Combinator Parsers

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* (Python like) 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)]
```

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

As can be seen it also, besides parsing, *evaluates* the expression with the last statements (not part of parsing).

Moreover, the "defs" are related here in the following way:

```
def digit_parser(s):
    ...
def operator_parser(s):
    ...
def expression_parser(s):
    ... = digit_parser(s)
    ... = operator_parser(rest1)
    ... = digit_parser(rest2)
```

When a match is made, e.g. a digit is found by 'digit_parser,' the returned value is a pair '(s[0], s[1:]),' the first a digit and the second the rest of the input for parsing:

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

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

One thing to take note of here is that the "defs" doesn't return a parser, but only what has been parsed. In real applications, often functional languages, a new function is returned for later use. Thus a *combinations of parsers* represent the grammar of the language.

Theory

Basics

- **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*.

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.

Base combinators

- `Choice`: The choice combinator 'alt' tries two parsers and returns the result of the first successful parser. The choice combinator relies on the concept of union $\cup$ of sets, representing the combination of possible parsing results from both P and Q.

  The 'alt' combinator takes two parsers P and Q and returns a new parser that tries to apply P first. If P succeeds, the result is returned, but if P fails, the parser tries Q instead. The first successful parse is then returned.

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

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

succeeds. It uses the Cartesian product to combine the results from two parsers, and applies a binary operation to merge their results.

The combinator 'seq' takes two parsers P and Q and returns a new parser that first applies P to the input. If P succeeds, the remaining input is passed to Q. The results from P and Q are combined using an operation $\oplus$.

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

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

- `Many`: The combinator 'many' applies a parser zero or more times. It is built on the concept of iteration, where a parser is applied in a loop until it fails, collecting results as it progresses.

  The 'many' combinator applies a parser P repeatedly to the input, zero or more times, and collects the results into a list.

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

- `Many1`: The combinator 'many1' applies a parser one or more times as it combines the 'sequence' combinator and the 'many' combinator, enforcing that the parser succeeds at least once.

  The combinator 'many1' is a variant of 'many' that ensures the parser P is applied at least once. It is defined as a sequence of P followed by 'many(P)'.

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

- `Option`: The option combinator 'opt' introduces conditional logic into parsing, applying a parser and returns a default value if the parser fails.

  The 'opt' tries to apply a parser P to the input. If P succeeds, its result is returned. If P fails, a default value d is returned instead.

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

The combinators *alt*, *seq*, *many*, *many1*, and *opt* form a foundation for general combinatorial parsing. These higher-order functions allow construction of complex parsers by combining simpler ones. Each combinator introduces a specific kind of control flow or repetition into the parsing process, making them versatile tools for defining grammars and parsers in a modular and reusable way.

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 'seq' above is a functor that maps *a pair of parsers* to *a new parser*.

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

1. Category of Parsers

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

- **Objects**: Parsers P, which are functions that take an input and return a set of possible parses.
- **Morphisms**: Functions between these parsers, which map the output of one parser to another.

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:

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

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

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

These parsers are functions from strings to sets of pairs (parsed value, remaining string). They are examples of deterministic functions that form the basis of the objects in our category $\mathcal{P}$.

3. Combining Parsers with Functors

To parse a full expression like 'add(x, y)', we need to combine these individual parsers. This is done using a *functor* F which operates on a product of parsers. 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 an expression. Here a functor F is a mapping between categories that preserves the structure of objects and morphisms. So, F maps a tuple of parsers to a single parser that processes the entire expression.

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.

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':

$$P_{\text{add}}(s) = \{(\text{"add"}, s') \mid s' \text{ is the remainder of} \tag{16}$$
$$s \text{ after matching the string "add"}\} \tag{17}$$

$$P_x(s') = \{(3, s'') \mid s'' \text{ is the remainder of} \tag{18}$$
$$s' \text{ after matching the string "3"}\} \tag{19}$$

$$P_y(s'') = \{(4, s''') \mid s''' \text{ is the remainder of} \tag{20}$$
$$s'' \text{ after matching the string "4"}\} \tag{21}$$

The functor F combines these parsers to create a new parser $F(P_{\text{add}}, P_x, P_y)$ that parses the entire expression:

$$F(P_{\text{add}}, P_x, P_y)(s) = \{(\text{"add"}(3, 4), s''') \mid (\text{"add"}, s') \tag{22}$$
$$\in P_{\text{add}}(s), (3, s'') \in P_x(s'), (4, s''') \in P_y(s'')\} \tag{23}$$

This expression means:

- P_{add} parses the 'add' part of the string.
- P_x parses the '3' part of the string.
- P_y parses the '4' part of the string.
- The functor F combines these parsed results into the final parsed expression, which is add(3, 4).

Thus, the parser $F(P_{\text{add}}, P_x, P_y)$ correctly 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:

$$\begin{array}{ccccc} P_{\text{add}} & & P_x & & P_y \longrightarrow F_{P_{\text{add}},P_x,P_y}(P_{\text{add}}, P_x, P_y) \\ \phi \downarrow & & \psi \downarrow & & \chi \downarrow \qquad\qquad \downarrow F_{\phi,\psi,\chi} \\ P_{\text{add}'} & & P_{x'} & & P_{y'} \longrightarrow F_{P_{\text{add}'},P_{x'},P_{y'}}(P_{\text{add}'}, P_{x'}, P_{y'}) \end{array} \tag{24}$$

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.

Example: Parsing `"add(x, y)"` with combinators

Above we have seen how sample parsers works in the context of mathematical constructs, category theory, formal languages, set theory, and so on. Looking at it from the previous fundamental combinators, let us see how instead parsing a **string** of the example `"add(x, y)"` can unfold using *combinators*.

We define parsers for each component using simple *string matching parsers*:

- P_{add}: Parses the string `"add"`.
- P_x: Parses the string `"x"`.
- P_y: Parses the string `"y"`.

Sequence 'seq'

To parse the entire expression `"add(x, y)"`, we need to parse `"add"`, then `"x"`, and finally `"y"`.

1. Parsing `"add(x, "`: We first parse `"add"` and then the `"x"` argument. We use the 'seq' combinator to create a parser that does this in a sequence:

$$P_{\text{add\_x}} = \text{seq}(P_{\text{add}}, P_x)$$

This parser first applies P_{add} to parse `"add"`, then uses P_x to parse `"x"`.

2. Parsing `"y)"`: Similarly, we can create a parser to handle `"y"` using the sequence combinator again:

$$P_{\text{add\_x\_y}} = \text{seq}(P_{\text{add\_x}}, P_y)$$

This parser first applies $P_{\text{add\_x}}$ to parse `"add(x,"` and then uses P_y to parse `"y)"`.

The sequence parser $P_{\text{add\_x\_y}}$ parse the entire expression by combining the individual parsers in the required order. What is not explicitly addressed here, but must be handled in practice, is the parsing of both the comma separator, parenthesis, and any surrounding whitespace. It is implied here that the parsers will handle such things.

Choice 'alt'

Suppose the expression can have a variant like `"add(y, x)"` (order reversed). We can use the 'alt' combinator to allow for both orders of arguments:

$$P_{\text{add\_choice}} = \text{alt}(P_{\text{add\_x\_y}}, P_{\text{add\_y\_x}})$$

Here, $P_{\text{add\_choice}}$ will first try to parse using $P_{\text{add\_x\_y}}$. If it fails, it will try $P_{\text{add\_y\_x}}$, allowing the parser to accept either `"add(x, y)"` or `"add(y, x)"`.

Many 'many'

If the expression can contain multiple arguments, such as `"add(x, y, z)"`, the `many` combinator could be used to parse a comma-separated list of arguments:

$$P_{\text{args}} = \text{many}(\text{seq}(P_x, P_y))$$

This would allow the parser to handle multiple arguments by *repeatedly* applying the parsers for each argument.

Option 'opt'

If the expression `"add(x, y)"` could optionally include additional features, such as a keyword argument, a flag, or explicit type declaration e.g. `"add(x, y) as bool"`. The 'opt' combinator could handle these optional parts, e.g. flag:

$$P_{\text{optional\_flag}} = \text{opt}(P_{\text{flag}}, \text{"default\_flag"})$$

This parser would apply P_{flag} if the flag is present in the input, otherwise, it would default to `"default_flag"`.

Thus,

- 'seq' is used to enforce the correct order of parsing components: `"add"`, followed by `"x"`, and then `"y"`.
- 'alt' allows for alternative structures, such as handling both `"add(x, y)"` and `"add(y, x)"`.
- 'many' would be used if the expression included a variable number of arguments.
- 'opt' can handle optional components, such as additional flags or parameters in the expression.

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.

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.

At last we can see how in practice some of the fundamental combinators can be used for parsing a string `"add(x, y)"`. We also notice how we can extend or change the parsing without changing too much dependency on other parsing parts. This is the robustness of combinator parsers.