Peggie

Release 0.2.1

Mar 03, 2021

Contents

0.1 Basic usage . . . 1
0.2 Grammars . . . 2
0.3 Error messages . . . 8
0.4 Parser API Reference . . . 10

Bibliography **17**

Index **19**

Peggie is a simple parsing library which may be used to parse text-based inputs given a suitable grammar.

Specifically, this parser implements a variant of the Parsing Expression Grammars (PEG) [PEG] formalism with extensions to support indentation sensitive parsing derived from the proposal by Adams and Ağacan [PEGIndent]. The parser itself uses the Packrat [Packrat] algorithm and is implemented in pure Python. This parser distinguishes itself from other Python-based parsing libraries in supporting indentation sensitivity directly rather than as a (typically only semi-supported) bolt-on to the lexing process.

0.1 Basic usage

A grammar should be defined in the typical PEG style (full details will be given later). For example, a simple grammar for matching nested lists of numbers such as `[1, 20, 300]` or `[[1, 2, []], [3, [4]]]`:

```
>>> grammar_source = r'''
...     start          <- space value space end_of_file
...     value          <- list_of_values / number
...     list_of_values <- "[" space (value space ("," space value space)*)? "]"
...     number         <- r"[0-9]+"
...     space          <- r"\s*"
...     end_of_file    <- !.
... '''
```

The first rule in the grammar is treated as the start rule. Quoted strings match literals, and quoted strings proceeded by an `r` are treated as Python-style regular expressions (see `re`[1]). A dot (`.`) matches any single character. The `?`, `*` and `+` operators match zero-or-one, zero-or-more and one-or-more instances of a pattern respectively. The `!` operator is a negative lookahead operator which only matches when the pattern after it does not – in this case matching only when no next value is present, i.e. the end of the file.

Next, the grammar must be compiled using *`compile_grammar()`* (page 11):

```
>>> from peggie import compile_grammar
>>> grammar = compile_grammar(grammar_source)
```

The compiled grammar may then be used by a *`Parser`* (page 10) object to parse strings according to the grammar. For example:

```
>>> from peggie import Parser
>>> parser = Parser(grammar)
>>> parse_tree = parser.parse("[1, 2, [3, [4]], []]")
```

Parse trees may be conveniently transformed into a more useful representation (or indeed evaluated into some final result) using a *`ParseTreeTransformer`* (page 14). A *`ParseTreeTransformer`* (page 14) subclass should be constructed which defines a transformation to be carried out for each rule in the grammar. For example, we can write a transformer which assembles a Python list from the parse tree like so:

```
>>> from peggie import ParseTreeTransformer

>>> class ListTransformer(ParseTreeTransformer):
...     def number(self, parse_tree, result):
...         return int(result)
...
...     def list_of_values(self, parse_tree, result):
...         _open, _sp, body, _close = result
...         if body is None:
...             return []
...         else:
...             first, _sp, rest = body
...             out = [first]
```

(continues on next page)

[1] https://docs.python.org/3/library/re.html#module-re

(continued from previous page)

```
...             for _comma, _sp1, value, _sp2 in rest:
...                 out.append(value)
...             return out
...
...     def start(self, parse_tree, result):
...         _sp1, value, _sp2, _eof = result
...         return value

>>> transformer = ListTransformer()
>>> transformer.transform(parse_tree)
[1, 2, [3, [4]], []]
```

When a *ParseTreeTransformer* (page 14) subclass' *ParseTreeTransformer.transform()* (page 15) method is called, the parse tree is traversed in a bottom-up fashion, and for each rule application encountered attempts to call the method named after the rule. These methods are passed two arguments: the *ParseTree* (page 13) corresponding with the matched subtree and the transformed result of transforming the children of the parse-tree at this point. The method should return the transformed value.

When no matching method is found, the default implementation transforms the parse tree into simple Python types automatically. For example, regular expression matches are transformed into strings, `?` into a value or None and concatenations, `*` and `+` into lists.

0.2 Grammars

The PEG grammar syntax accepted by this module is defined using its own syntax below:

```
# Hierarchical syntax
grammar            <- spacing definition+ end_of_file
definition         <- identifier LEFTARROW expression
expression         <- sequence (SLASH sequence)*
sequence           <- lookahead_prefix*
lookahead_prefix   <- (AND / NOT)? arity_suffix
arity_suffix       <- indent_rule_prefix (QUESTION / STAR / PLUS)?
indent_rule_prefix <- indent_rule? primary
primary            <- OPEN expression CLOSE
                    / literal / class / DOT
                    / identifier !LEFTARROW
class              <- '[' (!']' range)+ ']' spacing
range              <- char '-' char / char

# Lexical syntax
identifier <- r'[a-zA-Z_][a-zA-Z0-9_]*' spacing
literal    <- 'r'? "'" (!"'" char)* "'" spacing
            / 'r'? '"' (!'"' char)* '"' spacing
char       <- '\\' .
            / !'\\' .

indent_rule <- r'@(\*|=|>=|>)' spacing

# Symbols
LEFTARROW  <- '<-' spacing
SLASH      <- '/' spacing
AND        <- '&' spacing
NOT        <- '!' spacing
QUESTION   <- '?' spacing
STAR       <- '*' spacing
PLUS       <- '+' spacing
OPEN       <- '(' spacing
CLOSE      <- ')' spacing
```

(continues on next page)

(continued from previous page)

```
DOT         <- '.' spacing

# Whitespace
spacing       <- (space / comment)*
comment       <- '#' (!end_of_line .)* end_of_line
space         <- [ \t] / end_of_line
end_of_line   <- r'\r\n|\n\r|\n|\r'
end_of_file   <- !.
```

The syntax is based on Ford's [PEG] grammars. In summary:

- `name <- pattern` defines a rule called `name` matching the provided pattern.
- `rule_name` is a pattern matching whatever pattern is defined for the provided rule.
- `"foo"` is a pattern matching the string `foo`.
- `[a-z]` is a pattern matching the characters from `a` to `z`.
- `.` is a pattern which matches any single character.
- `pattern+` is a pattern matching one or more instances of the provided pattern.
- `pattern*` is a pattern matching zero or more instances of the provided pattern.
- `pattern?` is a pattern matching zero or one instances of the provided pattern.
- `pattern_a pattern_b` is a pattern matching a string matched by `pattern_a` followed by a string matched by `pattern_b`.
- `patern_a / pattern_b` is a pattern which matches `pattern_a` if that mattern matches, or `pattern_b` if `pattern_a` does not match and `pattern_b` does. (Note that there is no ambiguity if both `pattern_a` and `pattern_b` match).
- `!pattern` is a pattern which matches, without consuming any input, only when the provided pattern does not match.
- `&pattern` is a pattern which matches, without consuming any input, only when the provided pattern matches.

Comments are started with a hash (`#`) and run to the end of the line.

0.2.1 Regular expressions

In addition, Python-style regular expressions may also be included in `r` prefixed strings, eg. `r"foo+"`. This is partly a convenience to enable slightly more concise descriptions of some patterns and also a potential performance boost since longer patterns may be matched more quickly by Python's `re`[2] matcher.

Note: The `re.DOTALL`[3] flag is implicitly set for all patterns meaning `.` in a regular expression will also match newlines, which is not the default Python behaviour.

0.2.2 Indentation sensitivity

This parser also extends Ford's grammars to support indentation-sensitive languages using an extension loosely based on by that of Adams and Ağacan [PEGIndent].

[2] https://docs.python.org/3/library/re.html#module-re
[3] https://docs.python.org/3/library/re.html#re.DOTALL

Indentation prefixes

By default, all patterns are indentation insensitive but may be made indentation sensitive by prefixing them with `@=`, `@>=` or `@>`. These add the additional requirement that the pattern must be on a line whose indentation is equal, greater-or-equal or greater (respectively) than the current reference indentation.

Within a concatenation, the indentation level at the first character to be matched is used as the reference indentation. For example, the grammar:

```
start       <- space foo_bar_baz end_of_file

foo_bar_baz <- "foo"
               newline
               @="bar"
               newline
               @="baz"
               newline

space       <- r"[ \t]*"
newline     <- r"[ \t]*\n[ \t]*"
end_of_file <- !.
```

Will match any string containing `foo`, `bar` and `baz` on separate lines, but all indented to the same level.

Note: An indentation constraint may be applied to a pattern which matches any part of the line, not just the start. However, the indentation level for the line as a whole is always checked, not the number of characters into the line that particular pattern appears. For example, the following would match exactly the same strings as the example above:

```
foo_bar_baz <- "foo"
              newline
              @="bar"
              newline
              "baz"
              @=newline  # NB: @= is applied to the newline here!
```

Replacing `@=` with `@>=` will match `foo`, `bar` and `baz` so long as `bar` and `baz` are indented at least as much as `foo`, but may be indented more so. However, `bar` and `baz` may be intended arbitrarily with respect to eachother because indentation is only checked with respect to the first character in the concatenation. So, for example:

```
foo
    bar
        baz
```

And:

```
foo
bar
    baz
```

And:

```
foo
    bar
baz
```

Will all match but the following will *not*:

```
     foo
bar
     baz
```

Simillarly, replacing `@=` with `@>` requires that `bar` and `baz` have greater indentation than `foo` but, again, the relative indentation of `bar` and `baz` to each other is not constrained.

Note: We could have placed `@=` in front of the first pattern in `foo_bar_baz` but since the first pattern has, by definition, the same indentation as the start of the cancatnation, this has no effect.

If we had placed `@>` in front of the first pattern, the pattern will never match because we are requiring that this pattern has a greater indentation than itself.

Note: Any pattern without an indentation prefix has no indentation constraint. Where it is useful for reasons of readability, however, this can be made explicit with the `@*` prefix.

When used with `*` or `+`, the indentation constraints apply with respect to the indentation of the start of the first match. For example the grammar:

```
start       <- space foos end_of_file

foos <- @=("foo" newline)+

space       <- r"[ \t]*"
newline     <- r"[ \t]*\n[ \t]*"
end_of_file <- !.
```

Matches any number of lines containing `foo` indented to the same level. Likewise replacing `@=` with `@>=` will match any number of lines containing `foo` where all lines must be indented at least as much as the first (but with no other relative constraint on indentation).

When indentation prefixes are used in other contexts (for example in `@= "foo" ?`), they have no effect.

Note: The location of the perentheses used in this example are important. For example:

```
foos <- (@="foo" newline)+
```

Will match any number of lines containing foo with *any* indentation because the indentation prefix applies to `"foo"` within the concatenation `(@="foo" newline)`.

Indented block syntax example

For a simple example of indentation prefixes being used to specify a python-style intended block syntax, see the following:

```
start          <- space stmt space end_of_file

stmt           <- call_stmt / block_stmt
call_stmt      <- r"[a-z]+\(\)" newline
block_stmt     <- "block:" newline
                  @>(
                    @=stmt+
                  )

space          <- r"[ \t]*"
newline        <- r"[ \t]*\n[ \t]*"
end_of_file    <- !.
```

This grammar accepts strings of the form:

```
block:
    foo()
    bar()
    block:
        baz()
    block:
        qux()
    quo()
```

Breaking down the `block_stmt`:

- `"block:" newline`: A block starts with `block:` and a newline...
- `@>( ... )` ...followed by something indented more than `block:` was...
- `@=stmt+` ...specifically one or more `stmt`s, all indented to the same level as the first `stmt` matched.

Indentation prefixes and empty matches

When an indentation requirement is applied to a pattern which can match the empty string, the indentation level is still checked and the empty string will not be matched if the indentation level at the current parse position is incorrect. For example, in the grammar:

```
start       <- space foo_bar_baz end_of_file

foo_bar_baz <- "foo" newline
               @="" "bar" newline
               @="" "baz" newline

space       <- r"[ \t]*"
newline     <- r"[ \t]*\n[ \t]*"
end_of_file <- !.
```

Here we placed the indentation restriction on `""` (which matches the empty string), this empty string must be matched at the same indentation level as the start of the concatenation. Consequently this grammar matches the same strings as our earlier example (i.e. with `foo`, `bar` and `baz` having the same indentation).

There is, however, an exception to this rule. When a `?` or `*` pattern within a concatenation fails to match the indentation requirement, the pattern is deemed to have matched the empty string rather than failing to match. To see why this special case is useful, consider the following grammar:

```
start          <- space stmt space end_of_file

stmt           <- call_stmt / try_catch_stmt
call_stmt      <- r"[a-z]+\(\)" newline
try_catch_stmt <- "try" stmt_block
                  @=("catch" stmt_block)?

stmt_block     <- ":" newline
                  @>(
                    @=stmt+
                  )

space          <- r"[ \t]*"
newline        <- r"[ \t]*\n[ \t]*"
end_of_file    <- !.
```

This grammar describes a language matching strings like:

```
foo()
```

And:

```
try:
    foo()
    bar()
    baz()
```

And:

```
try:
    foo()
    bar()
catch:
    baz()
    qux()
```

In this case, the 'catch' part of the `try_catch_stmt` rule uses `@=` to ensure that 'catch' has the same indentation as the opening 'try'. However, this also helps us disambiguate the following case:

```
try:
    foo()
    try:
        bar()
catch:
    baz()
```

Specifically, the 'catch' here belongs to the outer try/catch block, not the inner one. In this instance while parsing the inner `try_catch_stmt` rule, we fail to match the `catch:` line because it has the wrong indentation. However, as a special case because this is enclosed in a `?` pattern, rather than throwing a parse failure we act as if the `?` matched the empty string, leaving the `catch:` to be parsed by the outer `try_catch_stmt` rule.

Note: If, for some reason, you wished to override this special case behaviour (and always fail to parse on indentation failure, this can be forced by turning the `?` pattern into something else, for example a concatenation with the empty string like:

```
try_catch_stmt <- "try" stmt_block
                  @=(("catch" stmt_block)? "")  # Will not match empty on
                                                # indentation failure!
```

0.2.3 Well-formedness

Grammars must adhere to the following well-formedness rules:

- At least one rule (the start rule) must be defined.
- Left recursion, whether direct, indirect or hidden, is not allowed.
- The `*` and `+` repetition operators may not be used on patterns which match the empty string.
- All rules used in patterns must have exactly one definition.

If a grammar does not meet these criteria, a *GrammarCompileError* (page 12) will be thrown during grammar compilation.

0.2.4 Unprocessed input

In common with other PEG parsers, the parser will not throw an error if the whole input is not parsed. To ensure the whole input is matched by the grammar, use the `!.` idiom which matches only at the end of the input.

0.3 Error messages

Upon parse failures a *ParseError* (page 10) exception will be thrown.

When cast to a `str`[4], these exceptions produce a descriptive parse failure. For example, given the following grammar:

```
>>> grammar = compile_grammar('''
...     summation   <- value space+ ("+" space+ summation)? end_of_file
...     value       <- r"[0-9]+"
...     space       <- [ \t]
...     end_of_file <- !.
... ''')
```

This matches simple strings such as `123` or `1 + 2 + 3`. If a non-matching string is provided, however, an error such as `1 + 2 - 3` the following will be produced:

```
>>> from peggie import ParseError

>>> try:
...     Parser(grammar).parse("1 + 2 - 3")
... except ParseError as e:
...     print(str(e))
At line 1 column 7:
    1 + 2 - 3
          ^
Expected '\\+' or end_of_file or space
```

Notice that the error suggests rules (`end_of_file` and `space`) and strings (`\\+`) which would have allowed parsing to continue.

Note: All string literals in a grammar are internally converted to Python regular expressions and therefore are subject to `re.escape()`[5]. As an unfortunate result, the strings shown in these error messages also contain extra escaping.

Likewise, if we try to parse an empty string:

```
>>> try:
...     Parser(grammar).parse("")
... except ParseError as e:
...     print(str(e))
At line 1 column 1:

    ^
Expected summation
```

Note that the error refers to the highest-level rule which applies, i.e. `summation` rather than `value` in this case. In complex grammars, this is usually what you'd want.

These default messages simply use the bare rule names and regular expression strings from the grammar to describe what is expected. This is not always the most user-friendly option. As a result, alternative names for these elements using the `ParseError.expr_explanations` `dict`[6] attribute. For example:

```
>>> from peggie import RuleExpr, RegexExpr

>>> expr_explanations = {
...     RuleExpr("end_of_file"): "<end of file>",
```

(continues on next page)

[4] https://docs.python.org/3/library/stdtypes.html#str
[5] https://docs.python.org/3/library/re.html#re.escape
[6] https://docs.python.org/3/library/stdtypes.html#dict

(continued from previous page)

```
...     RuleExpr("space"): "<space>",
...     RuleExpr("value"): "<number>",
...     RuleExpr("summation"): "<number>",
...     RegexExpr.literal("+"): "'+'",
... }

>>> try:
...     Parser(grammar).parse("1 + 2 - 3")
... except ParseError as e:
...     e.expr_explanations = expr_explanations
...     print(str(e))
At line 1 column 7:
    1 + 2 - 3
          ^
Expected '+' or <end of file> or <space>

>>> try:
...     Parser(grammar).parse("")
... except ParseError as e:
...     e.expr_explanations = expr_explanations
...     print(str(e))
At line 1 column 1:

    ^
Expected <number>
```

As a further refinement, we can filter out more unhelpful suggestions when others are available. For instance in the first example above the suggestion of `<space>`, while valid, is not especially useful in this case. We can suppress such suggestions using the `ParseError.last_resort_exprs` `set`[7] attribute:

```
>>> last_resort_exprs = {
...     RuleExpr("space"),
... }

>>> try:
...     Parser(grammar).parse("1 + 2 - 3")
... except ParseError as e:
...     e.expr_explanations = expr_explanations
...     e.last_resort_exprs = last_resort_exprs
...     print(str(e))
At line 1 column 7:
    1 + 2 - 3
          ^
Expected '+' or <end of file>
```

If our suppressed expression is the only suggestion, however, it will still be included. For example:

```
>>> try:
...     Parser(grammar).parse("1 +2")
... except ParseError as e:
...     e.expr_explanations = expr_explanations
...     e.last_resort_exprs = last_resort_exprs
...     print(str(e))
At line 1 column 4:
    1 +2
       ^
Expected <space>
```

[7] https://docs.python.org/3/library/stdtypes.html#set

0.4 Parser API Reference

The following sections provide reference documentation for the parser's API.

0.4.1 Parser

Parsing is performed by the `Parser` (page 10) class:

class Parser(*grammar: peggie.parser.Grammar*)
A parser.

Strings are parsed using the `parse()` (page 10) method.

Parameters

grammar [`Grammar` (page 11)] The grammar describing the language to be parsed.

parse(*string: str*) → peggie.parser.Rule
Parse a string, returning the `ParseTree` (page 13) if successful or raising a `ParseError` (page 10) if not.

When a non-matching string is provided, a `ParseError` (page 10) will be thrown:

exception ParseError(*line: int, column: int, snippet: str, expectations: Set[Tuple[Union[peggie.parser.RuleExpr, peggie.parser.RegexExpr], FrozenSet[Optional[peggie.parser.AbsoluteIndentation]]]], expr_explanations: Mapping[Union[peggie.parser.RuleExpr, peggie.parser.RegexExpr], Optional[str]] = <factory>, last_resort_exprs: Set[Union[peggie.parser.RuleExpr, peggie.parser.RegexExpr]] = <factory>, just_indentation: bool = False*)
Thrown when parsing fails.

Parameters

line [int] One-indexed line number where the error occurred.

column [int] One-indexed column number where the error occurred.

snippet [str] The contents of the offending line.

expectations [{(rule_or_regex, {AbsoluteIndentation or None, ...}), ...}] The set of `RuleExpr` (page 12) and `RegexExpr` (page 12) expressions which the parser would have accepted at this point, along any required differences in indentation.

expr_explanations [{rule_or_regex: str or None, ...}] Error message customization parameter. By default expected `Rule` (page 14) are shown as their rule name and `Regex` (page 13) as their pattern (in quotes). This representation may be overridden by a string entry in this dictionary. Alternatively, an expression may be suppressed from the explanation by providing None. Default = {}.

last_resort_exprs [{rule_or_regex, ...}] Error message customization parameter. A set of `Rule` (page 14) or `Regex` (page 13) expressions which are ordinarily suppressed from explanations except when these are the only matching expressions in which case they are included. Default = {}.

just_indentation [bool] Error message customization parameter. If True, when at least one expression with an indentation requirement is present, non-indentation related expectations are suppressed. Default = False.

just_indentation = False

explain(*expr_explanations: Optional[Mapping[Union[peggie.parser.RuleExpr, peggie.parser.RegexExpr], Optional[str]]] = None, last_resort_exprs: Optional[Set[Union[peggie.parser.RuleExpr, peggie.parser.RegexExpr]]] = None, just_indentation: Optional[bool] = None*) → str

Return a human-readable string describing the expected next values.

Parameters

expr_explanations [{rule_or_regex: str or None, ...}] See `expr_explanations`.

last_resort_exprs [{rule_or_regex, ...}] See `last_resort_exprs`.

just_indentation [bool] See *just_indentation* (page 10).

Indentation mismatches are recorded in the `ParseError.expectations` attribute in *AbsoluteIndentation* (page 11) objects:

class AbsoluteIndentation

An absolute indentation requirement.

start_indentation

The indentation level the rule is relative to.

indentation

The required (unmet) *RelativeIndentation* (page 13) rule.

If the *Parser* (page 10) is supplied with a *Grammar* (page 11) which is not well-formed, the following exceptions may be thrown at runtime during parsing:

exception GrammarError

Thrown when a problem is encountered with the grammar during parsing.

exception RepeatedEmptyTermError

Bases: `peggie.parser.GrammarError`

Thrown when a grammar contains a rule which repeats a term which matches the empty string.

exception LeftRecursionError

Bases: `peggie.parser.GrammarError`

Thrown when a grammar contains a direct/indirect/hidden left-recursive rule.

exception UndefinedRuleError

Bases: `peggie.parser.GrammarError`

Thrown when a grammar contains a reference to an undefined rule.

0.4.2 Grammars

Grammars are defined by a *Grammar* (page 11) object:

class Grammar(*rules: Mapping[str, peggie.parser.Expr], start_rule: str = 'start'*)

A PEG grammar description.

rules = None

The expression for each rule in the grammar.

start_rule = 'start'

Name of the starting rule in *rules* (page 11).

In the typical case, *Grammar* (page 11) compiled from a textual PEG grammar description using *compile_grammar()* (page 11):

compile_grammar(*grammar_spec: str*) → peggie.parser.Grammar

Parse and compile a PEG grammar from a string.

Throws a *GrammarCompileError* (page 12) exception if the supplied grammar is not well formed or a *ParseError* (page 10) if the grammar specification contains a syntax error.

When a grammar is compiled, it is automatically checked for well-formedness and the following exceptions are produced if the grammar fails these checks:

exception GrammarCompileError
Thrown during grammar compilation if the grammar is not valid.

exception RuleDefinedMultipleTimesError
Bases: `peggie.grammar_compiler.GrammarCompileError`

Name redefined in a grammar.

exception GrammarNotWellFormedError
Bases: `peggie.grammar_compiler.GrammarCompileError`

The grammar is not well formed.

Grammars can also be constructed 'by hand' from *Expr* (page 12) objects. These objects represent expressions in a grammar (e.g. regular expressions or concatenations of other expressions).

class Expr
An expression in a PEG grammar. Abstract base class.

class EmptyExpr(*indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Match the empty string.

class RegexExpr(*pattern: Union[Pattern[str], str], indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Match a compiled `re`[8] regular expression. If a string is provided, it will be compiled into a regular expression with the `re.DOTALL`[9] flag set.

classmethod literal(*text: str*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*) → RegexExprT
Return a *RegexExpr* (page 12) matching a string literal.

class RuleExpr(*name: str*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Match a named rule in the grammar.

class AltExpr(*exprs: Tuple[peggie.parser.Expr, ...], indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Prioritised alternation: matches first matching expression.

class ConcatExpr(*exprs: Tuple[peggie.parser.Expr, ...], indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Concatenation of several expressions.

class MaybeExpr(*expr: peggie.parser.Expr*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Match an expression, or an empty string.

class StarExpr(*expr: peggie.parser.Expr*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Kleene star: matches 0-or-more repetitions of an expression.

class PlusExpr(*expr: peggie.parser.Expr*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
'Kleene plus': match 1-or-more repetitions of an expression.

class LookaheadExpr(*expr: peggie.parser.Expr*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Negative lookahead. Matches when expression does not, without consuming input.

class PositiveLookaheadExpr(*expr: peggie.parser.Expr*, *indentation: peggie.parser.RelativeIndentation = <RelativeIndentation.any: '@*'>*)
Positive lookahead: match, but don't consume an expression.

[8] https://docs.python.org/3/library/re.html#module-re
[9] https://docs.python.org/3/library/re.html#re.DOTALL

Indentation requirements are indicated using the following enumerated type:

class RelativeIndentation
Describes the required relative indentation level of a pair of tokens.

any = '@*'

equal = '@='

greater_or_equal = '@>='

greater = '@>'

When a *Grammar* (page 11) is constructed by hand it is not automatically checked for well-formedness. This check can be performed manually using *Grammar.is_well_formed()* (page 13).

Grammar.**is_well_formed**() → peggie.parser.GrammarWellFormedness
Is this grammar well-formed? That is, is it free from missing rules, (direct/indirect/hidden) left recursive rules and iteration of empty patterns?

The well-formedness check result will come in the form of one of the following:

class GrammarWellFormedness
Base class of result from a well-formedness test.

class WellFormed
Bases: peggie.parser.GrammarWellFormedness

The grammar is well formed.

class UndefinedRule(*name: str*)
Bases: peggie.parser.GrammarWellFormedness

The grammar refers to an undefined rule.

class LeftRecursion(*name: str*)
Bases: peggie.parser.GrammarWellFormedness

The grammar contains a left-recursive rule.

class RepeatedEmptyTerm(*expr: peggie.parser.Expr*)
Bases: peggie.parser.GrammarWellFormedness

The grammar contains a repeated empty term.

0.4.3 Parse trees

The result of parsing a string is a *ParseTree* (page 13). This tree's structure mimics that of the grammar and consists of a hierarchy of the following classes:

class ParseTree
A parse tree generated by the parser. Base class.

iter_children() → Iterable[peggie.parser.ParseTree]
Iterate over child parse trees.

class Empty(*offset: int*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *EmptyExpr* (page 12) is parsed.

offset = None
The character offset where this empty was matched.

class Regex(*string: str*, *start: int*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *RegexExpr* (page 12) is parsed.

string = None
The matched string.

start = None
The offset at which this string appears in the parser input.

end
The offset just beyond the end of this string as it appears in the parser input.

class Rule(*name: str*, *value: peggie.parser.ParseTree*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *RuleExpr* (page 12) is parsed.

class Alt(*value: peggie.parser.ParseTree*, *choice_index: int*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *AltExpr* (page 12) is parsed.

class Concat(*values: Tuple[peggie.parser.ParseTree, ...]*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *ConcatExpr* (page 12) is parsed.

class Maybe(*value: Optional[peggie.parser.ParseTree]*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *MaybeExpr* (page 12) is parsed.

value = None
If the expression matched, its *ParseTree* (page 13), otherwise None.

class Star(*values: Tuple[peggie.parser.ParseTree, ...]*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *StarExpr* (page 12) is parsed.

class Plus(*values: Tuple[peggie.parser.ParseTree, ...]*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *PlusExpr* (page 12) is parsed.

class Lookahead(*offset: int*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *LookaheadExpr* (page 12) is parsed.

offset = None
The character offset where this lookahead was matched.

class PositiveLookahead(*offset: int*)
Bases: peggie.parser.ParseTree

ParseTree (page 13) produced when a *PositiveLookaheadExpr* (page 12) is parsed.

offset = None
The character offset where this positive lookahead was matched.

0.4.4 Transformers

To assist in the transformation of a parse tree into a useful data structure, the *ParseTreeTransformer* (page 14) base class is provided:

class ParseTreeTransformer
By default, this transformer will produce a representation containing a hierarchy of lists containing *Regex* (page 13) or string objects and None.

Transformations may be customised by defining methods with the name of the rule to be transformed. These will be called with the *ParseTree* (page 13) of the matched rule along with the transformed value of the

body of the rule. Methods should return the newly transformed body. If no matching method is defined, the *_default()* (page 15) method will be called. In the event that a method name is a Python reserved word, a method name should be given a "_" suffix.

Methods named `<rule_name>_enter` will be called (if defined) before the children of a rule are transformed.

The default transformation for Regex values is to return the matched string. This can be changed by overriding *_transform_regex()* (page 15).

The default transformation for Empty, Lookahead and PositiveLookahead values is to return None. This can be changed by overriding *_transform_empty()* (page 15), *_transform_lookahead()* (page 15) or *_transform_positive_lookahead()* (page 15) methods.

transform(*tree: peggie.parser.ParseTree*) → Any
Transform the provided parse tree with this transformer.

_transform_regex(*regex: peggie.parser.Regex*) → Any
The default transformation for Regex.

This default implementation returns the matched string but this method may be overridden to return custom values instead.

_transform_empty(*empty: peggie.parser.Empty*) → Any
The default transformation for Empty.

This default implementation returns None.

_transform_lookahead(*lookahead: peggie.parser.Lookahead*) → Any
The default transformation for Lookahead.

This default implementation returns None.

_transform_positive_lookahead(*positive_lookahead: peggie.parser.PositiveLookahead*) → Any
The default transformation for PositiveLookahead.

This default implementation returns None.

_default(*tree: peggie.parser.ParseTree*, *transformed_children: Any*) → Any
The default transformation for rules.

Bibliography

[PEG] Bryan Ford: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: Symposium on Principles of Programming Languages, January 14-16, 2004.

[Packrat] Bryan Ford: Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In: International Conference on Functional Programming, October 4-6, 2002.

[PEGIndent] Michael D. Adams; Ömer S Ağacan: Indentation-sensitive parsing for Parsec. In: ACM SIGPLAN Notices, September 2014.

Index

Symbols

_default() (*ParseTreeTransformer method*), 15
_transform_empty() (*ParseTreeTransformer method*), 15
_transform_lookahead() (*ParseTreeTransformer method*), 15
_transform_positive_lookahead() (*ParseTreeTransformer method*), 15
_transform_regex() (*ParseTreeTransformer method*), 15

A

AbsoluteIndentation (*class in peggie*), 11
Alt (*class in peggie*), 14
AltExpr (*class in peggie*), 12
any (*RelativeIndentation attribute*), 13

C

compile_grammar() (*in module peggie*), 11
Concat (*class in peggie*), 14
ConcatExpr (*class in peggie*), 12

E

Empty (*class in peggie*), 13
EmptyExpr (*class in peggie*), 12
end (*Regex attribute*), 14
equal (*RelativeIndentation attribute*), 13
explain() (*ParseError method*), 10
Expr (*class in peggie*), 12

G

Grammar (*class in peggie*), 11
GrammarCompileError, 12
GrammarError, 11
GrammarNotWellFormedError, 12
GrammarWellFormedness (*class in peggie*), 13
greater (*RelativeIndentation attribute*), 13
greater_or_equal (*RelativeIndentation attribute*), 13

I

indentation (*AbsoluteIndentation attribute*), 11
is_well_formed() (*Grammar method*), 13
iter_children() (*ParseTree method*), 13

J

just_indentation (*ParseError attribute*), 10

L

LeftRecursion (*class in peggie*), 13
LeftRecursionError, 11
literal() (*peggie.RegexExpr class method*), 12
Lookahead (*class in peggie*), 14
LookaheadExpr (*class in peggie*), 12

M

Maybe (*class in peggie*), 14
MaybeExpr (*class in peggie*), 12

O

offset (*Empty attribute*), 13
offset (*Lookahead attribute*), 14
offset (*PositiveLookahead attribute*), 14

P

parse() (*Parser method*), 10
ParseError, 10
Parser (*class in peggie*), 10
ParseTree (*class in peggie*), 13
ParseTreeTransformer (*class in peggie*), 14
peggie (*module*), 1
Plus (*class in peggie*), 14
PlusExpr (*class in peggie*), 12
PositiveLookahead (*class in peggie*), 14
PositiveLookaheadExpr (*class in peggie*), 12

R

Regex (*class in peggie*), 13
RegexExpr (*class in peggie*), 12
RelativeIndentation (*class in peggie*), 13
RepeatedEmptyTerm (*class in peggie*), 13
RepeatedEmptyTermError, 11
Rule (*class in peggie*), 14
RuleDefinedMultipleTimesError, 12
RuleExpr (*class in peggie*), 12
rules (*Grammar attribute*), 11

S

Star (*class in peggie*), 14
StarExpr (*class in peggie*), 12
start (*Regex attribute*), 14
start_indentation (*AbsoluteIndentation attribute*), 11
start_rule (*Grammar attribute*), 11
string (*Regex attribute*), 13

T

transform() (*ParseTreeTransformer method*), 15

U

UndefinedRule (*class in peggie*), 13
UndefinedRuleError, 11

V

value (*Maybe attribute*), 14

W

WellFormed (*class in peggie*), 13