What is a Parser?

Jeehoon Kang

> Parsing is the process of analyzing a *string of symbols* into *meaningful constituents* that contain *semantic information*.

Based on that definition, 'parsing' can be represented as a function that takes a `String` and extracts an *element of interest*, `a`

```
parser a :: String -> a
```

To understand the **entire string** (which might contain various elements of interest), you need to **continue working on the rest of the string**.

Also, the element of interest **might not be present** in the string!

> So return the leftover string as well, and wrap it in a `Maybe`

```
parser a = String -> Maybe (a, String)
```

You could probably extend the `Maybe` part to let the parser return other failure information, etc. (Such as using `Either`)

Structuring the Parser

> Abstraction, Encapsulation, Extensibility

- Having the parser as a function *exposes the implementation detail*. Working with it will quickly become dirty, and accidental matching type signatures can confuse the code further. (Type safety)
- Distinct types help add custom behavior, as you can define *instances of various type classes* for the type.
- Haskell's `newtype` has no runtime overhead compared to underlying types. (Can only have one constructor with one field)

```haskell
-- Create newtype Parser from generic type a
-- Define a record with one field with an actionable name (Haskell creates a function of the same name that unwraps and extracts the value)

-- Given a String, return parsed value 'a' and remaining 'String'
newtype Parser a = Parser
        { runParser :: String -> Maybe (a, String)
        }
```

Defining instances of useful Type Classes

> While building a parser, it's natural to come across patterns that emerge from Strings.
>
> For example, say you wanted to parse a specific character, then realize you want to parse a sequence of matching characters (a word).
>
> One might think extending from a parser for `Char`s to a parser for `String`s with a `map` operation should be easy. But this gives a *list of character parsers*, rather than a *parser for a list of characters*.
>
> Is there an easy way of doing this sort of inversion operation?

There are many cases where standard functional paradigms can help extend the parsers into more useful and general cases.

Several widely used type classes fundamental to functional programming, such as `Functor, Applicative, Monad, Foldable, Traversable`, and so on, help abstract over common patterns in computation.

We're going to focus on three classes, namely:

- Functor: Represents types that can be *mapped* over
- Applicative: Extends `Functor` with capabilities for *applying functions* wrapped in the context *to values in the context*
- Monad: Builds on `Applicative`, providing a way to *sequence/chain computations* with new contexts

Already, the `Applicative` class seems like it might have something we need for the String parser!

> The cons operator `(:)` creates a sequence out of elements, so if we can lift the operator into the context of `Parser`(*functions wrapped in context*),
>
> and *apply it* to the value of the String Parser recursively(*values in the context*),
>
> we can build a String Parser out of Char Parsers!

Character Parser

First, we define a character parser

```haskell
-- Given a character to parse, returns a Parser of it
charP :: Char -> Parser Char
charP x = _a  -- hole
```

`ghc` says the hole should be of type `Found hole: _a :: Parser Char`, so let's wrap the whole with `Parser`

```haskell
charP :: Char -> Parser Char
charP x =  Parser _a  -- hole
```

now it says `Found hole: _a :: String -> Maybe (Char, String)`, a function!

Let's convert the hole as a function,

```haskell
charP :: Char -> Parser Char
charP x =  Parser $ \input -> _a  -- hole
```

`Found hole: _a :: Maybe (Char, String)`, given that

```
Relevant bindings include
  input :: String
  x :: Char
```

We want to parse if the character exists at the start of the string, so

```haskell
charP :: Char -> Parser Char
charP x = Parser $ \input ->
    case input of
        y : ys | y == x -> Just (x, ys)
        _ -> Nothing
```

or equivalently

```haskell
charP :: Char -> Parser Char
charP x = Parser f
  where
    f (y:ys)
        | y == x = Just (x, ys)
        | otherwise = Nothing
    f [] = Nothing
```

`All good`. Yay!

Running the following in `ghci`

```
ghci> runParser (charP 'a') "apple"
```

returns:

```
Just ('a',"pple")
```

Extending Parser with the Functor Instance

> Great! We have our Character Parser, but as we've discussed above, if it were an Applicative, we could easily extend it into a String Parser..

> To prove that our Parser is an Applicative, we must first prove that it is an instance of a `Functor`. (`class Functor f => Applicative f where ...`)

> The minimum implementation of a `Functor` requires a `fmap` definition where `fmap :: (a -> b) -> f a -> f b)`. That is, you want to inject a function into the functor. (`fmap` is synonymous to `<$>`)

Give Parser a Functor instance (`fmap` that takes a function and Parser as Functor)

```
instance Functor Parser where
        fmap f (Parser p) = _a  --hole
```

```
Found hole: _a :: Parser b

• Relevant bindings include
        p :: String -> Maybe (a, String)
        f :: a -> b
        fmap :: (a -> b) -> Parser a -> Parser b
```

Again, like the character parser implementation, let's wrap the hole as a Parser, then the hole will become `Found hole: _a :: String -> Maybe (b, String)`, which we can also wrap in a lambda function giving:

```
instance Functor Parser where
        fmap f (Parser p) = Parser $ \input -> _a  --hole
```

```
Found hole: _a :: Maybe (b, String)

- Relevant bindings include
        input :: String
        p :: String -> Maybe (a, String)
        f :: a -> b
        fmap :: (a -> b) -> Parser a -> Parser b
```

So we have `input` which we can wrap the parser `p` around to expose `a` (although wrapped in a `Maybe`).

Then, applying `f` on it will transform that `a` to `b`.

Finally, we return it with the rest of the string, and voila!

```haskell
instance Functor Parser where
        fmap f (Parser p) = Parser $ \input ->
            -- Wrap input(String) with given Parser, returning Maybe(a, String)
                case p input of
                        Just (y, ys) -> Just (f y, ys)
                        Nothing -> Nothing
```

Now Parser is a Functor!

Running the following in `ghci`

```
ghci> :t fmap ord (charP 'a')
```

returns:

```
fmap ord (charP 'a') :: Parser Int
```

and thus

```
ghci> runParser (fmap ord (charP 'a')) "apple"
```

gives

```
Just (97,"pple")
```

As a functor, we were able to map functions to the values within the context of Parser.

Proving that Parser is an Applicative

> Now that we have a Functor instance of Parser, we can extend it into an Applicative.
>
> An Applicative requires a minimal implementation of `pure` and `<*>`

Which have the following type definitions:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

You know the drill, `pure` first:
`a -> f a`, whatever you're given, return an Applicative context that contains it.

In terms of the Parser, it needs to create a Parser out of what is given and just return itself without consuming or transforming the input. (A minimal context that parses itself without consuming it?)

```
instance Applicative Parser where
        pure p = Parser $ \input -> Just(p, input)
```

Next up, `(<*>)`:
`f (a -> b) -> f a -> f b`. Takes a function from `a` to `b`, wrapped as an Applicative, and an Applicative that contains `a` and returns an Applicative that contains `b`.

> It's similar to `fmap :: (a -> b) -> f a -> f b)`, the only difference is whether the function is wrapped in the context or not.

In the context of Parser, it means if you have two Parsers:

- One that *parses the input and produces a function*
- Another that *parses the remaining input and produces a value*
  you can apply the function to the value! It hints at a *sequential combination of Parsers*!

```
instance Applicative Parser where
        pure p = Parser $ \input -> Just(p, input)
        (Parser p1) <*> (Parser p2) = Parser $ \input -> _a
        -- hole, p1 is a function, p2 a value
```

```
Found hole: _a :: Maybe (b, String)
- Relevant bindings include
        input :: String
        p2 :: String -> Maybe (a, String)
        p1 :: String -> Maybe (a -> b, String)
        (<*>) :: Parser (a -> b) -> Parser a
```

Apply `p1` to `input` and get that function `a -> b`

```haskell
instance Applicative Parser where
        pure p = Parser $ \input -> Just(p, input)
        (Parser p1) <*> (Parser p2) = Parser $ \input ->
                case p1 input of
                        Just (f, rest) -> _a.  -- hole,  f is a -> b
                        Nothing -> Nothing
```

```
Found hole: _a :: Maybe (b, String)
- Relevant bindings include
        input :: String
        rest :: String
        f :: a -> b
        p2 :: String -> Maybe (a, String)
        p1 :: String -> Maybe (a -> b, String)
        (<*>) :: Parser (a -> b) -> Parser a
```

We have all the ingredients we need.

> If we apply `p2` to the `rest` of the string, we expose `a`,
>
> which we can transform to `b` with `f`,
>
> and return it as a `Maybe`.

```haskell
instance Applicative Parser where
        pure p = Parser $ \input -> Just(p, input)
        (Parser p1) <*> (Parser p2) = Parser $ \input ->
                case p1 input of
                        Just (f, rest) ->
                                case p2 rest of
                                        Just (x, leftover) -> Just(f x, leftover)
                                        Nothing -> Nothing
                        Nothing -> Nothing
```

```
All good
```

We have proven that Parser is an Applicative!

Running the following in `ghci`

```
:t fmap (,) (charP 'a')
```

we have a Parser that returns a function that returns a tuple (with 'a' as the first element)

```
fmap (,) (charP 'a') :: Parser (b -> (Char, b))
```

Chaining this with another `charP` that parses 'p' on the rest of the string:

```
ghci> :t fmap (,) (charP 'a') <*> charP 'p'


fmap (,) (charP 'a') <*> charP 'p' :: Parser (Char, Char)
```

You get a Parser that parses 'a' and 'p' into a tuple ('a', 'p')!

```
ghci> runParser (fmap (,) (charP 'a') <*> charP 'p') "apple"
Just (('a','p'),"ple")


ghci> runParser (fmap (,) (charP 'a') <*> charP 'p') "attack"
Nothing
```

That looks similar to a String Parser!

From `charP` to `stringP`

> With the Parser being an Applicative, we can extend the character parser into a string one.

```haskell
stringP :: String -> Parser String
-- Base case, minimally wrap an empty string into an Applicative Parser instance
stringP [] = pure []

-- Recursive chaining
stringP (c:cs) = (:) <$> charP c <*> stringP cs

```

Create a Parser parsing `c`, which is then transformed to a function that prepends it to a string with `fmap`. (Inside the context of Parser)

That function inside the Parser context is applied with `<*>` to the result of the string Parser of the rest of the string `cs` recursively until the `pure` case if necessary.

```
ghci> runParser (stringP "apple") "apples are tasty"

Just ("apple","s are tasty")
```

Or with the `sequenceA` function from Base Haskell, with signature `sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)`

```haskell
stringP :: String -> Parser String
stringP = sequenceA . map charP
```