Pada pengerjaan Parser Combinator ini, saya mempelajari hal yang paling rumit selama pembelajaran pemrograman fungsional, namun menurut saya juga merupakan salah satu hal yang paling menarik, jika tidak, hal yang paling menarik dan juga konkret karena mengaplikasikan semua yang telah dipelajari selama setengah semester ini dalam sebuah program parser yang sederhana, namun cukup berat konsepnya.

Saya ingin memberi outline hal-hal yang penting yang telah saya pelajari.

1. newtype Berikut adalah penggunaan newtype pada parser combinator

```
newtype Parser a = Parser { parse :: String -> Maybe (String, a)}
```

newtype ini hanya memiliki 1 konstruktor Cara membacanya adalah saya membuat type bernama Parser. Namun kenapa ada fungsi parse di dalamnya? Ini cara bacanya adalah untuk meng unwrap parser, makanya memiliki return value Maybe (String, a)

```
*Main Control.Applicative> :t parse
parse :: Parser a -> String -> Maybe (String, a)
*Main Control.Applicative> :t Parser
Parser :: (String -> Maybe (String, a)) -> Parser a
```

Sebalknya, tipe parser ini menerima fungsi dengan tipe String -> Maybe (String, a) dan mengembalikan sesuatu bertipe Parser.

2. <$> Functor

```
*Main Control.Applicative> :t (<$>)
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Setelah melihat definisinya, terlihat bahwa <$> menerima fungsi dan menerapkan fungsi tersebut ke dalam elem-elemen f.

```
*Main Control.Applicative> :t map
map :: (a -> b) -> [a] -> [b]
```

Wah mirip ya fmap dengan map, iya, ternyata map ini fmap yang khusus dilakukan kepada list, fmap lebih general.

fmap ini lalu digunakan untuk membuat Parser kita menjadi sebuah instance dari Functor!

```
*Main Control.Applicative> :info Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

Untuk membuat Parser menjadi instance dari Functor, minimal dibutuhkan fungsi `fmap`, yang definisinya sama persis dengan fmap atau <$> yang sudah ditunjukkan di atas. Berikut adalah penerapan fmap untuk parser.

```
instance Functor Parser where
  fmap f (Parser p) = Parser $ \inp -> do
    (input', x) <- p inp
    Just (input', f x)
```

Seperti yang sudah didefine pada newtype, Parser menerima fungsi String -> Maybe(String, a). f disini sebenarnya sedikit membingungkan, karena sebenarnya f ini berupa tipe, seperti list atau nanti yang akan diimplemen yaitu JsonValue. Fmap ini dapat dilihat sebagai perlakuan fungsi <$> terhadap elemen-elemen dalam suatu tipe.

Keren, Parser kita sudah merupakan sebuah Functor, lalu apa yang bisa dilakukan? Pertama-tama, kita buat dahulu `charParser`, yaitu parser untuk char

```
charParser :: Char -> Parser Char
charParser c = Parser f
  where
    f (x : xs)
      | c == x = Just (xs, c)
      | otherwise = Nothing
    f [] = Nothing

*Main Control.Applicative Data.Char> :t charParser 'j'
charParser 'j' :: Parser Char

*Main Control.Applicative> import Data.Char
*Main Control.Applicative Data.Char> :t fmap ord (charParser 'j')
fmap ord (charParser 'j') :: Parser Int

*Main Control.Applicative Data.Char> parse (charParser 'j') "joni"
Just ("oni",'j')
```

Sekarang, isi dari Parser dapat diubah dengan fungsi melalui fmap.

3. <*> Applicative

```
*Main Control.Applicative Data.Char> :info Applicative
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
```

```
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Agar Parser kita dapat menjadi instance sebuah Applicative, dibutuhkan fungsi `pure` dan juga `<*>` atau liftA2. Saya gatau liftA2, namun untuk <> *akan digunakan dalam parsec ini, sehingga* <> dan pure sudah cukup.

```
*Main Control.Applicative Data.Char> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Mirip yah dengan fmap, bedanya hanya pada inputnya, yaitu ini f (a -> b) sedangkan fmap (a -> b). Setelah coba coba, basically <*> ini berguna karena sifat haskell yang menerapkan `currying`.

kembali ke contoh fmap pada charParser.

```
*Main Control.Applicative Data.Char> :t fmap ((+2) . ord) (charParser 'j')
fmap ((+2) . ord) (charParser 'j') :: Parser Int
```

Dapat dilakukan! Namun bagaimana jika fungsinya adalah (+) saja, instead of (+2). Berarti fungsi dalam fmap ini masih butuh 1 argumen berupa Int kan?

```
*Main Control.Applicative Data.Char> :t fmap ((+) . ord) (charParser 'j')
fmap ((+) . ord) (charParser 'j') :: Parser (Int -> Int)
```

karena belum ada parser int, saya menggunakan contoh dengan just

```
*Main Control.Applicative Data.Char> :t (+) <$> Just 6
(+) <$> Just 6 :: Num a => Maybe (a -> a)
*Main Control.Applicative Data.Char> :t (+) <$> Just 6 <*> Just 5
(+) <$> Just 6 <*> Just 5 :: Num b => Maybe b -- bukan fungsi lagi
```

Berikut adalah definisi fungsi pure dan <*> Parser.

```
instance Applicative Parser where
  pure x = Parser $ \inp -> Just (inp, x)

  -- take the input, put it through the first parser, then the second
  (Parser p1) <*> (Parser p2) = Parser $ \inp -> do
```

```
    -- currying
    (inp1, f) <- p1 inp
    (inp2, a) <- p2 inp1
    Just (inp2, f a)
```

Dapat dilihat juga bahwa <*> memasukkan input pada parser pertama, lalu parser ke dua, seperti yang saya jelaskan di atas dimana ini menggunakan konsep `currying`. Pure dari definisinya yaitu a -> f a, digunakan untuk sebagai safety ketika sesuatu, pada konteks parsec ini, bukan sebuah Parser, sehingga diubah penjadi Parser a terlebih dahulu . https://stackoverflow.com/questions/51512233/what-is-the-purpose-of-pure-in-applicative-functor

4. sequenceA dan traverse Sekarang, sudah ada charParser, namun input kita akan berupa string.

```
*Main Control.Applicative Data.Char> :t map charParser "bob"
map charParser "bob" :: [Parser Char]
```

Yang diinginkan adalah Parser [Char], terbalik! Ternyata sudah ada fungsi untuk handle kasus seperti ini.

```
*Main Control.Applicative Data.Char> :t sequenceA (map charParser "bob")
sequenceA (map charParser "bob") :: Parser [Char]
```

even better

```
*Main Control.Applicative Data.Char> :t traverse charParser "bob"
traverse charParser "bob" :: Parser [Char]
```

Traverse ini saya julukkan sebagai inside-out operator. Sekarang kita sudah memiliki stringParser!

```
stringParser :: String -> Parser String
stringParser = traverse charParse
```

implementasi jsonNull

```
jsonNull :: Parser JsonValue
jsonNull = (\_ -> JsonNull) <$> stringParser "null"
*Main Control.Applicative Data.Char> parse jsonNull "null"
Just ("",JsonNull)
```

Selanjutnya jsonBool, yaitu stringParser "true" atau stringParser

5. <|> Berikut, untuk menerapkan jsonBool kita akan menerapkan interface `Alternative`.

```
instance Alternative Parser where
  empty = Parser $ \_ -> Nothing
  (Parser p1) <|> (Parser p2) = Parser $ \inp ->
    p1 inp <|> p2 inp

jsonBool :: Parser JsonValue
jsonBool = f <$> (stringParser "true" <|> stringParser "false")
  where
    f inp
      | inp == "true" = JsonBool True
      | inp == "false" = JsonBool False
```

Cukup intuitif, karena hanya cek apakah string merupakan salah satu dari dua value, true atau false.

6. <* dan *> Mirip dengan <*>, namun perbedaannya adalah dilakukan pembuangan elemen tergantung arah.

```
*Main Data.Char> :t (<*)
(<*) :: Applicative f => f a -> f b -> f a
*Main Data.Char> :t (*>)
(*>) :: Applicative f => f a -> f b -> f b
```

Ini sangat berguna untuk penerapan `JsonString` dan `JsonArray`, karena mereka berdua dibatasi oleh suatu hal (quotes dan brackets, respectively).

fungsi `/=` melakukan perbandingan antar Char dan return boolean. Ini merupakan negasi dari fungsi `==`

```
*Main Data.Char> (/='j') 'j'
False
*Main Data.Char> (=='j') 'j'
True
*Main Data.Char> (=='j') 'o'
False
*Main Data.Char> (/='j') 'o'
True
```

Fungsi `/=` cocok digunakan dengan `prefixParser`, yang menggunakan fungsi `span` untuk mengambil char satu per satu sampai predikat fungsi false. Predikat yang diberikan ini adalah fungsi yang menerima Char dan return Bool. Jadi /= digunakan sebagai predikat untuk terus parse string sampai ending karakter ditemukan.

```
prefixParser :: (Char -> Bool) -> Parser String
prefixParser predicate =
  Parser $ \inp -> Just (span predicate inp)
```

Berikut adalah implementasi yang menggunakan prefixParser untuk membuat jsonString.

```
literalParser :: Parser String
literalParser = prefixParser (/= '"')

jsonString :: Parser JsonValue
jsonString = JsonString <$> (charParser '"' *> literalParser <* charParser
'"')
```

7. many selama ini hanya dapat melakukan parsing untuk 1 parser saja, namun pada implementasi JsonArray, dibutuhkan kemampuan untuk parse semua kemungkinan parser pada sebuah array. Ternyata yang dicari adalah fungsi `many`.

```
*Main Data.Char> :info many
type Alternative :: (* -> *) -> Constraint
class Applicative f => Alternative f where
  ...
  many :: f a -> f [a]
        -- Defined in ‘GHC.Base’
```

Prekondisi Applicative dan Alternative sudah dipenuhi oleh Parser, maka dapat langsung dicoba.

```
*Main Data.Char> parse (many jsonValue) "nulltruefalse\"hello\""
Just ([JsonNull,JsonBool True,JsonBool False,JsonString "hello"],"")
```