

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:

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

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.

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

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

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

$$\text{and } (r_2, s'') \in Q(s')\} \quad (3)$$

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.

$$\text{many}(P) \text{ def } \lambda s. \{(results, s') \mid \quad (4)$$

$$results \text{ is a list of results from zero or more applications of} \quad (5)$$

$$P \text{ and } s' \text{ is the remaining input}\} \quad (6)$$

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

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

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

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

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, providing a rigorous foundation for understanding and designing parser combinators.

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:

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

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

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

3. Combining Parsers with Functors

Define a functor F that combines these parsers:

$$F : \mathcal{P} \times \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$$

The functor F maps P_{add} , P_x , and P_y to a new parser that parses the expression $\text{add}(x, y)$.

4. Functor Action on Objects

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

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

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 $\text{add}(x, y)$.

5. Functor Action on Morphisms

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

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

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

6. Example: Parsing $\text{add}(3, 4)$

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

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

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

$$P_y(s) = \{(4, s)\}$$

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

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

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

7. Diagram

Here is a commutative diagram showing the functor F :

$$\begin{array}{ccccc} P_{\text{add}} & P_x & P_y & \xrightarrow{F} & F(P_{\text{add}}, P_x, P_y) \\ \phi \downarrow & \phi \downarrow & \psi \downarrow & & \downarrow F(\phi, \psi) \\ P_{\text{add}} & P_{x'} & P_{y'} & \xrightarrow{F} & F(P_{\text{add}}, P_{x'}, P_{y'}) \end{array}$$