
  (R.Diary [R.push e1 (makeRecord (I.gregorian 2008 7 7))])
  ,

"parse_multiple_records" ~: do
  let input = unlines [
    "7_July_2008_14:30_Work_on_parsers",
    "7_July_2008_15:15_Celebrate"
  ]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let e1 = R.entry (I.instant d 14 30) ["Work_on_parsers"]
  let e2 = R.entry (I.instant d 15 15) ["Celebrate"]
  case runParser diaryParser () input input of
    (Left e)      → assertFailure $ show e
    (Right actual) → assertEquals "Multiple_records"
      (R.Diary [(R.push e1 $ makeRecord (I.gregorian 2008 7 7)),
        (R.push e2 $ makeRecord (I.gregorian 2008 7 7))])
      actual
]

```

Chapter 6

The Emacs Diary Parser

6.1 Parsing Dates and Times

```
{-|
Description: Parsers for calendar time-intervals.
-}

module EmacsDiary.Parser.Interval (
  interval,
  date,
  time
)where

import qualified EmacsDiary.Parser.Tokens as T
import EmacsDiary.Interval (
  Date(..),
  WeekDay(..),
  Time,
  Interval(..),

  utcDate,
  makeTime,
  gregorian
)

import Text.Parsec (
  (<|>),           -- ^ 'choice' operator
  (<?>),           -- ^ 'unexpected' operator

  choice,
  option,
  string,
  try,
  unexpected
)

import Text.Parsec.String (Parser)
import Data.Time.Calendar (fromGregorian)
import Data.Time.LocalTime (
  ZonedTime,
  zonedTimeZone
)
```

```

date      :: ZonedTime          -- ^ current local time
           → Parser Date
time      :: Date → Parser Time
interval  :: Date → Parser Interval

```

6.1.1 Date

```

date localtime = day localtime <|> weekday localtime

day localt = do
  d ← dayP
  m ← monthP
  y ← yearP <?> "date"
  return $ utcDate localt (gregorian y m d)
weekday localt = T.lexeme $ choice $
  map kvp weekdays
  where
    kvp :: (String, WeekDay) → Parser Date
    kvp (wstring, wd) = try (string wstring) >> return (DayOfWeek wd localt)
    weekdays = [
      ("Sunday",    Sunday)
    , ("Monday",   Monday)
    , ("Tuesday",  Tuesday)
    , ("Wednesday", Wednesday)
    , ("Thursday", Thursday)
    , ("Friday",   Friday)
    , ("Saturday", Saturday)
    ]

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

yearP :: Parser Integer
yearP = T.numeric

monthP :: Parser Int
monthP = T.lexeme $ choice $
  map kvp months
  where
    kvp :: (String, Int) → Parser Int
    kvp (mn,n) = try (string mn) >> return n

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)
]

```

6.1.2 Time

```
time d = do
  h ← T.whitespace *> try (T.numeric <* T.symbol ":") <|> unexpected "time"
  m ← T.numeric <|> unexpected "time"
  return $ makeTime d (fromInteger h) (fromInteger m)
```

6.1.3 Interval

An `Interval` is a range of times on a specified `Date`. The Emacs Diary, so far as I know, does not support events spanning more than one day.

```
interval d = do
  t1 ← try (time d) <|> unexpected "start_time"
  t2 ← option t1 (T.symbol "-" *> (time d))
  return $ Interval t1 t2
```

6.2 Testing the Parser

```
module EmacsDiary.ParserSpec (tests) where

import Test.HUnit
import SpecHelpers

import Text.Parsec (runParser, many)
import Text.Printf (printf)
import Data.Time.LocalTime (hoursToTimeZone)

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

Create a parsed-at time-stamp value for use in testing parsers.

```
import Data.Time.Calendar
import Data.Time.LocalTime

tstamp = ZonedTime local cdt
  where
    local = LocalTime d t
    d = fromGregorian 2008 07 07
    t = TimeOfDay 12 0 0
```

A sample `Record`

```
epochRecord = makeRecord (I.gregorian 1970 1 1)
makeRecord = R.empty ∘ (I.utcDate tstamp)
```

Parsers to test:

```
diaryParser = diary tstamp
recordParser = record tstamp
entryParser = entry epochRecord
```

```

tests = test [
  "parse_a_solitary_entry" ~: do
    let input = "\14:30entry\n"
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["entry"])
  ,
  "parse_multi-element_entry" ~: do
    let input = unlines [
      "\14:30field1",
      "\\field2",
      "\\field3"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2", "field3"])
  ,
  "requires_leading_whitespace_for_entry" ~: do
    let input = unlines [
      "\14:30field1;\field2",
      "OtherText"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2"])
  ,
  "entry_with_combinator" ~: do
    let input = unlines [
      "\14:30field1",
      "\15:30field2"
    ]
    assertParsesTo (many entryParser) input
      [(R.entry (I.instant I.epoch 19 30) ["field1"]),
       (R.entry (I.instant I.epoch 20 30) ["field2"])]
  ,
  "parse_semicolon_separator" ~: do
    let input = unlines [
      "\14:30field1;\field2",
      "\\field3"
    ]
    assertParsesTo entryParser input
      (R.entry (I.instant I.epoch 19 30) ["field1", "field2", "field3"])
  ,
  "parse_date_and_time_with_empty_description_produces_error" ~: do
    let input = "7July2008\14:30\n"
    case runParser diaryParser () 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 = "7July2008\n"
    assertParsesTo diaryParser input (R.Diary [makeRecord (I.gregorian 2008 7 7)])
  ,

```

```

"record_parser" ~: do
  let input = unlines ["7_July_2008_14:30_Work_on_parsers"]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  assertParsesTo recordParser input
    (R.push (R.entry (I.instant d 14 30) ["Work_on_parsers"])
     (R.empty d))
,

"parse_date_and_time_with_description" ~: do
  let input = unlines ["7_July_2008_14:30_Work_on_parsers"]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let e = R.entry (I.instant d 14 30) ["Work_on_parsers"]
  assertParsesTo diaryParser input (R.Diary [R.push e (R.empty d)])
,

"parse_multi-line_entry" ~: do
  let input = unlines [
    "7_July_2008_14:30_Work_on_parsers",
    "    With_gusto!",
    ""
  ]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let e = R.entry (I.instant d 14 30) ["Work_on_parsers", "With_gusto!"]
  case runParser diaryParser () input input of
    (Left e)      → assertFailure $ show e
    (Right actual) → assertEquals "Multi-line_entry"
      (R.Diary [R.push e $ makeRecord (I.gregorian 2008 7 7)])
      actual
,

```

In the Emacs Diary, weekly repeating events can be specified by using a week-day name instead of a specific date.

```

"weekday_entries" ~: do
  let input = "Wednesday_08:00_Wake_up"
  let e = R.entry (I.instant (I.utcDate tstamp (I.gregorian 2008 07 09)) 8 0) ["Wake_up"]
  assertParsesTo diaryParser input (R.Diary [
    R.push e (R.Record (I.DayOfWeek I.Wednesday tstamp) [])])
,

```

```

"parse_entry_with_time-interval" ~: do
  let input = unlines [
    "7_July_2008_13:00-16:00_Gorge_on_Cake"
  ]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let t1 = I.makeTime d 13 0
  let t2 = I.makeTime d 16 0
  let e1 = R.entry (I.interval t1 (Just t2)) ["Gorge_on_Cake"]
  assertParsesTo diaryParser input
    (R.Diary [R.push e1 (makeRecord (I.gregorian 2008 7 7))])
,

"parse_multiple_records" ~: do
  let input = unlines [
    "7_July_2008_14:30_Work_on_parsers",
    "7_July_2008_15:15_Celebrate"
  ]
  let d = I.utcDate tstamp (I.gregorian 2008 7 7)
  let e1 = R.entry (I.instant d 14 30) ["Work_on_parsers"]
  let e2 = R.entry (I.instant d 15 15) ["Celebrate"]

```

```

case runParser diaryParser () input input of
  (Left e)      → assertFailure $ show e
  (Right actual) → assertEquals "Multiple records"
    (R.Diary [(R.push e1 $ makeRecord (I.gregorian 2008 7 7)),
               (R.push e2 $ makeRecord (I.gregorian 2008 7 7))])
    actual
]

```

Bibliography

- [1] Dr. Brian Beckman. *Don't Fear the Monad*. YouTube. Nov. 2012. URL: <https://youtu.be/ZhuHCtR3xq8> (visited on 04/2018).
- [2] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, 2008. URL: <http://book.realworldhaskell.org/read/using-parsec.html>.
- [3] Sean Riley and Professor Graham Hutton. *Computerphile: What is a Monad?* YouTube. Nov. 2017. URL: <https://youtu.be/t1e8gqXLbsU> (visited on 04/2018).
- [4] Simon Thompson. *The Craft of Functional Programming*. Second. International Computer Science Series. Edinburgh Gate, England: Pearson Education Limited, 1999.