CSCI400 – 11/30 Handout

Recap

In the lecture, we constructed our own *parsers* and *parser combinators*, where 'parser combinators' basically means 'functions that can be used to combine parsers'. In the lecture, our combinators were `andThen` and `stuff`, which we then related to the functions in the `Monad` typeclass: `(>>=)` ('bind') and `return`.

```
-- same purpose as >>=
andThen :: Parser s a -> (a -> Parser s b) -> Parser s b
andThen parse next = \input ->
  case parse input of
    Nothing          -> Nothing
    Just (a, input') -> next a input'

-- same purpose as return
stuff :: a -> Parser s a
stuff a = \input -> Just (a, input)
```

These functions, `andThen` and `stuff`, allowed us to take a parser that looked like this:

```
versionDumb :: Parser String (Int, Int)
versionDumb = \input ->
  case string "HTTP/" input of
    Nothing -> Nothing
    Just (_,rest1) ->
      case number rest1 of
        Nothing -> Nothing
        Just (maj,rest2) ->
          case string "." rest2 of
            Nothing -> Nothing
            Just (n,rest3) ->
              case number rest3 of
                Nothing -> Nothing
                Just (min,rest4) -> Just ((maj,min),rest4)
```

and instead write it like this:

```
version2 :: Parser String (Int, Int)
version2 =
  string "HTTP/" `andThen` \_ ->
  number         `andThen` \maj ->
  string "."     `andThen` \_ ->
  number         `andThen` \min ->
  stuff (maj,min)
```

While this is certainly better than versionDumb, it's still not that pretty to look at.

do Notation

Let's assume that we'd implemented the Monad typeclass for our Parser type. As a reminder, the Monad typeclass looks like this:

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

This means that we'd have to implement the (>>=) and return functions for our Parser type. We can leverage the functions we've already written here:

```
instance Monad (Parser s) where
  (>>=)  = andThen
  return = stuff
```

The andThen and stuff functions behaved *exactly* as we'd want (>>=) and return to, so we just use those functions in defining the Monad typeclass instance for our parser type.

In doing this, we'd be able to rewrite our version2 parser to use these functions by replacing andThen with >>= and stuff with return:

```
version3 :: Parser String (Int, Int)
version3 =
  string "HTTP/" >>= \_ ->
  number         >>= \maj ->
  string "."     >>= \_ ->
  number         >>= \min ->
  return (maj,min)
```

This is just as ugly though – all we've done so far is rename a couple of functions.

Now we come to something that you may have seen around, but we haven't talked about yet: do notation. do notation is simply *syntactic sugar for expressions that are built using the* Monad *functions* (>>=) *and* return. We can use do notation to rewrite version3 as:

```
version4 :: Parser String (Int, Int)
version4 = do
  string "HTTP/"
  maj <- number
  string "."
  min <- number
  return (maj, min)
```

Note that, as far as the compiler is concerned, `version4` is *exactly the same* as `version3`. The compiler would simply de-sugar `version4` to `version3`, because `do` notation is just syntactic sugar.

You can use `do` notation for any function that returns a type that is a part of the `Monad` typeclass, which our parser type is now. This provides a means of using the `Monad` design pattern without the ugly syntax.

Applicative Parsing

`do` notation gives us one way to clean up our parser definitions. The thing about `do`, though, is that it's 1. fairly straightforward to use, but 2. it's difficult to really understand what it's doing.

So let's approach this from a different angle. Let's write our parsers using a few other typeclasses, namely `Functor`, which gives us the `(<$>)` ('fmap') operator, and `Applicative`, which gives us the `(<*>)` ('apply') operator and a few others (namely, `(*>)` and `(<*)`). We'll also see the `Alternative` typeclass, which gives us `(<|>)` – you'll get an idea of how this operator works from the chapter reading.

This style will allow us to take parsers that look like this:

```haskell
majorVersion :: Parser String Int
majorVersion = do
  string "HTTP/"
  maj <- number
  return maj
```

And rewrite them like this:

```haskell
majorVersion :: Parser String Int
majorVersion = string "HTTP/" *> number
```

You might think one of this is cleaner than the other, but the latter form is the one we'll be using on the second half of the project.

From here, you should read the `Parsec` chapter of Real World Haskell. `Parsec` is a parsing library in Haskell. The one we'll actually be using is `Megaparsec`, but they're similar enough that the book chapter should give you a solid start. *Note that the chapter starts off by using* `do` *notation to define parsers, then moves to the notation that we want to use (called 'applicative Parsec').* However, it explains applicative Parsec using `do` notation, so it's helpful to understand both.