Introduction

Combinator parsers are a powerful and flexible way to construct parsers by combining small, reusable parser components. These components, or ”combinators”, can be used to build complex parsers in a modular and readable way.

Example: Arithmetic expressions

Let’s consider a simple example of parsing arithmetic expressions like ”3 + 5”.

Basic parsers

We define basic parsers for digits and operators.

- **Digit parser**: Parses a single digit.
- **Operator parser**: Parses the addition operator ”+”.

Combining parsers

We combine these basic parsers above to parse an entire expression.

- **Expression parser**: Parses a digit, followed by an operator, followed by another digit.

Implementation

Here is a *pseudocode* implementation of the parsers:

```
def digit_parser(s):
    if s and s[0].isdigit():
        return [(s[0], s[1:])]
    else:
        return []

def operator_parser(s):
    if s and s[0] == '+':
        return [('+', s[1:])]
    else:
        return []

def expression_parser(s):
    result1 = digit_parser(s)
    if not result1:
        return []

    (digit1, rest1) = result1[0]
```

```
result2 = operator_parser(rest1)
if not result2:
    return []

(_, rest2) = result2[0]
result3 = digit_parser(rest2)
if not result3:
    return []

(digit2, rest3) = result3[0]
return [(int(digit1) + int(digit2), rest3)]
```

As can be seen it also, besides parsing, *evaluates* the expression with the last statements.

Other observations are that if no match is found, an empty result ("list") is returned. The split of parsed item and the rest unparsed shows how the parsers transfer information in between each other.

Usage

To parse the expression "3 + 5":

```
expression_parser("3+5")
# Output: [(8, "")]
```

Combinator Parsers

1. Basic

- **Parser**: A parser is a function that, given an input string, returns a set of possible parses. Each parse is typically *a pair* consisting of the *parsed result* and the *remaining unparsed* portion of the input.

- **Combinator**: A combinator is a higher-order function that takes *one or more parsers* as arguments and returns *a new parser*.

2. Formally defined parsers

Let P be a parser with the type signature:

$$P : \Sigma^* \to \mathcal{P}(\mathcal{R} \times \Sigma^*)$$

where Σ^* is the set of all possible input strings, $\mathcal{R}$ is the set of parse results, and $\mathcal{P}$ denotes the power set.

This means that a parser P takes an input string and returns a set of pairs. Each pair consists of a result and the remaining unparsed part of the input.

3. Combinators

- `Choice`: The choice combinator alt tries two parsers and returns the result of the first successful parser.

$$\text{alt}(P,Q)\ \textbf{def}\ \lambda s.\text{filter results from } (P(s) \cup Q(s)) \tag{1}$$

- `Sequence`: The sequence combinator seq combines two parsers such that the second parser is applied to the remaining input after the first parser succeeds.

$$\text{seq}(P,Q)\ \textbf{def}\ \lambda s.\{(r_1 \oplus r_2, s'') \mid (r_1, s') \in P(s) \tag{2}$$
$$\text{and } (r_2, s'') \in Q(s')\} \tag{3}$$

  where $\oplus$ denotes some combination of the results r_1 and r_2.

- `Many`: The many combinator many applies a parser zero or more times.

$$\text{many}(P)\ \textbf{def}\ \lambda s.\{(\textit{results}, s') \mid \tag{4}$$
$$\textit{results} \text{ is a list of results from zero or more applications of} \tag{5}$$
$$P \text{ and } s' \text{ is the remaining input}\} \tag{6}$$

- `Many1`: The many1 combinator many1 applies a parser one or more times.

$$\text{many1}(P)\ \textbf{def}\ \text{seq}(P, \text{many}(P)) \tag{7}$$

- `Option`: The option combinator opt applies a parser and returns a default value if the parser fails.

$$\text{opt}(P,d)\ \textbf{def}\ \lambda s.\{(r,s) \mid (r,s) \in P(s)\} \cup \{(d,s) \mid \text{if } P(s) \text{ fails}\} \tag{8}$$

4. Formal definitions using Category Theory

In category theory, parsers can be modeled as functors between categories. For instance:

- **Category of Parsers**:
  - Objects: Parsers.
  - Morphisms: Combinators.
- **Functorial Composition**: A parser combinator can be viewed as a functor that maps parsers to new parsers. For example, the sequence combinator is a functor that maps a pair of parsers to a new parser.

Summary

The mathematical formalism of combinator parsers involves defining parsers as functions that map input strings to sets of parse results and remaining input. Combinators are higher-order functions that combine parsers in various ways. This formalism can be described using set theory, formal language theory, and category theory.

Example: Parsing `add(x, y)` Expression

1. Category of Parsers

Let $\mathcal{P}$ be the category of parsers:

- **Objects**: Parsers P that parse components of the expression.
- **Morphisms**: Functions that transform parsed results.

2. Basic Parsers

Define basic parsers for the operator and the arguments:

- **Parser for "add"**: P_{add}
- **Parser for argument "x"**: P_x
- **Parser for argument "y"**: P_y

These parsers are defined as:

$$P_{\text{add}}(s) = \begin{cases} \{(\text{"add"}, s')\} & \text{if } s = \text{"add"} \cdot s' \\ \emptyset & \text{otherwise} \end{cases} \tag{9}$$

$$P_x(s) = \begin{cases} \{(x, s')\} & \text{if } s = x \cdot s' \\ \emptyset & \text{otherwise} \end{cases} \tag{10}$$

$$P_y(s) = \begin{cases} \{(y, s')\} & \text{if } s = y \cdot s' \\ \emptyset & \text{otherwise} \end{cases} \tag{11}$$

3. Combining Parsers with Functors

Define a functor F that combines these parsers:

$$F_{P_{\text{add}}, P_x, P_y} : \mathcal{P}_{\text{add}} \times \mathcal{P}_x \times \mathcal{P}_y \to \mathcal{P} \tag{12}$$

The functor F maps P_{add}, P_x, and P_y to a new parser that parses the expression `add(3, 4)`.

4. Functor Action on Objects

Let P_{add}, P_x, and P_y be parsers:

$$F(P_{\text{add}}, P_x, P_y) = \lambda s.\{(\text{add}(a,b), s'') \mid (\text{"add"}, s') \tag{13}$$
$$\in P_{\text{add}}(s), (a, s'') \in P_x(s'), (b, s'') \in P_y(s')\} \tag{14}$$

Here, $P_{\text{add}}(s)$, $P_x(s')$, and $P_y(s'')$ parse "add", x, and y from the input strings s, s', and s'', respectively. The functor combines these parsers to produce a parser that parses `add(3, 4)`.

5. Functor Action on Morphisms

Let $\phi : P_x \to P_{x'}$ and $\psi : P_y \to P_{y'}$ be morphisms (functions that transform parsed results):

$$F(\phi, \psi) = \lambda(\text{add}(a,b), s'').(\text{add}(\phi(a), \psi(b)), s'') \tag{15}$$

Where ϕ and ψ transform the results of the parsers P_x and P_y, respectively.

6. Example: Parsing `add(3, 4)`

Consider the parsers P_{add}, P_x, and P_y that parse the strings "add", "3", and "4":

$$P_{\text{add}}(s) = \{(\text{"add"}, s)\} \tag{16}$$

$$P_x(s) = \{(3, s)\} \tag{17}$$

$$P_y(s') = \{(4, s')\} \tag{18}$$

The functor F produces a new parser $F(P_{\text{add}}, P_x, P_y)$ that parses the expression:

$$F(P_{\text{add}}, P_x, P_y)(s) = \{(\text{add}(3,4), s'') \mid (\text{"add"}, s') \tag{19}$$
$$\in P_{\text{add}}(s), (3, s'') \in P_x(s'), (4, s'') \in P_y(s'')\} \tag{20}$$

Thus, the parser $F(P_{\text{add}}, P_x, P_y)$ parses the expression `add(3, 4)`.

7. Commutative Diagram

The commutative diagram for the functor F acting on parsers and their morphisms can be represented as follows:

$$\begin{tikzcd}
P_{\text{add}} \arrow[d, "\phi"'] & P_x \arrow[d, "\psi"'] & P_y \arrow[r] \arrow[d, "\chi"'] & F_{P_{\text{add}}, P_x, P_y}(P_{\text{add}}, P_x, P_y) \arrow[d, "F_{\phi,\psi,\chi}"] \\
P_{\text{add}'} & P_{x'} & P_{y'} \arrow[r] & F_{P_{\text{add}'}, P_{x'}, P_{y'}}(P_{\text{add}'}, P_{x'}, P_{y'})
\end{tikzcd}$$

In this diagram:

- P_{add}, P_x, and P_y are the initial parsers.
- $P_{\text{add}'}, P_{x'}$, and $P_{y'}$ are the transformed parsers.
- ϕ, ψ, and χ are the morphisms that transform the parsing results.
- $F_{P_{\text{add}}, P_x, P_y}$ is the functor that combines the parsers.
- $F_{\phi,\psi,\chi}$ is the functor that combines the morphisms.

This diagram shows how the functor F acts on both the parsers and their transformations, ensuring that the parsing process and the transformations commute appropriately.

Summary

The functor example illustrates how individual parsers for components of an expression (like `add`, `x`, and `y`) can be combined into a single parser that parses a composite expression `add(x, y)`. The functor acts on the parsers and their transformations, ensuring that the overall parsing process is consistent and commutative.

By defining parsers and their combinations formally, we can rigorously understand and design complex parsers using category theory.