Creating an Emacs Diary Parser with Haskell and Parsec

Toby Tripp

March 2018

Contents

Chapter 1

"Brian Beckman: Don't Fear the Monad"

Available on Youtube [2].

Dr. Beckman, astrophysicist and senior software engineer, begins with a basic introduction to functional programming as a concept. 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, ∘, to imply composition: `(f ∘ g) a`. So, given the above1.2.1, we can deduce:

```
h = (f ∘ g) a = f ∘ 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]:

[a]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: $\lambda a \to [(fa) >>= \lambda a \to (ga)]$ 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` → `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 [1]

```
-- Type and 2 type constuctors: 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[3, 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. [3, p. 403]

3.1 The Identity Monad [3, p. 404]

The identity monad 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*

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

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

```haskell
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

Parsec

4.1 Using Parsec [4, Ch. 16]

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

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")
```

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

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

Bibliography

[1] Sean Riley and Professor Graham Hutton, *Computerphile: What is a Monad?*, YouTube, November 2017.

[2] Dr. Brian Beckman, *Don't Fear the Monad*, YouTube, November 2012.

[3] Simon Thompson, *The Craft of Functional Programming*, Pearson, Edinburgh Gate, England, Second Edition, 1999.

[4] Bryan O'Sullivan, Don Stewart, and John Goerzen *Real World Haskell* O'Reilly Media 2008.