A Parse Toolkit Documentation

Release 0.6.2

Kay-Uwe (Kiwi) Lorenz

Sep 27, 2017

Contents

aPTK is a Parse Toolkit. It is useful to write documented grammars similar to BNF grammar language.

Typically you would use it like this:

```
from aptk import *

class AdditionGrammar(Grammar):
    '''This is the grammar of a simple addition.

    <addition> := <operand> <.ws> "+" <.ws> <operand>
    <ws>       := \s*
    <operand>  := \d+
    '''

class AdditionActions(ParseActions):
    def make_operand(self, p, lexem):
        return int(str(lexem))

    def make_addition(self, p, lexem):
        return lexem[0].ast + lexem[1].ast

tree = parse("5 + 4", AdditionGrammar)
result = ast("5 + 4", AdditionGrammar, AdditionActions)
```

The most interesting on the grammars derived from `BaseGrammar` is that they are compiled at compile-time of your python module. This is possible due to some python voodoo with metaclasses in `grammar`.

CHAPTER 1

aPTK Tutorial

Here shall be a tutorial.

CHAPTER 2

aPTK Grammar Syntax

Syntax of aPTK Grammars are oriented on BNF and a bit on Perl6 grammars.

A grammar consists of production rules and statements. Statements influence parsing and/or interpretation of the parsed. Optionally you may add assertions, to prove, that your rules meet your expectations.

General

All rules and statements have to start on same indentation level. If you want to continue a rule or statement on next line, you can do so by indenting next line a bit more than the line, where your rule or statement started:

```
:grammar grammar

grammar := [ <statement>
           | <production-rule>
           | <test-assertion>
           ]*

ws      := \s+
```

Lines abore define new grammar named "grammar" and define first rule, the default entry point of the grammar.

Statements

A statement is a line, which starts with a ":". There are following statements supported:

`:grammar <name> [ extends [ <grammar-name> ]+ ]?` Define a new grammar named <*name*>, which extends grammars <*grammar-name*>. If you do not pass <*grammar-name*>, it defaults to `Grammar`

This statement is available in contexts where you not have predefined a grammar, as for example if you define your grammar as python class.

Examples for `:grammar`:

```
:grammar very-simple-grammar

:grammar another-grammar extends very-simple-grammar

:grammar x extends aptk.BaseGrammar
```

:parse-actions [<name> <python-name>]+ Define a ParseActions class (or module), which can be later used in tests (or simply referenced by its name, when creating a parser):

```
:parse-actions my_module.MyParseAction
```

This statements imports parse-actions into your grammar, that you can make use of it in test assertions:

```
<some-rule> ~~ "some string" -MyParseAction-> some ast
```

:parse-action-map [<name> <method-name>]+ Map <*string*> to <*method-name*>, which is expected to exist in parse-actions passed to parser. After mapping <*string*> to <*method-name*>, you can use <*string*>= as operator in production rule, to assign a parse-action:

```
:parse-action-map
    "foo" make_foo

some-rule  foo= "some right-hand side"
```

These parse-action-map become handy, if there is an action which is done for more than one capture.

:sigspace [<non-terminal> | <terminal>] Set rule for significant whitespace.

:args-of <custom-rule-name> [[<arg-flag>]+ | <callable>] Specify how args of a complex custom rule are parsed:

```
arg-flag := "string" | "capturing" | "non-capturing" | "regex" |
               "raw" | "slashed-regex" | "char-class"

callable := <module-name> "." <>
```

Production Rules

A production rule consists of a name, an operator, and a statement on the right hand side:

```
production-rule := <token-def> | <rule-def>

token-name := "{" <name> "}" | <name>
rule-name  := "<" <name> ">" | <name>
```

You can have following operators:

This is the formal definition of production rules, here follow detailed explanations with examples:

- *Tokens*

Tokens

Tokens are a special form of production rules:

```
token-def  :-  <token-name> "=" <token-value>
```

<token-name> Can be any name. All characters except whitespace, with two limitations:

- `<token-name>` must not start and end with a ":" or be enclosed by "{:" and ":}
- `<token-name>` may be optionally be enclosed by "{" and "}" for better readability.

<token-value> `<token-value>` is interpreted as regular expression as described in `re`.

Tokens are simply macros where `{<token-name>}` is replaced by `<token-value>` such that quantifications of tokens hold:

```
foo1 = bar
foo2 = [bar]
foo3 = a
foo4 = \n
foo5 = [ bar ]*

<some-rule-1> := here\x20is\x20{foo1}*
<some-rule-2> := here\x20is\x20{foo2}*
<some-rule-3> := here\x20is\x20{foo3}*
<some-rule-4> := here\x20is\x20{foo4}*
<some-rule-5> := here\x20is\x20{:foo1:}*
<some-rule-6> := here\x20is\x20[{:foo1:}{:foo4:]]*
<some-rule-7> := here\x20is\x20{foo5}
```

Token replacement creates following rules from this, before really parsing them:

```
<some-rule-1> := here\x20is\x20(?:bar)*
<some-rule-2> := here\x20is\x20[bar]*
<some-rule-3> := here\x20is\x20a*
<some-rule-4> := here\x20is\x20\n*
<some-rule-5> := here\x20is\x20bar*
<some-rule-6> := here\x20is\x20[bar\n]*
<some-rule-7> := here\x20is\x20(?:(?:bar)*)
```

You see that tokens are used in a way that the quantification after the token always quantifies the entire token not like in `<some-rule-5>` where simply the value of the token was substituted.

So you can also let your token be exanded with `{:<token-name>:}` syntax, which is simply expanding the value of tokens without taking care of grouping for clean quantifications. This expansions are intended to be used e.g. as character-classes (this is also the reason for the choice of syntax), as seen in `<some-rule-6>`, but maybe there are other use cases.

In `<some-rule-7>` there is used `{foo5}` token. Where you see a special notation of:

```
foo5 = [ bar ]*
```

In tokens a "[" sorrounded by whitespace is replaced by "(?:" and a "]" surrounded by whitespace or followed by a quantifier like "?", "*", "+" or "{a,b}" is replaced by ")" and the optional quantifier. This is for convenience and better readability of the token rule. Do not confuse with:

```
foo6 = [bar]*
```

Because:

```
{foo5} ~~ barbar
```

```
{foo6} ~~ brarab

{foo5} !~ brarab
```

Rules

Formally rules are defined as this:

```
rule-def      :- <rule-name> <operator> <alternatives>
alternatives :- <sequence> [ {or} <sequence> ]
sequence      :- [ <non-terminal> | <terminal> ]
non-terminal := [ <capturing> | <non-capturing>
                | <sub-rule ] <quantification>?

terminal      := <string> | <regex>

quantification :=  "?" | "*" | "+" | "{" \d* "," \d* "}"

operator := <token-op> | <backtracking-op> | <non-backtracking-op>
          | <backtracking-sigspace-op> | <non-backtracking-sigspace-op>



token-op                     := "="
backtracking-op              := ":" <parse-action> "="
backtracking-sigspace-op     := ":" <parse-action> "-"
non-backtracking-op          := <parse-action> "="
non-backtracking-sigspace-op := <parse-action> "-"

parse-action                 := ":" | [^=]+
```

A production rule has the form:

```
:sigspace {ws}
after-ws          = (?<=\s)
before-ws         = (?=\s)
or                = {after-ws} \| {before-ws}

<production-rule> ::- <non-terminal> <rule-op> <alternatives>

<alternatives>    ::- <sequence> [ {or} <sequence> ]*

<sequence>        ::= [ <non-terminal> | <terminal> ]+

<terminal>        ::= <string> | <regex>

<non-terminal>    ::= [ <capturing> | <non-capturing> | <sub-rule>
                      ] <quantification>?

<quantification>  ::= "?" | "*" | "+" | "{" \d* "," \d* "}"
```

`<non-terminal>` May be enclosed by “<”, “>” for beeing closer to BNF or better readability, but this is not neccesserily needed. So:

```
<foo> ::= "bar"
```

is equivalent to:

```
foo   ::= "bar"
```

<rule-op> This is a tricky thing. Usually you will use ":=". But you can use any `<parse-action>=` for it. See also parse-actions.

There are more flavors of the *<rule-op>*, for specifying significant space and backtracking on failure:

rule-op	description
=	Specify a token, which can be used later as macro.
:=	Normal rule.
::=	Backtracking rule.
:foo=	Backtracking rule calling "foo" method from ParseActions
foo=	Normal rule calling "foo" method from ParseActions
:-	Normal rule using significant whitespace
::-	Backtracking rule using significant whitespace.

In short:

- a rule with a `<rule-op>` with a preceding ":", does backtracking on failure.
- a rule with a `<rule-op>` using a "-" instead of "=" has significant whitespace

<string> May be a double-quoted or a single-quoted string. Like:

```
"foo" "foo\n" "foo\"" 'foo"bar"' 'bar\''
```

This is a terminal in terms of grammars.

<regex> Anything, which is not anything else listed here is interpreted as regular expression like defined in `re`.

<non-capturing> From syntactical point of view it is a "`<.capturing>`" rule. So the same like a capturing rule, except you have a "." right behind the opening "<".

No-capturing rules pass their captured children to the parental rule, which combines the children of all non-capturing childrens to its own list of children.

Examples:

```
<.simple-rule> <.rule-with-arg:foo> <.ext-rule{ here is more }>
```

<capturing> Capturing rule has three syntactical flavours:

```
<ws>        ::= \s+

<simple>    ::= "<" <non-terminal-name> ">"

<with-arg>  ::= "<" <non-terminal-name> ":" <arg>  ">"

<with-args> ::= "<" <non-terminal-name> "{" [ <.ws> <extarg> ]* <.ws> "}>"

<arg>       ::= (?:\\\\|\\>|[^>])*

<extarg>    ::= (?!\}>\s) (?!\}>$) [^\s]+
```

Where *name* is the name of another non-terminal. The two extended versions of rule-calls are for invoking custom rules, which do more than simply parsing sequencenses or alternatives.

Please note for `<with-args>` rules:

Backtracking

Explain here how backtracking works

Significant Whitespace

Explain here how significant whitespace works.

Test Assertions

Assert, that your rules match

If you want to assert, that a rule matches a certain string you can add an assertion:

```
<my-rule> ~~ "foo"
```

Assert, that your rules do not match

If you want to assert that a rule does not match some string you can add an assertion:

```
<my-rule> !~ "foo"
```

Assert, that your rules produce some expected syntax tree

If you want to assert that a rule produces some syntax tree you can add an assertion:

```
<my-rule> ~~ "foo" -> my-rule("foo")
```

Token and exact match

Difference between *token match* and *exact match* is, that in *token matches* whitespace is ignored and only non-whitespace tokens are compared. In *exact match* there is compared complete string:

```
<my-rule> ~~ "something" ->
    In token
        match only
         non-whitespace          tokens
     are considered for
               comparison.

<my-rule> =~ "something" --> "Must output exact this string"
```

Multiline input

You can specify multiline input (or expected output) by lines preceded by ““| “”:

```
<my-rule> ~~
    | first line
    | second line
    |
    | And a line after an
    | empty line
    ->
    | Same for
    | expected output.
```

For testing your grammar you can setup test assertions for your rules:

```
<my-rule> ~~ "foo"
<my-rule> !~ "foo"
<my-rule> ~~ "foo" -> my-rule("foo")
<my-rule> =~ "foo" -> "foo"
<my-rule> =~
    | a really
    | long, long
    | text.
    |
    | with another paragraph
    -> here
       is what
       I expect
       to be the ast's output.

<my-rule> =~ "foo" -MyParseActions-> [ 'f', 'o', 'o' ]
```

Formally test assertions are created with following syntax:

```
<test-assertion> ::= <test-rule> <test-op> <string-to-match> [ <ast-op> <expected-
↪output> ]?

<test-rule> ::= "<" <non-terminal-name> ">"

<test-op>   ::= (?P<token-match>~~)|(?P<not-match>!~)|(?P<equal-match>=~)

<string-to-match>   ::= <quoted-string> | <multi-line-string>
<multi-line-string> ::= [ \s* [ \|(?P<line>\n) | \|\s <line> ] ]+

<ast-op>    ::= -> | -(?P<parse-actions-name>\w+)->

<expected-output> ::= <quoted-string> | <multi-line-string> | <tokens>
```

CHAPTER 3

Testing of aPTK Grammars

Another feature of aPTK is, that you can define your grammar-rule testcases right in your grammar:

```
:grammar AddGrammar1

<addition> @- <term> "+" <term>
<term>     #= \d+
<ws>       := \s*
```

So far our grammar, now here follow the tests:

1. Test, if addition matches some term:

   ```
   <addition> ~~ "5 + 4"
   ```

2. Test, if addition matches some term and produces some special syntax-tree:

   ```
   <addition> ~~ "5 + 4" -> addition( term( '5' ), term( '4' ) )
   ```

3. Test, if addition produces right AST:

   ```
   <addition> ~~ "5 + 5" --> [5, 5]
   ```

   In this case default `ParseActions` have been used. To use a different parse-action class you can specify it between the "-" and "->", for the above you could also write explicitely:

   ```
   :parse-actions ParseActions aptk.actions.ParseActions

   <addition> =~ "5 + 5" -ParseActions-> [5, 5]
   ```

4. Assert that addition does not match something:

   ```
   <addition> !~ "5- 4"
   ```

CHAPTER 4

aptk - API

This is the major interface for the user. Usually you will only:

```
from aptk import *
```

And then define your grammar, maybe parse-actions. This could for example look like this:

```
class AdditionGrammar(Grammar):
    r'''Parses addition-expressions.

    .. highlight aptk

    sum :- <number> "+" <sum> | <number>

    :parse-actions aptk.Sum

    <sum> ~~ 5 + 3 -Sum-> 8
    '''

class Sum(ParseActions):
    def sum(self, P, lex):
        return sum([ x.ast for x in lex ])
```

For parsing a string, you can use `parse()`:

```
parse_tree = parse("4 + 2", AdditionGrammar, Sum)
```

For convenience there is also a function `ast()`, which returns abstract syntax-tree of a node:

```
result = ast(parse_tree)
```

For convienece you can shortcut this with:

```
result = ast("4 + 2", AdditionGrammar, Sum)
```

class `aptk.Grammar`(*s=None*, ***kargs*)

Default grammar with basic tokens and rules.

This is the grammar, you will usually derive your grammars from.

It provides most common tokens:

```
SP   = \x20
NL   = \r?\n
LF   = \n
CR   = \r
CRLF = \r\n
ws   = \s+
ws?  = \s*
N    = [^\n]
HWS  = [\x20\t\v]
LINE = [^\n]*\n
```

And a general ActionMap, which lets you connect your grammar to basic `ParseActions`:

```
:parse-action-map
    "$" make_string
    "@" make_list
    "%" make_dict
    "#" make_number
    "<" make_inherit
    ">" make_name
    "~" make_quoted
```

And most common rules:

```
ident     $= [A-Za-z_\-][\w\-]*
number    #= [+-]?\d+(?:\.\d+)?
integer   #= \d+
dq-string ~= "(?:\\\\|\\[^\\]|[^"\\])*"
sq-string ~= '(?:\\\\|\\[^\\]|[^'\\])*'
ws        $= \b{ws}\b|{ws?}
line      $= [^\n]*\n
```

Making explicit the whitespace rule default from BaseGrammar:

```
:sigspace <.ws>
```

Define how args of BRANCH are parsed:

```
:args-of BRANCH string capturing non-capturing regex
```

Define operation precedence parser:

```
:args-of  EXPR string capturing non-capturing raw
            => aptk.oprec.OperatorPrecedenceParser
```

BRANCH(*P*, *s=None*, *start=None*, *end=None*, *args=None*)

lookahead and branch into some rule.

Example:

```
branched := <BRANCH{
             "a"    <a-rule>
```

```
             [bcd]   <bcd-rule>
             a|b     <a-or-b-rule>
             <default-rule>
            }>
```

If string to be matched startswith

ERROR (*P*, *s=None*, *start=None*, *end=None*, *args=None*)
raise a syntax error.

Example:

```
foo := <x> | <ERROR{Expected "x"}>
```

Please note that whitespace will be collapsed to single space.

aptk.**parse** (*s*, *grammar*, *actions=None*, *rule=None*)
parse *s* with given grammar and apply actions to produced lexems.

aptk.**ast** (*s*, *grammar=None*, *actions=None*, *rule=None*)
return ast of s if has one, else, parse s using grammar and actions and return it then

CHAPTER 5

aptk - module reference

class `aptk.parser.`**`Parser`**(*grammar*, *actions=None*)

Parser combines grabbar and parse-actions to parser.

An object of this class combines an abstract grammar and parse-actions to a parser, which produces an abstract syntax tree.

If no actions given, defaults to `ParseActions` object.

aptk.actions - Parse Actions

Parse Actions are used to create an abstract syntax tree from your parse tree.

Parse Actions are expected to be attributes of the parse-actions object passed to `Parser`. This can be an object of a class derived from `ParseActions`, but can be also a module with a collection of functions.

Parse-Action Callables

A parse-action is called from parser with two parameters:

- *parser* - current `Parser` object
- *lex* - current `Lexem` object

Whatever the parse-action returns will be then written into the `ast` attribute of the `Lexem` object.

Connecting Parse-Actions to Rules

The parser calls a parse-action for each captured match object, which is represented by a `Lexem` object:

- If there is defined a parse-action in the matching rule, it is called. In following rule there would be called parse-action "some_action", if you captured something using `<some-rule>`:

```
some-rule  some_action=  "some text"
```

You can map shortcuts to actions:

```
:parse-action-map
    "$" => other_action

other-rule $= "other text"
```

In this case there would be called parse-action "other_action", if you captured "other text" with `<other-rule>`.

- If there is not defined a parse-action in matching rule, it is tried to find following parse-actions if `<my_rule>` was matched:
  - `my_rule`
  - `make_my_rule`
  - `got_my_rule`
- If no parse-action found, there is nothing done

Pairs

Setting an ast to a pair *(name, result)*, where *name* is the rule's name and *result* is result from parse-action, can be achieved with following syntax:

```
paired  action=>  <some> <rule>
```

If you append a ">" to your operator and you define an action for your rule the ast of the capture of `<paired>` will be the pair *(paired, «result of action()»)*.

Example

```
>>> from aptk import *
>>>
>>> class DashArithmeticGrammar(Grammar):
...     r"""Simple grammar for addition and substraction.
...
...     dash_op    <= <sum> | <difference> | <number>
...     sum        := <number> "+" <dash_op>
...     difference := <number> "-" <dash_op>
...     """
>>>
>>> class CalculatorActions(ParseActions):
...     r"""inherit number from ParseActions"""
...     def sum(self, p, lex):
...         return lex[0].ast + lex[1].ast
...     def difference(self, p, lex):
...         return lex[0].ast - lex[1].ast
>>>
>>> ast("1 + 3 - 2",
...      grammar = DashArithmeticGrammar,
...      actions = CalculatorActions())
2
```

class `aptk.grammar.BaseGrammar`(*s=None*, ***kargs*)
Most basic grammar class.

Usually you will rather use *Grammar* instead of this for deriving you classes from. If you really need a blank grammar, you can derive your grammar from this class.

A Grammar class has following attributes:

__metaclass__ `GrammarType` - the type of a grammar class

TOKENS A dictionary of token-parsing regexes, which can be used with `{name}` for the smart value and `{:name:}` for the unchanged value.

Smart value means that if you specify a token like:

```
token = abcd
```

You still can quantify the token without having strange effects:

a-rule := foo{token}+

Will be translated to:

```
a-rule := foo(?:abcd)+
```

The other way of access:

```
b-rule := foo{:token:}+
```

Will be translated to:

```
b-rule := fooabcd+
```

You can use the second form for example for defining character classes:

```
word-chars = A-Za-z0-9_
dash       = \-
ident      = [{:word-chars:}{:dash:}]+
```

The tokens are evaluated directly after a rule-part is read.

ACTIONS This dictionary maps rule-names to action-names, which are methods in either ParseAction object passed to parser or in Grammar. This map is created from implicit parse-action directives. Parse-actions are run on lexing a MatchObject and fill the *ast*-attribute of Lexem with life.

Implicit parse-actions are specified by *_PARSE_ACTION_MAP_*.

_START_RULE_ Name of start-rule if no other given.

class `aptk.grammar.Grammar`(*s=None*, ***kargs*)
Default grammar with basic tokens and rules.

This is the grammar, you will usually derive your grammars from.

It provides most common tokens:

```
SP   = \x20
NL   = \r?\n
LF   = \n
CR   = \r
CRLF = \r\n
ws   = \s+
ws?  = \s*
```

```
N    = [^\n]
HWS  = [\x20\t\v]
LINE = [^\n]*\n
```

And a general ActionMap, which lets you connect your grammar to basic `ParseActions`:

```
:parse-action-map
    "$" make_string
    "@" make_list
    "%" make_dict
    "#" make_number
    "<" make_inherit
    ">" make_name
    "~" make_quoted
```

And most common rules:

```
ident     $= [A-Za-z_\-][\w\-]*
number    #= [+-]?\d+(?:\.\d+)?
integer   #= \d+
dq-string ~= "(?:\\\\|\\[^\\]|[^"\\])*"
sq-string ~= '(?:\\\\|\\[^\\]|[^'\\])*'
ws        $= \b{ws}\b|{ws?}
line      $= [^\n]*\n
```

Making explicit the whitespace rule default from BaseGrammar:

```
:sigspace <.ws>
```

Define how args of BRANCH are parsed:

```
:args-of BRANCH string capturing non-capturing regex
```

Define operation precedence parser:

```
:args-of  EXPR string capturing non-capturing raw
            => aptk.oprec.OperatorPrecedenceParser
```

BRANCH (*P*, *s=None*, *start=None*, *end=None*, *args=None*)

lookahead and branch into some rule.

Example:

```
branched := <BRANCH{
             "a"    <a-rule>
             [bcd]  <bcd-rule>
             a|b    <a-or-b-rule>
             <default-rule>
            }>
```

If string to be matched startswith

ERROR (*P*, *s=None*, *start=None*, *end=None*, *args=None*)

raise a syntax error.

Example:

```
foo := <x> | <ERROR{Expected "x"}>
```

Please note that whitespace will be collapsed to single space.

`aptk.grammar.`**`compile`**`(`*input, type=None, name=None, extends=None, grammar=None, filename=None*`)`

compile a grammar

You can pass different inputs to this class, which has influence on return value.

 # *input* is grammar class:

```
class MyGrammar(Grammar):
    r"""This is my grammar class

    .. highlight:: aptk

    My grammar has following rule::

        <foo> = "bar"
    """
```

This is the way you usually invoke *`compile()`* with a grammar class, because *`compile()`* is invoked by `GrammarType`.

 # Append whatever is defined in *input* to *grammar*:

```
class MyGrammar(Grammar):
    r"""Here are rules defined"""

...

compile("here are more rules", grammar=MyGrammar)
```

input may be either a file object (something having a read() method) or a string.

 # Create a new grammar named *name*, which extends grammars passed in iteratable *extends*. If you do not pass *extends*, then your grammar will extends *`Grammar`*, extracting the rules from *input*.

Simply compile *input* to a list of grammars.

list_of_grammars = compile(""" :grammar first some := <rule>

:grammar second another := <rule>

""")

input may be either a file object (something having a *read()* method) or a string.

Parameters

input Pass a grammar class, a string or whatever, which has a *read()* method, e.g. a file object.

type Type of input, "sphinx" or "native".

name Name of grammar, which shall be created and keep the rules given in *input*.

extends If you pass a *name* you may pass *extends* as a list of names of grammars.

grammar If you pass a grammar class, the input is added to this grammar class.

filename for informative purpose

Returns A GrammarClass or (if no specific grammar given in some way) a list of grammar classes.

aptk.oprec - Operation Precedence Parser

Operation precedence parsers are intended to parse expressions, where never is a sequence of non-terminals. Usually you will use it to parse (mathematical) expressions.

You can invoke OperationPrecedenceParser into your grammar by using:

```
:args-of OPTABLE string capturing non-capturing raw
         => aptk.oprec.OperatorPrecedenceParser
```

Then you can create rules like this:

```
my_rule_name1 := <OPTABLE{
                    :rule T <.term>
                    ...
                    }>

my_rule_name2 := <OPTABLE{
                    :rule T <.term2>
                    :rule W ""
                    :rule E
                    ...
                    }>
```

Every `OPTABLE` invokation creates a new rule.

In any `Grammar`-descending grammar this is already done for you and operation precedence is accessible via rule `EXPR`:

```
:grammar operation-precedence-parser-tests

expr  := <EXPR{
           :flags with-ops

           :op L E+E
       }>
```

You have to define a `<term>`, such that a term, which is the only non-terminal-rule in expressions, can be parsed:

```
term        := <number> | <ident>
```

Expression above parses for example following expressions:

```
<expr> ~~ 5 + 5
       -> expr( E+E( number( '5' ), op( '+' ), number( '5' ) ) )

<expr> ~~ 1 + 2 + 3
       -> expr( E+E(
            E+E(
              number( '1' ),
              op( '+' ),
              number( '2' )
            ),
            op( '+' ),
            number( '3' )
          ) )
```

You see in parse trees of expressions above, that the operator is also lexed (as "op"). This is triggered by flag `with-ops`. If you leave out this flag, operators are not lexed, as you see in further examples:

```
expr2 :- <EXPR{
           :op L E+E
           :op L E-E  = E+E
           :op L E*E  > E+E
           :op L E/E  = E*E
           :op L E**E > E*E
           :op L E++  > E**E
           :op R ++E  = E++
           :op R (E)  > E++
       }>
```

First example where operator precedence table is used:

```
<expr2> ~~ 5 + 5 * 4
        -> expr2( E+E(
              number( '5' ),
              E*E( number( '5' ), number( '4' ) )
           ) )
```

A more complex example:

```
<expr2> ~~ 5**2 + 4**2/3**1 * 2 + 1
        ->  expr2( E+E(
               E+E(
                 E**E( number( '5' ), number( '2' ) ),
                 E*E(
                   E/E(
                     E**E( number( '4' ), number( '2' ) ),
                     E**E( number( '3' ), number( '1' ) )
                   ),
                   number( '2' )
                 )
               ),
               number( '1' )
            ) )
```

Here you see how whitespace has influence on tokenizer:

```
<expr2> ~~ 1*3+++++1
        -> expr2( E+E(
             E*E( number( '1' ), E++( E++( number( '3' ) ) ) ),
             number( '1' )
           ) )

<expr2> ~~ 1*3++ + ++1
        -> expr2( E+E(
            E*E( number( '1' ), E++( number( '3' ) ) ),
            ++E( number( '1' ) )
        ) )

<expr2> ~~ 1*3+++(++1)
        -> expr2( E+E(
             E*E( number( '1' ), E++( number( '3' ) ) ),
             (E)( ++E( number( '1' ) ) )
           ) )
```

```
<expr2> ~~ (1*3)++
        -> expr2( E++(
             (E)(
               E*E(
                 number( '1' ),
                 number( '3' )
               )
             )
           ) )
```

Here you see how operator precedence has influence on interpretation of a term `++1--`:

```
prepostest1 := <EXPR{
               :op L ++E
               :op L E-- > ++E
              }>

<prepostest1> ~~ ++1-- -> prepostest1( ++E( E--( number( '1' ) ) ) )

prepostest2 := <EXPR{
               :op L ++E
               :op L E-- < ++E
              }>

<prepostest2> ~~ ++1-- -> prepostest2( E--( ++E( number( '1' ) ) ) )

postcirc1   :- <EXPR{
                  :op R E(E)
                  :op R E,E < E(E)
               }>

<postcirc1> ~~  sum(1, 2)
            -> postcirc1( E(E)(
                 E,E(
                   number( '1' ),
                   number( '2' )
                 )
               ) )

<postcirc1> ~~  sum(1, 2, 3, 4)
            -> postcirc1( E(E)(
                 E,E(
                   number( '1' ),
                   E,E(
                     number( '2' ),
                     E,E(
                       number( '3' ),
                       number( '4' )
                     )
                   )
                 )
               ) )
```

Typical operator association you find here:

- http://msdn.microsoft.com/en-us/library/126fe14k.aspx
- http://en.cppreference.com/w/cpp/language/operator_precedence

class `aptk.grammar_tester.`**`GrammarTest`**(*name*, *op*, *pos*, *input*, *actions*, *expected*, *skip=None*, *debug=False*)
simple class to save testdata

class `aptk.grammar_tester.`**`GrammarTestCase`**(*name*, *grammar_test*, *grammar*)
A TestCase for Grammar

class `aptk.grammar_tester.`**`RuleTest`**(*name*, *op*, *pos*, *input*, *actions*, *expected*, *skip=None*, *debug=False*)
name specifies a rule

class `aptk.grammar_tester.`**`TokenTest`**(*name*, *op*, *pos*, *input*, *actions*, *expected*, *skip=None*, *debug=False*)
name specifies a token

`aptk.grammar_tester.`**`generate_testsuite`**(*grammar*, *suite=None*, *patterns=None*)
gets a grammar class and maybe a suite

exception `aptk.grammar_compiler.`**`GrammarError`**(*grammar_compiler*, *msg*, ***kargs*)
exception in grammar compilation.

This exception is raised, if there is an error in grammar compilation.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index