Abstract

Parsita is a parser combinator library written in Python for Python. Parser combinators are a tool available in many programming languages for writing parsing grammars directly in the language of interest rather than writing a grammar file to be compiled by a parser generator. In the case of Parsita, this means writing parsers in Python. With the careful use/abuse of Python operators, Parsita makes it intuitive and easy to write parsers for any file the user desires to parse.

Introduction

To import common file types, like JSON and CSV, there are many available tools to do so, and users should use them when available. But scientific programmers occasionally encounter obscure file formats, often describing proprietary data formats or specialized mathematical models. If no parser exists, then one will need to be written if the objects are to be used in Python.

Scientists also occasionally want to design their own file formats, often to provide an easier way to input complex information, such as mathematical models, into the system. To do so requires designing a grammar and writing a parser. Using a standard parser generator such as ANTLR, YACC, or Bison is a daunting task. Even with the high-quality bindings available for Python, the user is still expected to learn the language of the generator. Parser combinators are an easier solution in many programming languages. Instead of requiring the user to learn a separate language in which to define the grammar, parser combinators allow writing the grammar directly in the target language. The code, written in the target language, is readable as a proper grammar with the clever use of operator overloading.

In keeping with the parser combinator design, each element of the grammar is a standalone parser. These elements can be combined into more complex parsers. To use a Parsita parser one calls the parse(source: str) method. This function returns a Result, which has two concrete subclasses Success and Failure.

Terminal Parser Combinators

Parsita does not require a tokenizer to be run over the text before parsing. While it is possible to run a Parsita parser over a stream of tokens, it is recommended to simply define the terminal patterns using lit or reg.

lit([litdata]) word parser

This is the simplest parser. It matches the exact string provided and returns the string as its value. If multiple arguments are provided, it tries each one in succession, returning the first one it finds.

```python
class HelloParsers(TextParsers):
    hello = lit('Hello World!') >> inter
    assert HelloParsers.hello.parse('Hello World!') == Success('Hello World!')
    assert HelloParsers.hello.parse('Hello World Hello World') == Success('Hello World!')
    assert HelloParsers.hello.parse('Hello World Hello World Hello World') == Success('Hello World!')
```

lit([litdata]) word parser

In most cases, the call to lit is handled automatically. If a bare string is provided to the functions and operators discussed to the left, it will be promoted to a literal parser whenever possible. Only when an operator is between two Python types, like a string and a string 'a' | 'b', will this "implicit conversion" not take place and you will have to use lit (e.g. lit('a', 'b') and lit('100') > int).

reg([patterns]) regular expression parser

Like lit, this matches a string and returns it, but the matching is done with a regular expression.

```python
class IntegerParsers(TextParsers):
    integer = reg(r'\d+') >> inter
    assert IntegerParsers.integer.parse('123') == Success('123')
    assert IntegerParsers.integer.parse('-123') == Success('-123')
```

Basic Parsers

The power and name of parser combinators comes from the ability to combine parsers into complex structures, whether it is to define a parser that is a sequence of simpler parsers, a choice of several parsers, or a transformation of the results of several parsers into a single object in the target language.

parser | parser alternative parser

This tries to match parser1. If it fails, it then tries to match parser2. If both fail, it returns the failure message from whichever one got farther. Either side can be a bare string, but not both, because 'a' | 'b' tries to call or on str which fails. To try alternative literals, use lit with multiple arguments.

```python
class ArithmeticParsers(TextParsers):
    expr = reg(r'\+\d\+\d\+\d') >> sum >> inter
    assert ArithmeticParsers.expr.parse('2+1+3') == Success(6)
```

parser & parser optional parser

All the parsers shown so far will match exactly one thing. A sequential parser is the syntax for matching one parser and then another after it. When working in the TextParsers context, the two may be separated by whitespace. The value returned is a list of all the values returned by each parser. If there are multiple parsers separated by &, a list of the same length as the number of parsers is returned. Like , either side may be a bare string, but not both. In accordance with Python's operator precedence, binds more tightly than |

```python
class ArithmeticParsers(TextParsers):
    expr = fwd() & reg(r'[^+]\d') >> inter
    assert ArithmeticParsers.expr.parse('2+(1+2)+3') == Success(8)
```

parser > parser required parser

The discard left and discard right parsers match the exact same text as parser1 & parser2, but rather than return a list of values from both, the left value in >> and the right value in << is discarded so that only the remaining value is returned. A mnemonic to help remember which is which is to imagine the symbols as open mouths eating the parser to be discarded.

```python
class ArithmeticParsers(TextParsers):
    expr = reg(r'^\d') >> inter & reg(r'[^+]\d') >> inter
    assert ArithmeticParsers.expr.parse('2+(1+2)+3') == Success(8)
```

parser >> parser required parser

An optional parser tries to match its argument. If the argument succeeds, it returns a list of length one with the successful value as its only element. If the argument fails, then opt succeeds anyway, but returns an empty list and consumes no input.

```python
class ArithmeticParsers(TextParsers):
    expr = reg(r'^\d') >> inter & reg(r'[^+]\d') >> inter
    assert ArithmeticParsers.expr.parse('2+(1+2)+3') == Success(8)
```

-parser (parser) optional parser

A repeated parser matches repeated instances of its parser argument. It returns a list with each element being the value of one match. repl1 only succeeds if at least one match is found. repl always succeeds, returning an empty list if no matches are found.

```python
class SummationParsers(TextParsers):
    expr = reg(r'^\d') >> inter & reg(r'[^+]\d') >> inter
    assert SummationParsers.summation.parse('2+1+2+3+5') == Success([2, 1, 2, 3, 5])
```

Miscellaneous Parsers

A predicate parser matches parser and, if it succeeds, runs a test function predicate on the parsed value. If predicate returns True, the predicate parser succeeds, returning the same value; if it returns False, the parser fails with the message that it is expecting description.

```python
class IntervalParsers(TextParsers):
    number = reg(r'^\d') >> inter
    def predicate(interval, lambda x: x[0] <= x[1], "ordered pair")
    assert IntervalParsers.interval.parse('[1, 2]') == Success([1, 2])
```

EOF end of file

A parser that matches the end of the input stream. It is not necessary to include this on every parser; the parse method on every parser is successful if it matches the entire input. The eof parser is only needed to indicate that the preceding parser is only valid at the end of the input. Most commonly, it is used as an alternative to an end token when the end token may be omitted at the end of the input. Note that eof is not a function—it is a complete parser itself.

```python
class ArithmeticParsers(TextParsers):
    option = reg(r'[^+\d]\d') >> & reg(r'[^+]\d') >> & inter
    assert ArithmeticParsers.options.parre downgrade
```

Fwd() forward declaration

This creates a forward declaration for a parser to be defined later. This function is not typically needed because forward declarations are created automatically within the class bodies of subclasses of TextParsers and GeneralParsers, which is the recommended way to use Parsita. This function exists so you can create a forward declaration manually because you are either working outside of the magic classes or wish to define them manually to make your IDE happier.

To use fwd, first assign fwd() to a variable, then use that variable in other combinators like any other parser, then call the define(parser; Parser method on the forward declaration to provide it with its definition. The forward declaration will now look and act like the definition provided.

```python
class ArithmeticParsers(TextParsers):
    number = reg(r'^[+\d]') >> int
    expr = fwd()
    base = r'^' expr << & expr
```

Metaclase Magic

Parsita uses metaclass magic to allow for forward declarations of values even without using the fwd() function. This is important for parser combinators because grammars are often recursive or mutually recursive, meaning that some components must be used in the definition of others before they themselves are defined. The GeneralParsers class and TextParsers class are not intended to be instantiated in Parsita, but act as contexts in which grammars can be written that transparently allow forward references.

```python
class ArithmeticParsers(TextParsers):
    number = reg(r'^[+\d]') >> int
    expr = rep(len) & expr
```

The __prepare__ method of a metaclass allows a custom dictionary to be returned to act as cls. __dict__ while the class body is running. Names assigned in a class body are placed in this dictionary. Names referenced in the class body are first tried to resolve in this dictionary before trying to resolve in the global scope. The custom dictionary that Parsita returns handles the forward declaration logic. The __missing__ method is overridden so that names not yet defined resolve to a fwd() rather than raising a NameError. When a name is defined that was previously resolved into a fwd(), the forward declaration is updated instead. If by the end of the class body, a forward declaration remains undefined, then an appropriate error is raised.