Parsley Documentation

Release 1.3

Allen Short

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Parsley Tutorial Part I: Basics and Syntax

From Regular Expressions To Grammars

Parsley is a pattern matching and parsing tool for Python programmers.

Most Python programmers are familiar with regular expressions, as provided by Python's *re* module. To use it, you provide a string that describes the pattern you want to match, and your input.

For example:

```
>>> import re
>>> x = re.compile("a(b|c)d+e")
>>> x.match("abddde")
<_sre.SRE_Match object at 0x7f587af54af8>
```

You can do exactly the same sort of thing in Parsley:

```
>>> import parsley
>>> x = parsley.makeGrammar("foo = 'a' ('b' | 'c') 'd'+ 'e'", {})
>>> x("abdde").foo()
'e'
```

From this small example, a couple differences between regular expressions and Parsley grammars can be seen:

Parsley Grammars Have Named Rules

A Parsley grammar can have many rules, and each has a name. The example above has a single rule named *foo*. Rules can call each other; calling rules in Parsley works like calling functions in Python. Here is another way to write the grammar above:

```
foo = 'a' baz 'd'+ 'e'
baz = 'b' | 'c'
```

Parsley Grammars Are Expressions

Calling *match* for a regular expression returns a match object if the match succeeds or None if it fails. Parsley parsers return the value of last expression in the rule. Behind the scenes, Parsley turns each rule in your grammar into Python methods. In pseudo-Python code, it looks something like this:

```
def foo(self):
    match('a')
    self.baz()
    match_one_or_more('d')
    return match('e')

def baz(self):
    return match('b') or match('c')
```

The value of the last expression in the rule is what the rule returns. This is why our example returns 'e'.

The similarities to regular expressions pretty much end here, though. Having multiple named rules composed of expressions makes for a much more powerful tool, and now we're going to look at some more features that go even further.

Rules Can Embed Python Expressions

Since these rules just turn into Python code eventually, we can stick some Python code into them ourselves. This is particularly useful for changing the return value of a rule. The Parsley expression for this is ->. We can also bind the results of expressions to variable names and use them in Python code. So things like this are possible:

```
x = parsley.makeGrammar("""
foo = 'a':one baz:two 'd'+ 'e' -> (one, two)
baz = 'b' | 'c'
""", {})
print x("abdde").foo()
```

```
('a', 'b')
```

Literal match expressions like *'a'* return the character they match. Using a colon and a variable name after an expression is like assignment in Python. As a result, we can use those names in a Python expression - in this case, creating a tuple.

Another way to use Python code in a rule is to write custom tests for matching. Sometimes it's more convenient to write some Python that determines if a rule matches than to stick to Parsley expressions alone. For those cases, we can use *?()*. Here, we use the builtin rule *anything* to match a single character, then a Python predicate to decide if it's the one we want:

```
digit = anything:x ?(x in '0123456789') -> x
```

This rule *digit* will match any decimal digit. We need the -> *x* on the end to return the character rather than the value of the predicate expression, which is just *True*.

Repeated Matches Make Lists

Like regular expressions, Parsley supports repeating matches. You can match an expression zero or more times with '* ', one or more times with '+', and a specific number of times with '{n, m}' or just '{n}'. Since all expressions in Parsley return a value, these repetition operators return a list containing each match they made.

```
x = parsley.makeGrammar("""
digit = anything:x ?(x in '0123456789') -> x
number = digit+
""", {})
print x("314159").number()
```

```
['3', '1', '4', '1', '5', '9']
```

The *number* rule repeatedly matches *digit* and collects the matches into a list. This gets us part way to turning a string like *314159* into an integer. All we need now is to turn the list back into a string and call *int()*:

```
x = parsley.makeGrammar("""
digit = anything:x ?(x in '0123456789') -> x
number = digit+:ds -> int(''.join(ds))
""", {})
print x("8675309").number()
```

```
8675309
```

Collecting Chunks Of Input

If it seemed kind of strange to break our input string up into a list and then reassemble it into a string using *join*, you're not alone. Parsley has a shortcut for this since it's a common case: you can use <> around a rule to make it return the slice of input it consumes, ignoring the actual return value of the rule. For example:

```
x = parsley.makeGrammar("""
digit = anything:x ?(x in '0123456789')
number = <digit+>:ds -> int(ds)
""", {})
print x("11235").number()
```

```
11235
```

Here, *<digit+>* returns the string *"11235"*, since that's the portion of the input that *digit+* matched. (In this case it's the entire input, but we'll see some more complex cases soon.) Since it ignores the list returned by *digit+*, leaving the *-> x* out of *digit* doesn't change the result.

Building A Calculator

Now let's look at using these rules in a more complicated parser. We have support for parsing numbers; let's do addition, as well.

```
x = parsley.makeGrammar("""
digit = anything:x ?(x in '0123456789')
number = <digit+>:ds -> int(ds)
expr = number:left ( '+' number:right -> left + right
                   | -> left)
""", {})
print x("17+34").expr()
print x("18").expr()
```

```
51
18
```

Parentheses group expressions just like in Python. the '|' operator is like *or* in Python - it short-circuits. It tries each expression until it finds one that matches. For *"17+34"*, the *number* rule matches "17", then Parsley tries to match + followed by another *number*. Since "+" and "34" are the next things in the input, those match, and it then runs the Python expression *left + right* and returns its value. For the input *"18"* it does the same, but + does not match, so Parsley tries the next thing after |. Since this is just a Python expression, the match succeeds and the number 18 is returned.

Now let's add subtraction:

```
digit = anything:x ?(x in '0123456789')
number = <digit+>:ds -> int(ds)
expr = number:left ( '+' number:right -> left + right
                   | '-' number:right -> left - right
                   | -> left)
```

This will accept things like '5-4' now.

Since parsing numbers is so common and useful, Parsley actually has 'digit' as a builtin rule, so we don't even need to define it ourselves. We'll leave it out in further examples and rely on the version Parsley provides.

Normally we like to allow whitespace in our expressions, so let's add some support for spaces:

```
number = <digit+>:ds -> int(ds)
ws = ' '*
expr = number:left ws ('+' ws number:right -> left + right
                      |'-' ws number:right -> left - right
                      | -> left)
```

Now we can handle "17 +34", "2 - 1", etc.

We could go ahead and add multiplication and division here (and hopefully it's obvious how that would work), but let's complicate things further and allow multiple operations in our expressions – things like "1 - 2 + 3".

There's a couple different ways to do this. Possibly the easiest is to build a list of numbers and operations, then do the math.:

```
x = parsley.makeGrammar("""
number = <digit+>:ds -> int(ds)
ws = ' '*
add = '+' ws number:n -> ('+', n)
sub = '-' ws number:n -> ('-', n)
addsub = ws (add | sub)
expr = number:left (addsub+:right -> right
                   | -> left)
""", {})
print x("1 + 2 - 3").expr()
```

```
[('+', 2), ('-, 3)]
```

Oops, this is only half the job done. We're collecting the operators and values, but now we need to do the actual calculation. The easiest way to do it is probably to write a Python function and call it from inside the grammar.

So far we have been passing an empty dict as the second argument to `makeGrammar`. This is a dict of variable bindings that can be used in Python expressions in the grammar. So we can pass Python objects, such as functions, this way:

```
def calculate(start, pairs):
    result = start
    for op, value in pairs:
        if op == '+':
            result += value
        elif op == '-':
            result -= value
    return result
x = parsley.makeGrammar("""
number = <digit+>:ds -> int(ds)
ws = ' '*
add = '+' ws number:n -> ('+', n)
sub = '-' ws number:n -> ('-', n)
addsub = ws (add | sub)
expr = number:left (addsub+:right -> calculate(left, right)
                   | -> left)
""", {"calculate": calculate})
print x("4 + 5 - 6").expr()
```

```
3
```

Introducing this function lets us simplify even further: instead of using `addsub+`, we can use `addsub*`, since `calculate(left, [])` will return `left` – so now `expr` becomes:

```
expr = number:left addsub*:right -> calculate(left, right)
```

So now let's look at adding multiplication and division. Here, we run into precedence rules: should "4 * 5 + 6" give us 26, or 44? The traditional choice is for multiplication and division to take precedence over addition and subtraction, so the answer should be 26. We'll resolve this by making sure multiplication and division happen before addition and subtraction are considered:

```
def calculate(start, pairs):
    result = start
    for op, value in pairs:
        if op == '+':
            result += value
        elif op == '-':
            result -= value
        elif op == '*':
            result *= value
        elif op == '/':
            result /= value
    return result
x = parsley.makeGrammar("""
number = <digit+>:ds -> int(ds)
ws = ' '*
add = '+' ws expr2:n -> ('+', n)
sub = '-' ws expr2:n -> ('-', n)
mul = '*' ws number:n -> ('*', n)
div = '/' ws number:n -> ('/', n)

addsub = ws (add | sub)
muldiv = ws (mul | div)

expr = expr2:left addsub*:right -> calculate(left, right)
expr2 = number:left muldiv*:right -> calculate(left, right)
""", {"calculate": calculate})
```

```
print x("4 * 5 + 6").expr()
```

```
26
```

Notice particularly that `add`, `sub`, and `expr` all call the `expr2` rule now where they called `number` before. This means that all the places where a number was expected previously, a multiplication or division expression can appear instead.

Finally let's add parentheses, so you can override the precedence and write "4 * (5 + 6)" when you do want 44. We'll do this by adding a `value` rule that accepts either a number or an expression in parentheses, and replace existing calls to `number` with calls to `value`.

```
def calculate(start, pairs):
    result = start
    for op, value in pairs:
        if op == '+':
            result += value
        elif op == '-':
            result -= value
        elif op == '*':
            result *= value
        elif op == '/':
            result /= value
    return result
x = parsley.makeGrammar("""
number = <digit+>:ds -> int(ds)
parens = '(' ws expr:e ws ')' -> e
value = number | parens
ws = ' '*
add = '+' ws expr2:n -> ('+', n)
sub = '-' ws expr2:n -> ('-', n)
mul = '*' ws value:n -> ('*', n)
div = '/' ws value:n -> ('/', n)

addsub = ws (add | sub)
muldiv = ws (mul | div)

expr = expr2:left addsub*:right -> calculate(left, right)
expr2 = value:left muldiv*:right -> calculate(left, right)
""", {"calculate": calculate})

print x("4 * (5 + 6) + 1").expr()
```

```
45
```

And there you have it: a four-function calculator with precedence and parentheses.

CHAPTER 2

Parsley Tutorial Part II: Parsing Structured Data

Now that you are familiar with the basics of Parsley syntax, let's look at a more realistic example: a JSON parser.

The JSON spec on http://json.org/ describes the format, and we can adapt its description to a parser. We'll write the Parsley rules in the same order as the grammar rules in the right sidebar on the JSON site, starting with the top-level rule, 'object'.

```
object = ws '{' members:m ws '}' -> dict(m)
```

Parsley defines a builtin rule `ws` which consumes any spaces, tabs, or newlines it can.

Since JSON objects are represented in Python as dicts, and `dict` takes a list of pairs, we need a rule to collect name/value pairs inside an object expression.

```
members = (pair:first (ws ',' pair)*:rest -> [first] + rest)
          | -> []
```

This handles the three cases for object contents: one, multiple, or zero pairs. A name/value pair is separated by a colon. We use the builtin rule `spaces` to consume any whitespace after the colon:

```
pair = ws string:k ws ':' value:v -> (k, v)
```

Arrays, similarly, are sequences of array elements, and are represented as Python lists.

```
array = '[' elements:xs ws ']' -> xs
elements = (value:first (ws ',' value)*:rest -> [first] + rest) | -> []
```

Values can be any JSON expression.

```
value = ws (string | number | object | array
           | 'true'  -> True
           | 'false' -> False
           | 'null'  -> None)
```

Strings are sequences of zero or more characters between double quotes. Of course, we need to deal with escaped characters as well. This rule introduces the operator ~, which does negative lookahead; if the expression following it succeeds, its parse will fail. If the expression fails, the rest of the parse continues. Either way, no input will be consumed.

```
string = '"' (escapedChar | ~'"' anything)*:c '"' -> ''.join(c)
```

This is a common pattern, so let's examine it step by step. This will match leading whitespace and then a double quote character. It then matches zero or more characters. If it's not an `escapedChar` (which will start with a backslash), we check to see if it's a double quote, in which case we want to end the loop. If it's not a double quote, we match it using the rule `anything`, which accepts a single character of any kind, and continue. Finally, we match the ending double quote and return the characters in the string. We cannot use the <> syntax in this case because we don't want a literal slice of the input – we want escape sequences to be replaced with the character they represent.

It's very common to use ~ for "match until" situations where you want to keep parsing only until an end marker is found. Similarly, ~~ is positive lookahead: it succeed if its expression succeeds but not consume any input.

The `escapedChar` rule should not be too surprising: we match a backslash then whatever escape code is given.

```
escapedChar = '\\' (('"' -> '"')     |('\\' -> '\\')
                   |('/' -> '/')     |('b' -> '\b')
                   |('f' -> '\f')    |('n' -> '\n')
                   |('r' -> '\r')    |('t' -> '\t')
                   |('\'' -> '\'')   | escapedUnicode)
```

Unicode escapes (of the form `\u2603`) require matching four hex digits, so we use the repetition operator `{}`, which works like + or * except taking either a `{min, max}` pair or simply a `{number}` indicating the exact number of repetitions.

```
hexdigit = :x ?(x in '0123456789abcdefABCDEF') -> x
escapedUnicode = 'u' <hexdigit{4}>:hs -> unichr(int(hs, 16))
```

With strings out of the way, we advance to numbers, both integer and floating-point.

```
number = spaces ('-' | -> ''):sign (intPart:ds (floatPart(sign ds)
                                               | -> int(sign + ds)))
```

Here we vary from the json.org description a little and move sign handling up into the `number` rule. We match either an `intPart` followed by a `floatPart` or just an `intPart` by itself.

```
digit = :x ?(x in '0123456789') -> x
digits = <digit*>
digit1_9 = :x ?(x in '123456789') -> x

intPart = (digit1_9:first digits:rest -> first + rest) | digit
floatPart :sign :ds = <('.' digits exponent?) | exponent>:tail
                    -> float(sign + ds + tail)
exponent = ('e' | 'E') ('+' | '-')? digits
```

In JSON, multi-digit numbers cannot start with 0 (since that is Javascript's syntax for octal numbers), so `intPart` uses `digit1_9` to exclude it in the first position.

The `floatPart` rule takes two parameters, `sign` and `ds`. Our `number` rule passes values for these when it invokes `floatPart`, letting us avoid duplication of work within the rule. Note that pattern matching on arguments to rules works the same as on the string input to the parser. In this case, we provide no pattern, just a name: `:ds` is the same as `anything:ds`.

(Also note that our float rule cheats a little: it does not really parse floating-point numbers, it merely recognizes them and passes them to Python's `float` builtin to actually produce the value.)

The full version of this parser and its test cases can be found in the `examples` directory in the Parsley distribution.

CHAPTER 3

Parsley Tutorial Part III: Parsing Network Data

This tutorial assumes basic knowledge of writing Twisted TCP clients or servers.

Basic parsing

Parsing data that comes in over the network can be difficult due to that there is no guarantee of receiving whole messages. Buffering is often complicated by protocols switching between using fixed-width messages and delimiters for framing. Fortunately, Parsley can remove all of this tedium.

With `parsley.makeProtocol()`, Parsley can generate a Twisted IProtocol-implementing class which will match incoming network data using Parsley grammar rules. Before getting started with `makeProtocol()`, let's build a grammar for netstrings. The netstrings protocol is very simple:

```
4:spam,4:eggs,
```

This stream contains two netstrings: `spam`, and `eggs`. The data is prefixed with one or more ASCII digits followed by a `:`, and suffixed with a `,`. So, a Parsley grammar to match a netstring would look like:

```
nonzeroDigit = digit:x ?(x != '0')
digits = <'0' | nonzeroDigit digit*>:i -> int(i)

netstring = digits:length ':' <anything{length}>:string ',' -> string
```

`makeProtocol()` takes, in addition to a grammar, a factory for a "sender" and a factory for a "receiver". In the system of objects managed by the `ParserProtocol`, the sender is in charge of writing data to the wire, and the receiver has methods called on it by the Parsley rules. To demonstrate it, here is the final piece needed in the Parsley grammar for netstrings:

```
receiveNetstring = netstring:string -> receiver.netstringReceived(string)
```

The receiver is always available in Parsley rules with the name `receiver`, allowing Parsley rules to call methods on it.

When data is received over the wire, the *ParserProtocol* tries to match the received data against the current rule. If the current rule requires more data to finish matching, the *ParserProtocol* stops and waits until more data comes in, then tries to continue matching. This repeats until the current rule is completely matched, and then the *ParserProtocol* starts matching any leftover data against the current rule again.

One specifies the current rule by setting a *currentRule* attribute on the receiver, which the *ParserProtocol* looks at before doing any parsing. Changing the current rule is addressed in the *Switching rules* section.

Since the *ParserProtocol* will never modify the *currentRule* attribute itself, the default behavior is to keep using the same rule. Parsing netstrings doesn't require any rule changing, so, the default behavior of continuing to use the same rule is fine.

Both the sender factory and receiver factory are constructed when the *ParserProtocol*'s connection is established. The sender factory is a one-argument callable which will be passed the *ParserProtocol*'s Transport. This allows the sender to send data over the transport. For example:

```
class NetstringSender(object):
    def __init__(self, transport):
        self.transport = transport

    def sendNetstring(self, string):
        self.transport.write('%d:%s,' % (len(string), string))
```

The receiver factory is another one-argument callable which is passed the constructed sender. The returned object must at least have *prepareParsing()* and *finishParsing()* methods. *prepareParsing()* is called with the *ParserProtocol* instance when a connection is established (i.e. in the `connectionMade` of the *ParserProtocol*) and *finishParsing()* is called when a connection is closed (i.e. in the `connectionLost` of the *ParserProtocol*).

Note: Both the receiver factory and its returned object's *prepareParsing()* are called at in the *ParserProtocol*'s `connectionMade` method; this separation is for ease of testing receivers.

To demonstrate a receiver, here is a simple receiver that receives netstrings and echos the same netstrings back:

```
class NetstringReceiver(object):
    currentRule = 'receiveNetstring'

    def __init__(self, sender):
        self.sender = sender

    def prepareParsing(self, parser):
        pass

    def finishParsing(self, reason):
        pass

    def netstringReceived(self, string):
        self.sender.sendNetstring(string)
```

Putting it all together, the Protocol is constructed using the grammar, sender factory, and receiver factory:

```
NetstringProtocol = makeProtocol(
    grammar, NetstringSender, NetstringReceiver)
```

`The complete script is also available for download.`

Intermezzo: error reporting

If an exception is raised from within Parsley during parsing, whether it's due to input not matching the current rule or an exception being raised from code the grammar calls, the connection will be immediately closed. The traceback will be captured as a Failure and passed to the *finishParsing()* method of the receiver.

At present, there is no way to recover from failure.

Composing senders and receivers

The design of senders and receivers is intentional to make composition easy: no subclassing is required. While the composition is easy enough to do on your own, Parsley provides a function: *stack()*. It takes a base factory followed by zero or more wrappers.

Its use is extremely simple: `stack(x, y, z)` will return a callable suitable either as a sender or receiver factory which will, when called with an argument, return `x(y(z(argument)))`.

An example of wrapping a sender factory:

```
class NetstringReversalWrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def sendNetstring(self, string):
        self.wrapped.sendNetstring(string[::-1])
```

And then, constructing the Protocol:

```
NetstringProtocol = makeProtocol(
    grammar,
    stack(NetstringReversalWrapper, NetstringSender),
    NetstringReceiver)
```

A wrapper doesn't need to call the same methods on the thing it's wrapping. Also note that in most cases, it's important to forward unknown methods on to the wrapped object. An example of wrapping a receiver:

```
class NetstringSplittingWrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def netstringReceived(self, string):
        splitpoint = len(string) // 2
        self.wrapped.netstringFirstHalfReceived(string[:splitpoint])
        self.wrapped.netstringSecondHalfReceived(string[splitpoint:])

    def __getattr__(self, attr):
        return getattr(self.wrapped, attr)
```

The corresponding receiver and again, constructing the Protocol:

```
class SplitNetstringReceiver(object):
    currentRule = 'receiveNetstring'
```

```
    def __init__(self, sender):
        self.sender = sender

    def prepareParsing(self, parser):
        pass

    def finishParsing(self, reason):
        pass

    def netstringFirstHalfReceived(self, string):
        self.sender.sendNetstring(string)

    def netstringSecondHalfReceived(self, string):
        pass
```

```
NetstringProtocol = makeProtocol(
    grammar,
    stack(NetstringReversalWrapper, NetstringSender),
```

The complete script is also available for download.

Switching rules

As mentioned before, it's possible to change the current rule. Imagine a "netstrings2" protocol that looks like this:

```
3:foo,3;bar,4:spam,4;eggs,
```

That is, the protocol alternates between using `:` and using `;` delimiting data length and the data. The amended grammar would look something like this:

```
nonzeroDigit = digit:x ?(x != '0')
digits = <'0' | nonzeroDigit digit*>:i -> int(i)
netstring :delimiter = digits:length delimiter <anything{length}>:string ',' -> string

colon = digits:length ':' <anything{length}>:string ',' -> receiver.netstringReceived(
↪':', string)
semicolon = digits:length ';' <anything{length}>:string ',' -> receiver.
↪netstringReceived(';', string)
```

Changing the current rule is as simple as changing the *currentRule* attribute on the receiver. So, the `netstringReceived` method could look like this:

```
    def netstringReceived(self, delimiter, string):
        self.sender.sendNetstring(string)
        if delimiter == ':':
            self.currentRule = 'semicolon'
        else:
            self.currentRule = 'colon'
```

While changing the *currentRule* attribute can be done at any time, the *ParserProtocol* only examines the *currentRule* at the beginning of parsing and after a rule has finished matching. As a result, if the *currentRule* changes, the *ParserProtocol* will wait until the current rule is completely matched before switching rules.

The complete script is also available for download.

CHAPTER 4

Extending Grammars and Inheritance

warning Unfinished

Another feature taken from OMeta is *grammar inheritance*. We can write a grammar with rules that override ones in a parent. If we load the grammar from our calculator tutorial as `Calc`, we can extend it with some constants:

```
from parsley import makeGrammar
import math
import calc
calcGrammarEx = """
value = super | constant
constant = 'pi' -> math.pi
         | 'e' -> math.e
"""
CalcEx = makeGrammar(calcGrammar, {"math": math}, extends=calc.Calc)
```

Invoking the rule `super` calls the rule `value` in Calc. If it fails to match, our new `value` rule attempts to match a constant name.

CHAPTER 5

TermL

TermL ("term-ell") is the Term Language, a small expression-based language for representing arbitrary data in a simple structured format. It is ideal for expressing abstract syntax trees (ASTs) and other kinds of primitive data trees.

Creating Terms

```
>>> from terml.nodes import termMaker as t
>>> t.Term()
term('Term')
```

That's it! We've created an empty term, *Term*, with nothing inside.

```
>>> t.Num(1)
term('Num(1)')
>>> t.Outer(t.Inner())
term('Outer(Inner)')
```

We can see that terms are not just *namedtuple* lookalikes. They have their own internals and store data in a slightly different and more structured way than a normal tuple.

Parsing Terms

Parsley can parse terms from streams. Terms can contain any kind of parseable data, including other terms. Returning to the ubiquitous calculator example:

```
add = Add(:x, :y) -> x + y
```

Here this rule matches a term called *Add* which has two components, bind those components to a couple of names (*x* and *y*), and return their sum. If this rule were applied to a term like *Add(3, 5)*, it would return 8.

Terms can be nested, too. Here's an example that performs a slightly contrived match on a negated term inside an addition:

```
add_negate = Add(:x, Negate(:y)) -> x - y
```

CHAPTER 6

Parsley Reference

Basic syntax

`foo = ....`: Define a rule named foo.

`expr1 expr2`: Match expr1, and then match expr2 if it succeeds, returning the value of expr2. Like Python's `and`.

`expr1 | expr2`: Try to match `expr1` — if it fails, match `expr2` instead. Like Python's `or`.

`expr*`: Match `expr` zero or more times, returning a list of matches.

`expr+`: Match `expr` one or more times, returning a list of matches.

`expr?`: Try to match `expr`. Returns `None` if it fails to match.

`expr{n, m}`: Match `expr` at least `n` times, and no more than `m` times.

`expr{n}`: Match `expr` `n` times exactly.

`~expr`: Negative lookahead. Fails if the next item in the input matches `expr`. Consumes no input.

`~~expr`: Positive lookahead. Fails if the next item in the input does *not* match `expr`. Consumes no input.

`ruleName` or `ruleName(arg1 arg2 etc)`: Call the rule `ruleName`, possibly with args.

`'x'`: Match the literal character 'x'.

`<expr>`: Returns the string consumed by matching `expr`. Good for tokenizing rules.

`expr:name`: Bind the result of expr to the local variable `name`.

`-> pythonExpression`: Evaluate the given Python expression and return its result. Can be used inside parentheses too!

`!(pythonExpression)`: Invoke a Python expression as an action.

`?(pythonExpression)`: Fail if the Python expression is false, Returns True otherwise.

`expr ^(CustomLabel)`: If the expr fails, the exception raised will contain CustomLabel. Good for providing more context when a rule is broken. CustomLabel can contain any character other than "(" and ")".

Comments like Python comments are supported as well, starting with # and extending to the end of the line.

Python API

`parsley.`**`makeGrammar`**(*source*, *bindings*, *name='Grammar'*, *unwrap=False*, *extends=<function makeParser>*, *tracefunc=None*)

Create a class from a Parsley grammar.

Parameters

- **`source`** – A grammar, as a string.
- **`bindings`** – A mapping of variable names to objects.
- **`name`** – Name used for the generated class.
- **`unwrap`** – If True, return a parser class suitable for subclassing. If False, return a wrapper with the friendly API.
- **`extends`** – The superclass for the generated parser class.
- **`tracefunc`** – A 3-arg function which takes a fragment of grammar source, the start/end indexes in the grammar of this fragment, and a position in the input. Invoked for terminals and rule applications.

`parsley.`**`unwrapGrammar`**(*w*)

Access the internal parser class for a Parsley grammar object.

`parsley.`**`term`**(*termString*)

Build a TermL term tree from a string.

`parsley.`**`quasiterm`**(*termString*)

Build a quasiterm from a string.

`parsley.`**`makeProtocol`**(*source*, *senderFactory*, *receiverFactory*, *bindings=None*, *name='Grammar'*)

Create a Twisted `Protocol` factory from a Parsley grammar.

Parameters

- **`source`** – A grammar, as a string.
- **`senderFactory`** – A one-argument callable that takes a twisted `Transport` and returns a *sender*.
- **`receiverFactory`** – A one-argument callable that takes the sender returned by the `senderFactory` and returns a *receiver*.
- **`bindings`** – A mapping of variable names to objects which will be accessible from python code in the grammar.
- **`name`** – The name used for the generated grammar class.

Returns A nullary callable which will return an instance of `ParserProtocol`.

`parsley.`**`stack`**(**wrappers*)

Stack some senders or receivers for ease of wrapping.

`stack(x, y, z)` will return a factory usable as a sender or receiver factory which will, when called with a transport or sender as an argument, return `x(y(z(argument)))`.

Protocol parsing API

class `ometa.protocol.`**`ParserProtocol`**

The Twisted `Protocol` subclass used for *parsing stream protocols using Parsley*. It has two public attributes:

`sender`

After the connection is established, this attribute will refer to the sender created by the sender factory of the *`ParserProtocol`*.

`receiver`

After the connection is established, this attribute will refer to the receiver created by the receiver factory of the *`ParserProtocol`*.

It's common to also add a `factory` attribute to the *`ParserProtocol`* from its factory's `buildProtocol` method, but this isn't strictly required or guaranteed to be present.

Subclassing or instantiating *`ParserProtocol`* is not necessary; *`makeProtocol()`* is sufficient and requires less boilerplate.

class `ometa.protocol.`**`Receiver`**

`Receiver` is not a real class but is used here for demonstration purposes to indicate the required API.

`currentRule`

`ParserProtocol` examines the *`currentRule`* attribute at the beginning of parsing as well as after every time a rule has completely matched. At these times, the rule with the same name as the value of *`currentRule`* will be selected to start parsing the incoming stream of data.

`prepareParsing`(*parserProtocol*)

`prepareParsing()` is called after the *`ParserProtocol`* has established a connection, and is passed the *`ParserProtocol`* instance itself.

Parameters **`parserProtocol`** – An instance of `ProtocolParser`.

`finishParsing`(*reason*)

`finishParsing()` is called if an exception was raised during parsing, or when the *`ParserProtocol`* has lost its connection, whichever comes first. It will only be called once.

An exception raised during parsing can be due to incoming data that doesn't match the current rule or an exception raised calling python code during matching.

Parameters **`reason`** – A Failure encapsulating the reason parsing has ended.

Senders do not have any required API as *`ParserProtocol`* will never call methods on a sender.

Built-in Parsley Rules

`anything:` Matches a single character from the input.

`letter:` Matches a single ASCII letter.

`digit:` Matches a decimal digit.

`letterOrDigit:` Combines the above.

`end:` Matches the end of input.

`ws:` Matches zero or more spaces, tabs, or newlines.

`exactly(char):` Matches the character *char*.

Python Module Index

Index