Refleksi Parser Combinator

Gani Ilham irsyadi (1906350616) - Github Repository

Parser Combinator merupakan sebuah fungsi yang dapat mengonversi sebuah String menjadi data type yang dapat digunakan secara programatis, pada konteks ini adalah bahasa Haskell. Materi **Parser Combinator** menggabungkan semua konsep pemrograman fungsional yang sudah dipelajari selama ini. Meskipun demikian, terdapat pula konsep-konsep baru yang saya pelajari, utamanya ketika mempelajari sendiri contoh Parser Combinator melalui Tugas UTS. Beberapa konsep tersebut adalah:

1. Deklarasi `newtype`

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

Deklarasi `newtype` memiliki kegunaan yang sama dengan `data`, yaitu untuk membuat sebuah type baru. Perbedaanya adalah `newtype` hanya memiliki tepat satu constructor dan field. Apa sebenarnya field `runParser` tersebut?

```
ghci> :t runParser
runParser :: Parser a -> String -> Maybe (String, a)
```

Jika dilihat dari tipenya, `runParser` menerima sebuah value dengan tipe Parser dan akan men-generate sebuah fungsi sesuai dengan tipe Parser tersebut. Contoh terdapat sebuah fungsi jsonValue:

```
jsonValue :: Parser JsonValue
jsonValue = undefined
```

Dihasilkan sebuah fungsi, (type variable `a` berubah menjadi `JsonValue`)

```
ghci> :t runParser jsonValue
runParser jsonValue :: String -> Maybe (String, JsonValue)
```

2. Hole Sebuah hole pada haskell biasa digunakan untuk membantu dalam melengkapi implementasi sebuah fungsi. Hole ditandai dengan `_` pada awal sebuah variable. Compiler akan memberikan error dan memberikan expected type dari hole tersebut. Kemudian kita dapat mengimplementasi tipe tersebut step by step hingga error tidak muncul.
3. Functor dan `fmap` Functor adalah sebuah typeclass seperti `Ord` dan `Eq`. Functor merepresentasikan tipe yang dapat di-map, contoh paling mudahnya adalah list.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Fungsi `fmap` adalah implementasi dari Functor . Jika kita anggap f adalah sebuah list, maka `fmap` menerima sebuah fungsi, mengaplikasikannya pada list dan mengembalikan list baru setelah diterapkan fungsi tersebut. Kenyataannya, `map` adalah sebuah `fmap` untuk tipe list. Terdapat juga `<$>`, yaitu versi inflix dari `fmap`.

4. Traversable Setelah mengetahui Functor, terdapat istilah `Traversable`, yaitu class of data structure yang dapat di-**traversed** atau dilintasi dari kiri ke kanan sembari melakukan suatu action untuk setiap elemennya.

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
```

5. Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Contoh paling mudah menurut saya untuk memahami Applicative adalah dengan menggunakan list comprehension, misalkan terdapat:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Maka kita dapat mendapatkan list tersebut menggunakan `<*>`, yaitu dengan:

```
(*) <$> [2,5,10] <*> [8,10,11]
--[(*2),(*5),(*10)] <*> [8,10,11]
--[(*2) 8,(*2) 10,(*2) 11,(*5) 8,(*5) 10,(*5) 11,(*10) 8,(*10) 10,(*10) 11]
--[16,20,22,40,50,55,80,100,110]
```

Just dan Maybe merupakan Applicative. `<*>` dapat digunakan untuk melakukan operasi pada keduanya tanpa perlu "unwrap". Contoh:

```
ghci> (+) <$> Just 5 <*> Just 10
Just 15
ghci> (,,) <$> Just 1 <*> Just 2 <*> Just 3
Just (1,2,3)
```

6. sequanceA Jika dilihat dari tipenya,

```
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
```

Jika dilihat dari tipenya, fungsi sequenceA dapat membalik yang sebelumnya sebuah Traversable t berisi (Applicative f a) menjadi Applicative f dari Traversable t yang berisi a. Hal ini dilakukan dengan mengevaluasi setiap **action** dalam struktur dari kiri ke kanan, dan mengumpulkan hasilnya. Contoh sederhananya adalah, kita tahu jika Just merupakan sebuah Applicative, maka:

```
ghci> sequenceA [Just 1, Just 2, Just 3]
Just [1,2,3]
```

Pada kasus ini t = list dan f = Just.

7. Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Terdapat dua fungsi minimum yang perlu diimplementasi untuk menjadi Alternative, yaitu `empty` dan `(<|>)`. Fungsi `empty` menjadi identitas dari fungsi `<|>` sedangkan fungsi `<|>` adalah sebuah fungsi biner yang bersifat asosiatif, Contoh kasusnya pada Parser Combinator ini adalah:

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

Identitas (empty) dari Parser didefinisikan sebagai sebuah Parser dengan fungsi yang me-return nothing. Efeknya adalah ketika terdapat operasi `Parser a <|> Parser b` dan `Parser a` me-return Nothing, sedangkan `Parser b` mereturn x, maka hasilnya adalah tetap x.