
DHParser Documentation

Release 0.8

Eckhart Arnold

Apr 02, 2018

CONTENTS:

1	DHParser's Step by Step Guide	1
1.1	Setting up a new DHParser project	1
1.1.1	Installing DHParser from the git repository	1
2	DHParser User's Guide	3
2.1	Introduction	3
3	DHParser Reference Manual	5
3.1	Fundamentals	5
3.2	Compiler Creation Workflow	6
3.3	Component Guide	6
3.3.1	Parser	6
3.3.2	AST-Transformation	6
3.3.3	Compiler	6
3.4	Module Structure of DHParser	6
3.5	Class Hierarchy of DHParser	6
4	Module Reference	7
4.1	Main Modules Reference	8
4.1.1	Module <code>preprocess</code>	8
4.1.2	Module <code>syntaxtree</code>	9
4.1.3	Module <code>parse</code>	12
4.1.4	Module <code>transform</code>	21
4.1.5	Module <code>compile</code>	25
4.1.6	Module <code>error</code>	26
4.2	Domain Specific Language Modules Reference	27
4.2.1	Module <code>ebnf</code>	28
4.2.2	Module <code>dsl</code>	30
4.2.3	Module <code>testing</code>	32
4.3	Supporting Modules Reference	33
4.3.1	Module <code>toolkit</code>	33
4.3.2	Module <code>log</code>	34
4.3.3	Module <code>stringview</code>	36
4.3.4	Module <code>versionnumber</code>	37
5	Indices and tables	39
	Python Module Index	41
	Index	43

DHPARSER'S STEP BY STEP GUIDE

This step by step guide goes through the whole process of desining and testing a domain specific notation from the very start. (The terms “domain specific noation” and “domain specific language” are used interchangeably in the following. Both will abbreviated by “DSL”, however.) We will design a simple domain specific notation for poems as a teaching example. On the way we will learn:

1. how to setup a new DHParser project
2. how to use the test driven development approach to designing a DSL
3. how to describe the syntax of a DSL with the EBNF-notation
4. how to specify the transformations for converting the concrete syntax tree that results from parsing a text denoted in our DLS into an abstract syntax tree that represents or comes close to representing out data model.
5. how to write a compiler that transforms the abstract syntax tree into a target representation which might be a html page, epub or printable pdf in the case of typical Digital-Humanities-projects.

1.1 Setting up a new DHParser project

Since DHParser, while quite mature in terms of implemented features, is still in a pre-first-release state, it is for the time being more recommendable to clone the most current version of DHParser from the git-repository rather than installing the packages from the Python Package Index (PyPI).

This section takes you from cloning the DHParser git repository to setting up a new DHParser-project in the `experimental`-subdirectory and testing whether the setup works. Similarly to current web development practices, most of the work with DHParser is done from the shell. In the following, we assume a Unix (Linux) environment. The same can most likely be done on other operating systems in a very similar way, but there might be subtle differences.

1.1.1 Installing DHParser from the git repository

In order to install DHParser from the git repository, open up a shell window and type:

```
$ git clone git@gitlab.lrz.de:badw-it/DHParser.git
$ cd DHParser
```

The second command changes to the DHParser directory. Within this directory you should recognise the following subdirectories and files. There are more files and directories for sure, but those will not concern us for now:

DHParser/	- the DHParser python packages
documentation/	- DHParser's documentation in html-form
documentation_source	- DHParser's documentation in reStructuredText-Format
examples/	- some exmamples for DHParser (mostly incomplete)

(continues on next page)

(continued from previous page)

experimental/	- an empty directory for experimenting
test/	- DHParser's unit-tests
dhparser.py	- DHParser's command line tool for setting up projects
README.md	- General information about DHParser
LICENSE.txt	- DHParser's license. It's open source (hooray!)
Introduction.md	- An introduction and appetizer for DHParser

In order to setup a new DHParser project, you run the `dhparser.py`-script with the name of the new project. For the sake of the example, let's type:

```
$ python dhparser.py experimental/poetry
```

```
Creating new DHParser-project "poetry". Creating file "grammar_tests/01_test_word.ini". Creating file  
"grammar_tests/02_test_document.ini". Creating file "poetry.ebnf". Creating file "README.md". Cre-  
ating file "tst_poetry_grammar.py". Creating file "example.dsl". ready.
```

```
$ cd experimental/poetry
```

This creates a new DHParser-project with the name "poetry" in directory with the same name within the subdirectory "experimental".

DHPARSER USER'S GUIDE

This user's guide explains how to use create, test and employ a domain specific language with DHParser for encoding text or data in a Digital Humanities Project.

2.1 Introduction

Most Digital Humanities projects or least most text-centered DH projects involve in some way or other the entering and encoding of annotated text or data into a computer. And the systems that scientists use for that purpose consist of an input surface (or “redaction system”) for entering the data, a storage system to keep the data and a presentation system for providing the data and possibly also functionality for working with the data to human or machine recipients. A typical example of this type of system is Berlin' Ediarum-System, which consists of an XML-Editor for entering data, an XML-Database for storing the data and a web application for providing the data to human readers or other web services via an application programming interface (API). Ediarum is also typical, because like many DH-projects it assumes an XML-based workflow.

DHPARSER REFERENCE MANUAL

This reference manual explains the technology used by DHParser. It is intended for people who would like to extend or contribute to DHParser. The reference manual does not explain how a Domain Specific Language (DSL) is developed (see the User's Manual for that). It explains the technical approach that DHParser employs for parsing, abstract syntax tree transformation and compilation of a given DSL. And it describes the module and class structure of the DHParser Software. The programming guide requires a working knowledge of Python programming and a basic understanding or common parser technology from the reader. Also, it is recommended to read the introduction and the user's guide first.

3.1 Fundamentals

DHParser is a parser generator aimed at but not restricted to the creation of domain specific languages in the Digital Humanities (DH), hence the name "DHParser". In the Digital Humanities, DSLs allow to enter annotated texts or data in a human friendly and readable form with a Text-Editor. In contrast to the prevailing XML-approach, the DSL-approach distinguishes between a human-friendly *editing data format* and a machine friendly *working data format* which can be XML but does not need to be. Therefore, the DSL-approach requires an additional step to reach the *working data format*, that is, the compilation of the annotated text or data written in the DSL (editing data format) to the working data format. In the following a text or data file written in a DSL will simply be called *document*. The editing data format will also be called *source format* and the working data format be denoted as *target format*.

Compiling a document specified in a domain specific language involves the following steps:

1. **Parsing** the document which results in a representation of the document as a concrete syntax tree.
2. **Transforming** the concrete syntax tree (CST) into an abstract syntax tree (AST), i.e. a streamlined and simplified syntax tree ready for compilation.
3. **Compiling** the abstract syntax tree into the working data format.

All of these steps are carried out by the computer without any user intervention, i.e. without the need of humans to rewrite or enrich the data during any of these steps. A DSL-compiler therefore consists of three components which are applied in sequence, a *parser*, a *transformer* and a *compiler*. Creating, i.e. programming these components is the task of compiler construction. The creation of all of these components is supported by DHParser, albeit to a different degree:

1. *Creating a parser*: DHParser fully automizes parser generation. Once the syntax of the DSL is formally specified, it can be compiled into a python class that is able to parse any document written in the DSL. DHParser uses Parsing-Expression-Grammars in a variant of the Extended-Backus-Naur-Form (EBNF) for the specification of the syntax. (See *examples/EBNF/EBNF.ebnf* for an example.)
2. *Specifying the AST-transformations*: DHParser supports the AST-transformation with a depth-first tree traversal algorithm (see *DHParser.transform.traverse*) and a number of stock transformation functions which can also be

combined. Most of the AST-transformation is specified in a declarative manner by filling in a transformation-dictionary which associates the node-types of the concrete syntax tree with such combinations of transformations. See *DHParseR.ebnf.EBNF_AST_transformation_table* as an example.

3. *Filling in the compiler class skeleton:* Compiler generation cannot be automated like parser generation. It is supported by DHParseR merely by generating a skeleton of a compiler class with a method-stub for each definition (or “production” as the definition are sometimes also called) of the EBNF-specification. (See *examples/EBNF/EBNFCompiler.py*) If the target format is XML, there is a chance that the XML can simply be generated by serializing the abstract syntax tree as XML without the need of a dedicated compilation step.

3.2 Compiler Creation Workflow

TODO: Describe: - setting up a new projekt - invoking the DSL Compiler - conventions and data types - the flat namespace of DH Parser

3.3 Component Guide

3.3.1 Parser

Parser-creation is supported by DHParseR by an EBNF to Python compiler which yields a working python class that parses any document the EBNF-specified DSL to a tree of Node-objects, which are instances of the *class Node* defined in *DHParseR/syntaxtree.py*

The EBNF to Python compiler is actually a DSL-compiler that has been crafted with DHParseR itself. It is located in *DHParseR/enbf.py*. The formal specification of the EBNF variant used by DHParseR can be found in *examples/EBNF/EBNF.ebnf*. Comparing the automatically generated *examples/EBNF/EBNFCompiler.py* with *DHParseR/enbf.py* can give you an idea what additional work is needed to create a DSL-compiler from an autogenerated DSL-parser. In most DH-projects this task will be less complex, however, as the target format is XML which usually can be derived from the abstract syntax tree with fewer steps than the Python code in the case of DHParseR’s EBNF to Python compiler.

3.3.2 AST-Transformation

Other than for the compiler generation (see the next point below), a functional rather than object-oriented approach has been employed, because it allows for a more concise specification of the AST-transformation since typically the same combination of transformations can be used for several node types of the AST. It would therefore be tedious to fill in a method for each of these. In a sense, the specification of AST-transformation constitutes an “internal DSL” realized with the means of the Python language itself.

3.3.3 Compiler

3.4 Module Structure of DHParseR

3.5 Class Hierarchy of DHParseR

MODULE REFERENCE

DHParser is split into a number of modules plus one command line utility (`dhparser.py`, which will not be described here.)

Usually, the user or “importer” of DHParser does not need to worry about its internal module structure, because DHParser provides a flat namespace form which all of its symbols can be imported, e.g.:

```
from DHParser import *
```

or:

```
from DHParser import recompile_grammar, grammar_suite, compile_source
```

However, in order to add or change the source code of DHParser, its module structure must be understood. DHParser’s modules can roughly be sorted into three different categories:

1. Modules that contain the basic functionality for packrat-parsing, AST-transformation and the skeleton for a DSL-compilers.
2. Modules for EBNF-Grammars and DSL compilation.
3. Support or “toolkit”-modules that contain different helpful functions

The import-order of DHParser’s modules runs accross these categories. In the following list the modules further below in the list may import one or more of the modules further above in the list, but not the other way round:

- `versionnumber.py` – contains the verison number of DHParser
- `toolkit.py` – utility functions for DHParser
- `stringview.py` – a string class where slices are views not copies as with the standard Python strings.
- `preprocess.py` – preprocessing of source files for DHParser
- `error.py` – error handling for DHParser
- `syntaxtree.py` – syntax tree classes for DHParser
- **`transform.py` – transformation functions for converting the concrete** into the abstract syntax tree
- `logging.py` – logging and debugging for DHParser
- `parse.py` – parser combinators for for DHParser
- **`compile.py` – abstract base class for compilers that transform an AST** into something useful
- `ebnf.py` – EBNF -> Python-Parser compilation for DHParser
- `dsl.py` – Support for domain specific notations for DHParser
- `testing.py` – test support for DHParser based grammars and compilers

4.1 Main Modules Reference

The core of DHParse are the modules containing the functionality for the parsing and compiling process. The modules `preprocess`, `parse`, `transform` and `compile` represent particular stages of the parsing/compiling process, while `syntaxtree` and `error` define classes for syntax trees and parser/compiler errors, respectively.

4.1.1 Module `preprocess`

Module `preprocess` contains functions for preprocessing source code before the parsing stage as well as source mapping facilities to map the locations of parser and compiler errors to the non-preprocessed source text.

Preprocessing (and source mapping of errors) will only be needed for some domain specific languages, most notable those that cannot completely be described with context-free grammars.

`make_token` (*token: str, argument: str = ""*) → str

Turns the `token` and `argument` into a special token that will be caught by the `PreprocessorTokenParser`.

This function is a support function that should be used by preprocessors to inject preprocessor tokens into the source text.

`strip_tokens` (*tokenized: str*) → str

Replaces all tokens with the token's arguments.

`nil_preprocessor` (*text: str*) → Tuple[str, Union[typing.Callable[[int], int], funtools.partial]]

A preprocessor that does nothing, i.e. just returns the input.

`chain_preprocessors` (**preprocessors*) → Union[typing.Callable[[str], typing.Union[str, typing.Tuple[str, typing.Union[typing.Callable[[int], int], funtools.partial]]], funtools.partial]

Merges a sequence of preprocessor functions in to a single function.

`prettyprint_tokenized` (*tokenized: str*) → str

Returns a pretty-printable version of a document that contains tokens.

`class SourceMap` (*positions, offsets*)

`offsets`

Alias for field number 1

`positions`

Alias for field number 0

`tokenized_to_original_mapping` (*tokenized_source: str*) → `preprocess.SourceMap`

Generates a source map for mapping positions in a text that has been enriched with token markers to their original positions.

Parameters `tokenized_source` – the source text enriched with token markers

Returns a source map, i.e. a list of positions and a list of corresponding offsets. The list of positions is ordered from smallest to highest. An offset is valid for its associated position and all following positions until (and excluding) the next position in the list of positions.

`source_map` (*position: int, srcmap: preprocess.SourceMap*) → int

Maps a position in a (pre-)processed text to its corresponding position in the original document according to the given source map.

Parameters

- **`position`** – the position in the processed text

- **srcmap** – the source map, i.e. a mapping of locations to offset values

Returns the mapped position

with_source_mapping (*result: Union[str, typing.Tuple[str, typing.Union[typing.Callable[[int], int], functools.partial]]])* → *Tuple[str, Union[typing.Callable[[int], int], functools.partial]]*

Normalizes preprocessors results, by adding a mapping if a preprocessor only returns the transformed source code and no mapping by itself. It is assumed that in this case the preprocessor has just enriched the source code with tokens, so that a source mapping can be derived automatically with *tokenized_to_original_mapping()* (see above).

4.1.2 Module `syntaxtree`

Module `syntaxtree` defines the `Node`-class for syntax trees as well as an abstract base class for parser-objects. The latter is defined here, because node-objects refer to parser-objects. All concrete parser classes are defined in the `parse` module.

class ParserBase (*name=""*)

`ParserBase` is the base class for all real and mock parser classes. It is defined here, because `Node` objects require a parser object for instantiation.

apply (*func: Callable*) → *bool*

Applies the function *func* to the parser. Returns `False`, if - for whatever reason - the functions has not been applied, `True` otherwise.

grammar () → *object*

Returns the Grammar object to which the parser belongs. If not yet connected to any Grammar object, `None` is returned.

name

Returns the name of the parser or the empty string "" for unnamed parsers.

p_type

Returns the type of the parser. By default this is the parser's class name preceded by a colon, e.g. `'ZeroOrMore'`.

repr

Returns the parser's name if it has a name and `repr()`

reset ()

Resets any parser variables. (Should be overridden.)

class MockParser (*name="", p_type=""*)

`MockParser` objects can be used to reconstruct syntax trees from a serialized form like S-expressions or XML. Mock objects can mimic different parser types by assigning them a `p_type` on initialization.

Mock objects should not be used for anything other than syntax tree (re-)construction. In all other cases where a parser object substitute is needed, chose the singleton `ZOMBIE_PARSER`.

class ZombieParser

Serves as a substitute for a `Parser` instance.

`ZombieParser` is the class of the singleton object `ZOMBIE_PARSER`. The `ZOMBIE_PARSER` has a name and can be called, but it never matches. It serves as a substitute where only these (or one of these properties) is needed, but no real `Parser`- object is instantiated.

class Node (*parser, result: Union[typing.Tuple[_ForwardRef('Node'), ...], _ForwardRef('Node')], DH-Parser.stringview.StringView, str, NoneType], leafhint: bool = False*) → *None*

Represents a node in the concrete or abstract syntax tree.

tag_name

str – The name of the node, which is either its parser’s name or, if that is empty, the parser’s class name

result

str or tuple – The result of the parser which generated this node, which can be either a string or a tuple of child nodes.

children

tuple – The tuple of child nodes or an empty tuple if there are no child nodes. READ ONLY!

content

str – Yields the contents of the tree as string. The difference to `str(node)` is that `node.content` does not add the error messages to the returned string.

parser

Parser – The parser which generated this node. WARNING: In case you use mock syntax trees for testing or parser replacement during the AST-transformation: DO NOT rely on this being a real parser object in any phase after parsing (i.e. AST-transformation and compiling), for example by calling `isinstance(node.parser, ...)`.

errors

list – A list of parser- or compiler-errors: tuple(position, string) attached to this node

error_flag

int – 0 if no error occurred in either the node itself or any of its descendants. Otherwise contains the highest warning or error level or all errors that occurred.

len

int – The full length of the node’s string result if the node is a leaf node or, otherwise, the concatenated string result’s of its descendants. The figure always represents the length before AST-transformation and will never change through AST-transformation. READ ONLY!

pos

int – the position of the node within the parsed text.

The value of `pos` is -1 meaning invalid by default. Setting this value will set the positions of all child nodes relative to this value.

To set the `pos` values of all nodes in a syntax tree, the `pos` value of the root node should be set to 0 right after parsing.

Other than that, this value should be considered READ ONLY. At any rate, it should only be reassigned during the parsing stage and never during or after the AST-transformation.

attributes

dict – An optional dictionary of XML-attributes. This dictionary is created lazily upon first usage. The attributes will only be shown in the XML-Representation, not in the S-Expression-output.

_parent

Node – SLOT RESERVED FOR FUTURE USE!

add_error (*message: str, code: int = 1000*) → *syntaxtree.Node*

Adds an error to this Node. :param message: A string with the error message. :type message: str :param code: An error code to identify the kind of error :type code: int

as_sxpr (*src: str = None, compact: bool = False, showerrors: bool = True*) → *str*

Returns content as S-expression, i.e. in lisp-like form.

Parameters

- **src** – The source text or *None*. In case the source text is given the position of the element in the text will be reported as line and column.

- **compact** – If True a compact representation is returned where brackets are omitted and only the indentation indicates the tree structure.

as_xml (*src: str = None, showerrors: bool = True*) → str

Returns content as XML-tree.

Parameters *src* – The source text or *None*. In case the source text is given the position will also be reported as line and column.

attributes

Returns a dictionary of XML-Attributes attached to the Node.

collect_errors (*clear_errors=False*) → List[DHParse.error.Error]

Recursively adds line- and column-numbers to all error objects. Returns all errors of this node or any child node in the form of a set of tuples (position, error_message), where position is always relative to this node.

content

Returns content as string, omitting error messages.

errors

Returns the errors that occurred at this Node, not including any errors from child nodes.

init_pos (*pos: int, overwrite: bool = False*) → syntaxtree.Node

(Re-)initialize position value. Usually, the parser guard (*parsers.add_parser_guard()*) takes care of assigning the position in the document to newly created nodes. However, where Nodes are created outside the reach of the parser guard, their document-position must be assigned manually. This function recursively reassigns the position values of the child nodes, too.

pick (*tag_names: Union[str, typing.Set[str]]*) → Union[_ForwardRef('Node'), NoneType]

Picks the first descendant with one of the given tag_names.

This function is just syntactic sugar for `next(node.select_by_tag(tag_names, False))`. However, rather than raising a `StopIterationError` if no descendant with the given tag-name exists, it returns `None`.

pos

Returns the position of the Node's content in the source text.

result

Returns the result from the parser that created the node. Error messages are not included in the result. Use *self.content()* if the result plus any error messages is needed.

select (*match_function: Callable, include_root: bool = True*) → Iterator[_ForwardRef('Node')]

Finds nodes in the tree that fulfill a given criterion.

select is a generator that yields all nodes for which the given *match_function* evaluates to True. The tree is traversed pre-order.

See function *Node.select_by_tag* for some examples.

Parameters

- **match_function** (*function*) – A function that takes as Node object as argument and returns True or False
- **include_root** (*bool*) – If False, only descendant nodes will be checked for a match.

Yields *Node* – All nodes of the tree for which *match_function(node)* returns True

select_by_tag (*tag_names: Union[str, typing.Set[str]], include_root: bool = True*) → Iterator[_ForwardRef('Node')]

Returns an iterator that runs through all descendants that have one of the given tag names.

Examples:

```

>>> tree = mock_syntax_tree('(a (b "X") (X (c "d")) (e (X "F")))' )
>>> list(flatten_sxpr(item.as_sxpr()) for item in tree.select_by_tag("X",
↪False))
['(X (c "d"))', '(X "F")']
>>> list(flatten_sxpr(item.as_sxpr()) for item in tree.select_by_tag({"X", "b
↪"}, False))
['(b "X")', '(X (c "d"))', '(X "F")']
>>> any(tree.select_by_tag('a', False))
False
>>> list(flatten_sxpr(item.as_sxpr()) for item in tree.select_by_tag('a',
↪True))
['(a (b "X") (X (c "d")) (e (X "F")))' ]
>>> flatten_sxpr(next(tree.select_by_tag("X", False)).as_sxpr())
'(X (c "d"))'

```

Parameters

- **tag_name** (*set*) – A tag name or set of tag names that is being searched for
- **include_root** (*bool*) – If False, only descendant nodes will be checked for a match.

Yields *Node* – All nodes which have a given tag name.

structure

Return structure (and content) as S-expression on a single line without any line breaks.

tag_name

Returns the tag name of Node, i.e. the name for XML or S-expression representation. By default the tag name is the name of the node's parser or, if the node's parser is unnamed, the node's parser's *type*.

tree_size () → int

Recursively counts the number of nodes in the tree including the root node.

mock_syntax_tree (sxpr)

Generates a tree of nodes from an S-expression. The main purpose of this is to generate test data.

Example: >>> mock_syntax_tree("(a (b c))").as_sxpr() '(an (bn "c"n)n)'

flatten_sxpr (sxpr: str) → str

Returns S-expression *sxpr* as a one-liner without unnecessary whitespace.

Example: >>> flatten_sxpr('(an (bn cn)n)n') '(a (b c))'

4.1.3 Module parse

Module `parse` contains the python classes and functions for DHParser's packrat-parser. It's central class is the `Grammar`-class, which is the base class for any concrete Grammar. Grammar-objects are callable and parsing is done by calling a Grammar-object with a source text as argument.

The different parsing functions are callable descendants of class `Parser`. Usually, they are organized in a tree and defined within the namespace of a grammar-class. See `ebnf.EBNFGrammar` for an example.

class Parser (name: str = "") → None

(Abstract) Base class for Parser combinator parsers. Any parser object that is actually used for parsing (i.e. no mock parsers) should be derived from this class.

Since parsers can contain other parsers (see classes `UnaryOperator` and `NaryOperator`) they form a cyclical directed graph. A root parser is a parser from which all other parsers can be reached. Usually, there is one root

parser which serves as the starting point of the parsing process. When speaking of “the root parser” it is this root parser object that is meant.

There are two different types of parsers:

1. *Named parsers* for which a name is set in field `parser.name`. The results produced by these parsers can later be retrieved in the AST by the parser name.
2. *Anonymous parsers* where the name-field just contains the empty string. AST-transformation of Anonymous parsers can be hooked only to their class name, and not to the individual parser.

Parser objects are callable and parsing is done by calling a parser object with the text to parse.

If the parser matches it returns a tuple consisting of a node representing the root of the concrete syntax tree resulting from the match as well as the substring `text[i:]` where `i` is the length of matched text (which can be zero in the case of parsers like *ZeroOrMore* or *Option*). If `i > 0` then the parser has “moved forward”.

If the parser does not match it returns `(None, text)`. ***Note** that this is not the same as an empty match* `(“”, text)`. Any empty match can for example be returned by the *ZeroOrMore*-parser in case the contained parser is repeated zero times.

Attributes and Properties:

visited: Mapping of places this parser has already been to during the current parsing process onto the results the parser returned at the respective place. This dictionary is used to implement memoizing.

recursion_counter: Mapping of places to how often the parser has already been called recursively at this place. This is needed to implement left recursion. The number of calls becomes irrelevant once a result has been memoized.

cycle_detection: The `apply()`-method uses this variable to make sure that one and the same function will not be applied (recursively) a second time, if it has already been applied to this parser.

grammar: A reference to the Grammar object to which the parser is attached.

ApplyFunc

alias of `typing.Callable`

apply (*func: Callable[_ForwardRef('Parser'), NoneType]*) → bool

Applies function `func(parser)` recursively to this parser and all descendant parsers if any exist. The same function can never be applied twice between calls of the `reset()`-method! Returns *True*, if function has been applied, *False* if function had been applied earlier already and thus has not been applied again.

grammar

Returns the Grammar object to which the parser belongs. If not yet connected to any Grammar object, *None* is returned.

reset()

Initializes or resets any parser variables. If overwritten, the `reset()`-method of the parent class must be called from the `reset()`-method of the derived class.

exception UnknownParserError

UnknownParserError is raised if a Grammar object is called with a parser that does not exist or if in the course of parsing a parser is referred to that does not exist.

class Grammar (*root: parse.Parser = None*) → None

Class Grammar directs the parsing process and stores global state information of the parsers, i.e. state information that is shared across parsers.

Grammars are basically collections of parser objects, which are connected to an instance object of class Grammar. There exist two ways of connecting parsers to grammar objects: Either by passing the root parser object to the constructor of a Grammar object (“direct instantiation”), or by assigning the root parser to the class variable “**root__**” of a descendant class of class Grammar.

Example for direct instantiation of a grammar:

```
>>> number = RE('\d+') + RE('\. ') + RE('\d+') | RE('\d+')
>>> number_parser = Grammar(number)
>>> number_parser("3.1416").content
'3.1416'
```

Collecting the parsers that define a grammar in a descendant class of class `Grammar` and assigning the named parsers to class variables rather than global variables has several advantages:

1. It keeps the namespace clean.
2. The parser names of named parsers do not need to be passed to the constructor of the `Parser` object explicitly, but it suffices to assign them to class variables, which results in better readability of the Python code.
3. The parsers in the class do not necessarily need to be connected to one single root parser, which is helpful for testing and building up a parser successively of several components.

As a consequence, though, it is highly recommended that a `Grammar` class should not define any other variables or methods with names that are legal parser names. A name ending with a double underscore `'__'` is *not* a legal parser name and can safely be used.

Example:

```
class Arithmetic(Grammar):
    # special fields for implicit whitespace and comment configuration
    COMMENT__ = r'#.*(?:\n|$)' # Python style comments
    wspR__ = mixin_comment(whitespace=r'[\t ]*', comment=COMMENT__)

    # parsers
    expression = Forward()
    INTEGER = RE('\d+')
    factor = INTEGER | Token("(") + expression + Token(")")
    term = factor + ZeroOrMore((Token("*") | Token("/")) + factor)
    expression.set(term + ZeroOrMore((Token("+") | Token("-")) + term))
    root__ = expression
```

Upon instantiation the parser objects are deep-copied to the `Grammar` object and assigned to object variables of the same name. Any parser that is directly assigned to a class variable is a ‘named’ parser and its field `parser.name` contains the variable name after instantiation of the `Grammar` class. All other parsers, i.e. parsers that are defined within a *named* parser, remain “anonymous parsers” where `parser.name` is the empty string, unless a name has been passed explicitly upon instantiation. If one and the same parser is assigned to several class variables such as, for example the parser `expression` in the example above, the first name sticks.

`Grammar` objects are callable. Calling a grammar object with a UTF-8 encoded document, initiates the parsing of the document with the root parser. The return value is the concrete syntax tree. `Grammar` objects can be reused (i.e. called again) after parsing. Thus, it is not necessary to instantiate more than one `Grammar` object per thread.

`Grammar` classes contain a few special class fields for implicit whitespace and comments that should be overwritten, if the defaults (no comments, horizontal right aligned whitespace) don’t fit:

COMMENT__
regular expression string for matching comments

WSP__
regular expression for whitespace and comments

wspL__
regular expression string for left aligned whitespace, which either equals **WSP__** or is empty.

wspR__

regular expression string for right aligned whitespace, which either equals **WSP__** or is empty.

root__

The root parser of the grammar. Theoretically, all parsers of the grammar should be reachable by the root parser. However, for testing of yet incomplete grammars class Grammar does not assume that this is the case.

parser_initialization__

Before the parser class (!) has been initialized, which happens upon the first time it is instantiated (see :func:assign_parser_names() for an explanation), this class field contains a value other than “done”. A value of “done” indicates that the class has already been initialized.

all_parsers__

A set of all parsers connected to this grammar object

history_tracking__

A flag indicating that the parsing history shall be tracked

wsp_left_parser__

A parser for the default left-adjacent-whitespace or the :class:zombie-parser if the default is empty. The default whitespace will be used by parsers *Token* and, if no other parsers are passed to its constructor, by parser *RE*.

wsp_right_parser__

The same for the default right-adjacent-whitespace. Both **wsp_left_parser__** and **wsp_right_parser__** merely serve the purpose to avoid having to specify the default whitespace explicitly every time an *RE*-parser-object is created.

_dirty_flag__

A flag indicating that the Grammar has been called at least once so that the parsing-variables need to be reset when it is called again.

document__

the text that has most recently been parsed or that is currently being parsed.

document_length__

the length of the document.

document_lbreaks__

list of linebreaks within the document, starting with -1 and ending with EOF. This helps generating line and column number for history recording and will only be initialized if *history_tracking__* is true.

_reversed__

the same text in reverse order - needed by the *Lookbehind*-parsers.

variables__

A mapping for variable names to a stack of their respective string values - needed by the *Capture*-, *Retrieve*- and *Pop*-parsers.

rollback__

A list of tuples (location, rollback-function) that are deposited by the *Capture*- and *Pop*-parsers. If the parsing process reaches a dead end then all rollback-functions up to the point to which it retreats will be called and the state of the variable stack restored accordingly.

last_rb_loc__

The last, i.e. most advanced location in the text where a variable changing operation occurred. If the parser backtracks to a location at or before *last_rb_loc__* (i.e. location <= *last_rb_loc__*) then a rollback of all variable changing operations is necessary that occurred after the location to which the parser backtracks. This is done by calling method `rollback_to__(location)()`.

call_stack__

A stack of all parsers that have been called. This is required for recording the parser history (for debugging) and, eventually, i.e. one day in the future, for tracing through the parsing process.

history__

A list of parser-call-stacks. A parser-call-stack is appended to the list each time a parser either matches, fails or if a parser-error occurs.

moving_forward__

This flag indicates that the parsing process is currently moving forward . It is needed to reduce noise in history recording and should not be considered as having a valid value if history recording is turned off! (See `add_parser_guard()` and its local function `guarded_call()`)

recursion_locations__

Stores the locations where left recursion was detected. Needed to provide minimal memoization for the left recursion detection algorithm, but, strictly speaking, superfluous if full memoization is enabled. (See `add_parser_guard()` and its local function `guarded_call()`)

memoization__

Turns full memoization on or off. Turning memoization off results in less memory usage and sometimes reduced parsing time. In some situations it may drastically increase parsing time, so it is safer to leave it on. (Default: on)

left_recursion_handling__

Turns left recursion handling on or off. If turned off, a recursion error will result in case of left recursion.

push_rollback__ (*location, func*)

Adds a rollback function that either removes or re-adds values on the variable stack (*self.variables*) that have been added (or removed) by Capture or Pop Parsers, the results of which have been dismissed.

reversed__

Returns a reversed version of the currently parsed document. As about the only case where this is needed is the Lookbehind-parser, this is done lazily.

rollback_to__ (*location*)

Rolls back the variable stacks (*self.variables*) to its state at an earlier location in the parsed document.

class PreprocessorToken (*token: str*) → None

Parses tokens that have been inserted by a preprocessor.

Preprocessors can generate Tokens with the `make_token`-function. These tokens start and end with magic characters that can only be matched by the PreprocessorToken Parser. Such tokens can be used to insert BEGIN - END delimiters at the beginning or ending of a quoted block, for example.

class RegExp (*regexp, name: str = ""*) → None

Regular expression parser.

The RegExp-parser parses text that matches a regular expression. RegExp can also be considered as the “atomic parser”, because all other parsers delegate part of the parsing job to other parsers, but do not match text directly.

Example:

```
>>> word = RegExp(r'\w+')
>>> Grammar(word) ("Haus").content
'Haus'
```

EBNF-Notation: / ... /

EBNF-Example: word = /\w+ /

class RE (*regexp, wL=None, wR=None, name: str = ""*) → None

Regular Expressions with optional leading or trailing whitespace.

The RE-parser parses pieces of text that match a given regular expression. Other than the `RegExp-Parser` it can also skip “implicit whitespace” before or after the matched text.

The whitespace is in turn defined by a regular expression. It should be made sure that this expression also matches the empty string, e.g. use `r's*` or `r'[t]+`, but not `r's+`. If the respective parameters in the constructor are set to `None` the default whitespace expression from the Grammar object will be used.

Example (allowing whitespace on the right hand side, but not on the left hand side of a regular expression):

```
>>> word = RE(r'\w+', wR=r'\s*')
>>> parser = Grammar(word)
>>> result = parser('Haus ')
>>> result.content
'Haus '
>>> result.structure
'(:RE (:RegExp "Haus") (:Whitespace " "))'
>>> str(parser(' Haus'))
' <<< Error on " Haus" | Parser did not match! Invalid source file?\n      Most_
->advanced: None\n      Last match:      None; >>> '
```

EBNF-Notation: / ... /~` or `~/ ... /` or `~/ ... /~

EBNF-Example: `word = /\w+/~`

apply (*func: Callable[_ForwardRef('Parser'), NoneType]*) → bool

Applies function *func(parser)* recursively to this parser and all descendant parsers if any exist. The same function can never be applied twice between calls of the `reset()`-method! Returns *True*, if function has been applied, *False* if function had been applied earlier already and thus has not been applied again.

create_main_parser (*arg*) → `parse.Parser`

Creates the main parser of this compound parser. Can be overridden.

class Token (*token: str, wL=None, wR=None, name: str = ""*) → `None`

Class `Token` parses simple strings. Any regular regular expression commands will be interpreted as simple sequence of characters.

Other than that class `Token` is essentially a renamed version of class `RE`. Because tokens often have a particular semantic different from other REs, parsing them with a separate parser class allows to distinguish them by their parser type.

create_main_parser (*arg*) → `parse.Parser`

Creates the main parser of this compound parser. Can be overridden.

mixin_comment (*whitespace: str, comment: str*) → `str`

Returns a regular expression that merges comment and whitespace regexps. Thus comments can occur wherever whitespace is allowed and will be skipped just as implicit whitespace.

Note, that because this works on the level of regular expressions, nesting comments is not possible. It also makes it much harder to use directives inside comments (which isn't recommended, anyway).

class Synonym (*parser: parse.Parser, name: str = ""*) → `None`

Simply calls another parser and encapsulates the result in another node if that parser matches.

This parser is needed to support synonyms in EBNF, e.g.:

```
jahr          = JAHRESZAHL
JAHRESZAHL    = /\d\d\d\d/
```

Otherwise the first line could not be represented by any parser class, in which case it would be unclear whether the parser `RE('dddd')` carries the name 'JAHRESZAHL' or 'jahr'.

class Option (*parser: parse.Parser, name: str = ""*) → None

Parser *Option* always matches, even if its child-parser did not match.

If the child-parser did not match *Option* returns a node with no content and does not move forward in the text.

If the child-parser did match, *Option* returns the a node with the node returned by the child-parser as its single child and the text at the position where the child-parser left it.

Examples:

```
>>> number = Option(Token('-')) + RegExp(r'\d+') + Option(RegExp(r'\.\d+'))
>>> Grammar(number)('3.14159').content
'3.14159'
>>> Grammar(number)('3.14159').structure
'(:Series (:Option) (:RegExp "3") (:Option (:RegExp ".14159")))'
>>> Grammar(number)('-1').content
'-1'
```

EBNF-Notation: [...]

EBNF-Example: number = ["-"] /\d+ / [/\.\d+ /]

class ZeroOrMore (*parser: parse.Parser, name: str = ""*) → None

ZeroOrMore applies a parser repeatedly as long as this parser matches. Like *Option* the *ZeroOrMore* parser always matches. In case of zero repetitions, the empty match (*()*, *text*) is returned.

Examples:

```
>>> sentence = ZeroOrMore(RE(r'\w+,?')) + Token('.')
>>> Grammar(sentence)('Wo viel der Weisheit, da auch viel des Grämens.').content
'Wo viel der Weisheit, da auch viel des Grämens.'
>>> Grammar(sentence)('.').content # an empty sentence also matches
'.'
```

EBNF-Notation: { ... }

EBNF-Example: sentence = { /\w+,? / } "."

class OneOrMore (*parser: parse.Parser, name: str = ""*) → None

OneOrMore applies a parser repeatedly as long as this parser matches. Other than *ZeroOrMore* which always matches, at least one match is required by *OneOrMore*.

Examples:

```
>>> sentence = OneOrMore(RE(r'\w+,?')) + Token('.')
>>> Grammar(sentence)('Wo viel der Weisheit, da auch viel des Grämens.').content
'Wo viel der Weisheit, da auch viel des Grämens.'
>>> str(Grammar(sentence)('.')) # an empty sentence also matches
' <<< Error on "." | Parser did not match! Invalid source file?\n      Most_
↪advanced: None\n      Last match:      None; >>> '
```

EBNF-Notation: { ... }+

EBNF-Example: sentence = { /\w+,? / }+

class Series (**parsers, mandatory: int = 1000, name: str = ""*) → None

Matches if each of a series of parsers matches exactly in the order of the series.

Example:

```
>>> variable_name = RegExp('(!\d)\w') + RE('\w*')
>>> Grammar(variable_name)('variable_1').content
'variable_1'
>>> str(Grammar(variable_name)('1_variable'))
' <<< Error on "1_variable" | Parser did not match! Invalid source file?\n
↳Most advanced: None\n      Last match:      None; >>> '
```

EBNF-Notation: ... (sequence of parsers separated by a blank or new line)

EBNF-Example: series = letter letter_or_digit

static combined_mandatory (*right: parse.Parser*)

Returns the position of the first mandatory element (if any) when parsers *left* and *right* are joined to a sequence.

class Alternative (*parsers, name: str = "") → None

Matches if one of several alternatives matches. Returns the first match.

This parser represents the EBNF-operator “|” with the qualification that both the symmetry and the ambiguity of the EBNF-or-operator are broken by selecting the first match.:

```
# the order of the sub-expression matters!
>>> number = RE('\d+') | RE('\d+') + RE('\.') + RE('\d+')
>>> str(Grammar(number)('3.1416'))
'3 <<< Error on ".141" | Parser stopped before end! trying to recover... >>> '

# the most selective expression should be put first:
>>> number = RE('\d+') + RE('\.') + RE('\d+') | RE('\d+')
>>> Grammar(number)('3.1416').content
'3.1416'
```

EBNF-Notation: ... | ...

EBNF-Example: sentence = /\d+\.\d+/ | /\d+/

reset ()

Initializes or resets any parser variables. If overwritten, the *reset()*-method of the parent class must be called from the *reset()*-method of the derived class.

class AllOf (*parsers, name: str = "") → None

Matches if all elements of a list of parsers match. Each parser must match exactly once. Other than in a sequence, the order in which the parsers match is arbitrary, however.

Example:

```
>>> prefixes = AllOf(Token("A"), Token("B"))
>>> Grammar(prefixes)('A B').content
'A B'
>>> Grammar(prefixes)('B A').content
'B A'
```

EBNF-Notation: <... > (sequence of parsers enclosed by angular brackets)

EBNF-Example: set = <letter letter_or_digit>

class SomeOf (*parsers, name: str = "") → None

Matches if at least one element of a list of parsers match. No parser must match more than once. Other than in a sequence, the order in which the parsers match is arbitrary, however.

Example:

```
>>> prefixes = SomeOf(Token("A"), Token("B"))
>>> Grammar(prefixes)('A B').content
'A B'
>>> Grammar(prefixes)('B A').content
'B A'
>>> Grammar(prefixes)('B').content
'B'
```

EBNF-Notation: `<... >` (sequence of parsers enclosed by angular brackets)

EBNF-Example: `set = <letter letter_or_digit>`

Unordered (*parser: parse.NaryOperator, name: str = ""*) → *parse.NaryOperator*

Returns an *Allof*- or *SomeOf*-parser depending on whether *parser* is a *Series* (*Allof*) or an *Alternative* (*SomeOf*).

class Lookahead (*parser: parse.Parser, name: str = ""*) → *None*

Matches, if the contained parser would match for the following text, but does not consume any text.

class NegativeLookahead (*parser: parse.Parser, name: str = ""*) → *None*

Matches, if the contained parser would *not* match for the following text.

sign (*bool_value*) → *bool*

Returns the value. Can be overridden to return the inverted bool.

class Lookbehind (*parser: parse.Parser, name: str = ""*) → *None*

Matches, if the contained parser would match backwards. Requires the contained parser to be a *RegExp*, *RE*, *PlainText* or *Token* parser.

EXPERIMENTAL

class NegativeLookbehind (*parser: parse.Parser, name: str = ""*) → *None*

Matches, if the contained parser would *not* match backwards. Requires the contained parser to be a *RegExp*-parser.

sign (*bool_value*) → *bool*

Returns the value. Can be overridden to return the inverted bool.

class Capture (*parser: parse.Parser, name: str = ""*) → *None*

Applies the contained parser and, in case of a match, saves the result in a variable. A variable is a stack of values associated with the contained parser's name. This requires the contained parser to be named.

class Retrieve (*symbol: parse.Parser, rfilter: Callable[List[str], str] = None, name: str = ""*) → *None*

Matches if the following text starts with the value of a particular variable. As a variable in this context means a stack of values, the last value will be compared with the following text. It will not be removed from the stack! (This is the difference between the *Retrieve* and the *Pop* parser.) The constructor parameter *symbol* determines which variable is used.

retrieve_and_match (*text: DHParse.stringview.StringView*) → *Tuple[Union[DHParse.syntaxtree.Node, NoneType], DHParse.stringview.StringView]*

Retrieves variable from stack through the filter function passed to the class' constructor and tries to match the variable's value with the following text. Returns a *Node* containing the value or *None* accordingly.

This functionality has been move from the `__call__` method to an independent method to allow calling it from a subclasses `__call__` method without triggering the parser guard a second time.

class Pop (*symbol: parse.Parser, rfilter: Callable[List[str], str] = None, name: str = ""*) → *None*

Matches if the following text starts with the value of a particular variable. As a variable in this context means a stack of values, the last value will be compared with the following text. Other than the *Retrieve*-parser, the *Pop*-parser removes the value from the stack in case of a match.

The constructor parameter *symbol* determines which variable is used.

class Forward

Forward allows to declare a parser before it is actually defined. Forward declarations are needed for parsers that are recursively nested, e.g.:

```
class Arithmetic(Grammar):
    '''
    expression = term { "+" | "-" } term }
    term       = factor { "*" | "/" } factor }
    factor     = INTEGER | "(" expression ")"
    INTEGER    = /\d+/~
    '''
    expression = Forward()
    INTEGER    = RE('\d+')
    factor     = INTEGER | Token("(") + expression + Token(")")
    term       = factor + ZeroOrMore((Token("*") | Token("/")) + factor)
    expression.set(term + ZeroOrMore((Token("+") | Token("-")) + term))
    root__     = expression
```

apply (*func*: Callable[_ForwardRef('Parser'), NoneType]) → bool

Applies function *func(parser)* recursively to this parser and all descendant parsers if any exist. The same function can never be applied twice between calls of the `reset()`-method! Returns *True*, if function has been applied, *False* if function had been applied earlier already and thus has not been applied again.

set (*parser*: parse.Parser)

Sets the parser to which the calls to this Forward-object shall be delegated.

4.1.4 Module transform

Module `transform` contains the functions for transforming the concrete syntax tree (CST) into an abstract syntax tree (AST).

As these functions are very generic, they can in principle be used for any kind of tree transformations, not necessarily only for CST → AST transformations.

TransformationDict

alias of `typing.Dict`

TransformationProc

alias of `typing.Callable`

ConditionFunc

alias of `typing.Callable`

KeyFunc

alias of `typing.Callable`

transformation_factory (*t1=None, t2=None, t3=None, t4=None, t5=None*)

Creates factory functions from transformation-functions that dispatch on the first parameter after the context parameter.

Decorating a transformation-function that has more than merely the `node`-parameter with `transformation_factory` creates a function with the same name, which returns a partial-function that takes just the context-parameter.

Additionally, there is some syntactic sugar for transformation-functions that receive a collection as their second parameter and do not have any further parameters. In this case a list of parameters passed to the factory function will be converted into a collection.

Main benefit is readability of processing tables.

Usage:

```
@transformation_factory(AbstractSet[str])
def remove_tokens(context, tokens):
    ...
```

or, alternatively:

```
@transformation_factory
def remove_tokens(context, tokens: AbstractSet[str]):
    ...
```

Example:

```
trans_table = { 'expression': remove_tokens('+', '-') }
```

instead of:

```
trans_table = { 'expression': partial(remove_tokens, tokens={'+', '-'}) }
```

Parameters *t1* – type of the second argument of the transformation function, only necessary if the transformation functions’ parameter list does not have type annotations.

traverse (*root_node*: *DHParse.syntaxtree.Node*, *processing_table*: *Dict[str, Union[typing.Sequence[typing.Callable], typing.Dict[str, typing.Sequence[typing.Callable]]], key_func: Callable[DHParse.syntaxtree.Node, str] = <function key_tag_name>)* → None

Traverses the syntax tree starting with the given node depth first and applies the sequences of callback-functions registered in the *processing_table*-dictionary.

The most important use case is the transformation of a concrete syntax tree into an abstract tree (AST). But it is also imaginable to employ tree-traversal for the semantic analysis of the AST.

In order to assign sequences of callback-functions to nodes, a dictionary (“processing table”) is used. The keys usually represent tag names, but any other key function is possible. There exist three special keys:

- ‘+’: always called (before any other processing function)
- ‘*’: called for those nodes for which no (other) processing function appears in the table
- ‘~’: always called (after any other processing function)

Parameters

- **root_node** (*Node*) – The root-node of the syntax tree to be traversed
- **processing_table** (*dict*) – node key -> sequence of functions that will be applied to matching nodes in order. This dictionary is interpreted as a *compact_table*. See `expand_table()` or `EBNFCompiler.EBNFTransTable()`
- **key_func** (*function*) – A mapping *key_func*(node) -> *keystr*. The default *key_func* yields *node.parser.name*.

Example:

```
table = { "term": [replace_by_single_child, flatten],
          "factor, flowmarker, retrieveop": replace_by_single_child }
traverse(node, table)
```

is_named (*context*: List[DHParse.syntaxtree.Node]) → bool

Returns True if the current node's parser is a named parser.

replace_by_single_child (*context*: List[DHParse.syntaxtree.Node])

Removes single branch node, replacing it by its immediate descendant. Replacement only takes place, if the last node in the context has exactly one child.

reduce_single_child (*context*: List[DHParse.syntaxtree.Node])

Reduces a single branch node by transferring the result of its immediate descendant to this node, but keeping this node's parser entry. Reduction only takes place if the last node in the context has exactly one child.

replace_or_reduce (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable = <function is_named>)

Replaces node by a single child, if condition is met on child, otherwise (i.e. if the child is anonymous) reduces the child.

replace_parser (*context*: List[DHParse.syntaxtree.Node], *name*: str)

Replaces the parser of a Node with a mock parser with the given name.

Parameters

- **context** – the context where the parser shall be replaced
- **name** – “NAME:PTYPE” of the surrogate. The ptype is optional

collapse (*context*: List[DHParse.syntaxtree.Node])

Collapses all sub-nodes of a node by replacing them with the string representation of the node.

merge_children (*context*: List[DHParse.syntaxtree.Node], *tag_names*: List[str])

Joins all children next to each other and with particular tag-names into a single child node with a mock-parser with the name of the first tag-name in the list.

replace_content (*context*: List[DHParse.syntaxtree.Node], *func*: Callable)

Replaces the content of the node. *func* takes the node's result as an argument and returns the mapped result.

apply_if (*context*: List[DHParse.syntaxtree.Node], *transformation*: Callable, *condition*: Callable)

Applies a transformation only if a certain condition is met.

apply_unless (*context*: List[DHParse.syntaxtree.Node], *transformation*: Callable, *condition*: Callable)

Applies a transformation if a certain condition is *not* met.

traverse_locally (*context*: List[DHParse.syntaxtree.Node], *processing_table*: Dict, *key_func*: Callable = <function key_tag_name>)

Transforms the syntax tree starting from the last node in the context according to the given processing table. The purpose of this function is to apply certain transformations locally, i.e. only for those nodes that have the last node in the context as their parent node.

is_anonymous (*context*: List[DHParse.syntaxtree.Node]) → bool

Returns True if the current node's parser is an anonymous parser.

is_whitespace (*context*: List[DHParse.syntaxtree.Node]) → bool

Removes whitespace and comments defined with the @comment-directive.

is_empty (*context*: List[DHParse.syntaxtree.Node]) → bool

Returns True if the current node's content is empty.

is_expendable (*context*: List[DHParse.syntaxtree.Node]) → bool

Returns True if the current node either is a node containing whitespace or an empty node.

is_token (*context*: List[DHParse.syntaxtree.Node], *tokens*: AbstractSet[str] = frozenset()) → bool

Checks whether the last node in the context has *ptype* == *TOKEN_PTYPE* and its content matches one of the given tokens. Leading and trailing whitespace-tokens will be ignored. In case an empty set of tokens is passed, any token is a match. If only “:” is given all anonymous tokens but no other tokens are a match.

is_one_of (*context*: List[DHParse.syntaxtree.Node], *tag_name_set*: AbstractSet[str]) → bool

Returns true, if the node's tag_name is one of the given tag names.

has_content (*context*: List[DHParse.syntaxtree.Node], *regexp*: str) → bool

Checks a node's content against a regular expression.

In contrast to `re.match` the regular expression must match the complete string and not just the beginning of the string to succeed!

has_parent (*context*: List[DHParse.syntaxtree.Node], *tag_name_set*: AbstractSet[str]) → bool

Checks whether a node with one of the given tag names appears somewhere in the context before the last node in the context.

lstrip (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable = <function is_expendable>)

Recursively removes all leading child-nodes that fulfill a given condition.

rstrip (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable = <function is_expendable>)

Recursively removes all leading nodes that fulfill a given condition.

strip (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable = <function is_expendable>)

Removes leading and trailing child-nodes that fulfill a given condition.

keep_children (*context*: List[DHParse.syntaxtree.Node], *section*: slice = slice(None, None, None))

Keeps only child-nodes which fall into a slice of the result field.

keep_children_if (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable)

Removes all children for which *condition()* returns *True*.

keep_tokens (*context*: List[DHParse.syntaxtree.Node], *tokens*: AbstractSet[str] = frozenset())

Removes any among a particular set of tokens from the immediate descendants of a node. If *tokens* is the empty set, all tokens are removed.

keep_nodes (*context*: List[DHParse.syntaxtree.Node], *tag_names*: AbstractSet[str])

Removes children by tag name.

keep_content (*context*: List[DHParse.syntaxtree.Node], *regexp*: str)

Removes children depending on their string value.

remove_children_if (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable)

Removes all children for which *condition()* returns *True*.

remove_nodes (*context*: List[DHParse.syntaxtree.Node], *tag_names*: AbstractSet[str])

Removes children by tag name.

remove_content (*context*: List[DHParse.syntaxtree.Node], *regexp*: str)

Removes children depending on their string value.

remove_tokens (*context*: List[DHParse.syntaxtree.Node], *tokens*: AbstractSet[str] = frozenset())

Removes any among a particular set of tokens from the immediate descendants of a node. If *tokens* is the empty set, all tokens are removed.

flatten (*context*: List[DHParse.syntaxtree.Node], *condition*: Callable = <function is_anonymous>, *recursive*: bool = True)

Flattens all children, that fulfil the given *condition* (default: all unnamed children). Flattening means that wherever a node has child nodes, the child nodes are inserted in place of the node.

If the parameter *recursive* is *True* the same will recursively be done with the child-nodes, first. In other words, all leaves of this node and its child nodes are collected in-order as direct children of this node.

Applying flatten recursively will result in these kinds of structural transformation:

(1 (+ 2) (+ 3))	->	(1 + 2 + 3)
(1 (+ (2 + (3))))	->	(1 + 2 + 3)

assert_condition (*context*: List[DHParseer.syntaxtree.Node], *condition*: Callable, *error_msg*: str = "")

Checks for *condition*; adds an error message if condition is not met.

4.1.5 Module compile

Module `compile` contains a skeleton class for syntax driven compilation support. Class `Compiler` can serve as base class for a compiler. Compiler objects are callable and receive the Abstract syntax tree (AST) as argument and yield whatever output the compiler produces. In most Digital Humanities applications this will be XML-code. However, it can also be anything else, like binary code or, as in the case of DHParseer's EBNF-compiler, Python source code.

Function `compile_source` invokes all stages of the compilation process, i.e. pre-processing, parsing, CST to AST-transformation and compilation.

See module `ebnf` for a sample of the implementation of a compiler object.

exception CompilerError

Exception raised when an error of the compiler itself is detected. Compiler errors are not to be confused with errors in the source code to be compiled, which do not raise Exceptions but are merely reported as an error.

class Compiler (*grammar_name*="", *grammar_source*="")

Class `Compiler` is the abstract base class for compilers. Compiler objects are callable and take the root node of the abstract syntax tree (AST) as argument and return the compiled code in a format chosen by the compiler itself.

Subclasses implementing a compiler must define *on_XXX()*-methods for each node name that can occur in the AST where 'XXX' is the node's name (for unnamed nodes it is the node's ptype without the leading colon ':').

These compiler methods take the node on which they are run as argument. Other than in the AST transformation, which runs depth-first, compiler methods are called forward moving starting with the root node, and they are responsible for compiling the child nodes themselves. This should be done by invoking the *compile(node)*-method which will pick the right *on_XXX*-method. It is not recommended to call the *on_XXX*-methods directly.

context

A list of parent nodes that ends with the currently compiled node.

grammar_name

The name of the grammar this compiler is related to

grammar_source

The source code of the grammar this compiler is related to.

_dirty_flag

A flag indicating that the compiler has already been called at least once and that therefore all compilation variables must be reset when it is called again.

compile (*node*: DHParseer.syntaxtree.Node) → Any

Calls the compilation method for the given node and returns the result of the compilation.

The method's name is derived from either the node's parser name or, if the parser is anonymous, the node's parser's class name by adding the prefix *on_*.

Note that `compile` does not call any compilation functions for the parsers of the sub nodes by itself. Rather, this should be done within the compilation methods.

fallback_compiler (*node*: DHParseer.syntaxtree.Node) → Any

This is a generic compiler function which will be called on all those node types for which no compiler method *on_XXX* has been defined.

static method_name () → str

Returns the method name for *node_name*, e.g.:

```
>>> Compiler.method_name('expression')
```

```
'on_expression'
```

static propagate_error_flags (*lazy: bool = True*) → None

Propagates error flags from children to parent nodes to make sure that the parent's error flag is always greater or equal the maximum of the children's error flags.

set_grammar_name (*grammar_name="", grammar_source=""*)

Changes the grammar's name and the grammar's source.

The grammar name and the source text of the grammar are metadata about the grammar that do not affect the compilation process. Classes inheriting from *Compiler* can use this information to name and annotate its output.

compile_source (*source: str, preprocessor: Union[typing.Callable[[str], typing.Union[str, typing.Tuple[str, typing.Union[typing.Callable[[int], int], functools.partial]]], functools.partial, NoneType], parser: DHParse.parse.Grammar, transformer: Union[typing.Callable[[DHParse.syntaxtree.Node], typing.Any], functools.partial], compiler: compile.Compiler)* → Tuple[[Any, List[DHParse.error.Error]], DHParse.syntaxtree.Node]

Compiles a source in four stages: 1. Pre-Processing (if needed) 2. Parsing 3. AST-transformation 4. Compiling.

The compilations stage is only invoked if no errors occurred in either of the two previous stages.

Parameters

- **source** (*str*) – The input text for compilation or a the name of a file containing the input text.
- **preprocessor** (*function*) – text -> text. A preprocessor function or None, if no preprocessor is needed.
- **parser** (*function*) – A parsing function or grammar class
- **transformer** (*function*) – A transformation function that takes the root-node of the concrete syntax tree as an argument and transforms it (in place) into an abstract syntax tree.
- **compiler** (*function*) – A compiler function or compiler class instance

Returns (tuple): The result of the compilation as a 3-tuple (result, errors, abstract syntax tree). In detail: 1. The result as returned by the compiler or None in case of failure 2. A list of error or warning messages 3. The root-node of the abstract syntax tree

4.1.6 Module error

Module `error` defines class `Error` and a few helpful functions that are needed for error reporting of `DHParse`. Usually, what is of interest are the string representations of the error objects. For example:

```
from DHParse import compile_source, has_errors

result, errors, ast = compile_source(source, preprocessor, grammar,
                                     transformer, compiler)

if errors:
    for error in errors:
        print(error)

if has_errors(errors):
```

(continues on next page)

(continued from previous page)

```

print("There have been fatal errors!")
sys.exit(1)
else:
    print("There have been warnings, but no errors.")

```

is_error (*code: int*) → bool

Returns True, if error is an error, not just a warning.

is_warning (*code: int*) → bool

Returns True, if error is merely a warning.

has_errors (*messages: Iterable[error.Error], level: int = 1000*) → bool

Returns True, if at least one entry in *messages* has at least the given error *level*.

only_errors (*messages: Iterable[error.Error], level: int = 1000*) → Iterator[error.Error]

Returns an Iterator that yields only those messages that have at least the given error level.

linebreaks (*text: Union[DHParse.stringview.StringView, str]*) → List[int]

Returns a list of indices all line breaks in the text.

line_col (*lbreaks: List[int], pos: int*) → Tuple[int, int]

Returns the position within a text as (line, column)-tuple based on a list of all line breaks, including -1 and EOF.

adjust_error_locations (*errors: List[error.Error], original_text: Union[DHParse.stringview.StringView, str], source_mapping: Union[typing.Callable[[int], int], functools.partial] = <function <lambda>>>*) → List[error.Error]

Adds (or adjusts) line and column numbers of error messages in place.

Parameters

- **errors** – The list of errors as returned by the method `collect_errors()` of a Node object
- **original_text** – The source text on which the errors occurred. (Needed in order to determine the line and column numbers.)
- **source_mapping** – A function that maps error positions to their positions in the original source file.

Returns The list of errors. (Returning the list of errors is just syntactical sugar. Be aware that the `line`, `col` and `orig_pos` attributes have been changed in place.)

4.2 Domain Specific Language Modules Reference

DHParse contains additional support for domain specific languages. Module `ebnf` provides a self-hosting parser for EBNF-Grammars as well as an EBNF-compiler that compiles an EBNF-Grammar into a DHParse based Grammar class that can be executed to parse source text conforming to this grammar into concrete syntax trees.

Module `dsl` contains additional functions to support the compilation of arbitrary domain specific languages (DSL).

One very indispensable part of the systematic construction of domain specific languages is testing. DHParse supports unit testing of smaller as well as larger components of the Grammar of a DSL.

4.2.1 Module ebnf

Module `ebnf` provides a self-hosting parser for EBNF-Grammars as well as an EBNF-compiler that compiles an EBNF-Grammar into a DHParser based Grammar class that can be executed to parse source text conforming to this grammar into concrete syntax trees.

```
class EBNFGrammar (root: DHParser.parse.Parser = None)  $\rightarrow$  None
```

Parser for an EBNF source file, with this grammar:

```
# EBNF-Grammar in EBNF

@ comment      =  /#.*(?:\n|$)/          # comments start with '#' and
                                           # eat all chars up to and including
↳ '\n'

@ whitespace =  /\s*/                    # whitespace includes linefeed
@ literalws   =  right                    # trailing whitespace of literals
↳ will be

                                           # ignored tacitly

syntax        =  [~//] { definition | directive } $EOF
definition    =  symbol $"=" expression
directive     =  "@ " $symbol "=" ( regexp | literal | list_ )

expression    =  term { "|" term }
term          =  { ["$"] factor }+        # "$" means all following factors
↳ mandatory

factor        =  [flowmarker] [retrieveop] symbol !"="  # negative lookahead to be
↳ sure                                                # it's not a definition

                | [flowmarker] literal
                | [flowmarker] regexp
                | [flowmarker] oneormore
                | [flowmarker] group
                | [flowmarker] unordered
                | repetition
                | option

flowmarker    =  "!" | "&"                # '!' negative lookahead, '&'
↳ positive lookahead
                | "-!" | "-&"            # '-' negative lookbehind, '-&'
↳ positive lookbehind

retrieveop    =  "::" | ":"                # '::' pop, ':' retrieve

group         =  "(" $expression ")"
unordered    =  "<" $expression ">"      # elements of expression in
↳ arbitrary order

oneormore    =  "{" expression }+"
repetition   =  "{" $expression "}"
option       =  "[" $expression "]"

symbol       =  /(?:!d)\w+/~              # e.g. expression, factor, parameter_
↳ list

literal      =  /"(?:[^\]|\\")*?"/~       # e.g. "(", '+', 'while'
                | /'(?:[^\']|\\')*?'/~    # whitespace following literals will
↳ be ignored

regexp       =  /~?\/(?:\\\/|[\^\/])?*\~?\/~?~?/  # e.g. /\w+/, ~/#.*(?:\n|$)/~
↳ present leading
                # '~' is a whitespace-marker, if
```

(continues on next page)

(continued from previous page)

```

↪regular expression                                # or trailing whitespace of a_
list_      = /\w+/\~ { ", " /\w+/\~ }             # will be ignored tacitly.
                                                    # comma separated list of symbols,
                                                    # e.g. BEGIN_LIST, END_LIST,
                                                    # BEGIN_QUOTE, END_QUOTE
                                                    # see CommonMark/markdown.py for an_
↪example
EOF = !/./

```

exception EBNFCompilerError

Error raised by *EBNFCompiler* class. (Not compilation errors in the strict sense, see *CompilationError* in module *dsl.py*)

class EBNFCompiler (*grammar_name=""*, *grammar_source=""*)

Generates a Parser from an abstract syntax tree of a grammar specified in EBNF-Notation.

Instances of this class must be called with the root-node of the abstract syntax tree from an EBNF-specification of a formal language. The returned value is the Python-source-code of a Grammar class for this language that can be used to parse texts in this language. See classes *parser.Compiler* and *parser.Grammar* for more information.

Additionally, class *EBNFCompiler* provides helper methods to generate code-skeletons for a preprocessor, AST-transformation and full compilation of the formal language. These method's names start with the prefix *gen_*.

current_symbols

During compilation, a list containing the root node of the currently compiled definition as first element and then the nodes of the symbols that are referred to in the currently compiled definition.

rules

Dictionary that maps rule names to a list of Nodes that contain symbol-references in the definition of the rule. The first item in the list is the node of the rule- definition itself. Example:

alternative = a | b

Now *[node.content for node in self.rules['alternative']]* yields *['alternative = a | b', 'a', 'b']*

symbols

A mapping of symbol names to their first usage (not their definition!) in the EBNF source.

variables

A set of symbols names that are used with the Pop or Retrieve operator. Because the values of these symbols need to be captured they are called variables. See *test_parser.TestPopRetrieve* for an example.

recursive

A set of symbols that are used recursively and therefore require a *Forward*-operator.

definitions

A dictionary of definitions. Other than *rules* this maps the symbols to their compiled definienda.

deferred_taks

A list of callables that is filled during compilation, but that will be executed only after compilation has finished. Typically, it contains sementatic checks that require information that is only available upon completion of compilation.

root

The name of the root symbol.

directives

A dictionary of all directives and their default values.

re_flags

A set of regular expression flags to be added to all regular expressions found in the current parsing process

assemble_parser (*definitions: List[Tuple[str, str]], root_node: DHParse.syntaxtree.Node*) → str

Creates the Python code for the parser after compilation of the EBNF-Grammar

gen_compiler_skeleton () → str

Returns Python-skeleton-code for a Compiler-class for the previously compiled formal language.

gen_preprocessor_skeleton () → str

Returns Python-skeleton-code for a preprocessor-function for the previously compiled formal language.

gen_transformer_skeleton () → str

Returns Python-skeleton-code for the AST-transformation for the previously compiled formal language.

non_terminal (*node: DHParse.syntaxtree.Node, parser_class: str, custom_args: List[str] = []*) → str

Compiles any non-terminal, where *parser_class* indicates the Parser class name for the particular non-terminal.

grammar_changed (*grammar_class, grammar_source: str*) → bool

Returns True if *grammar_class* does not reflect the latest changes of *grammar_source*

Parameters

- **grammar_class** – the parser class representing the grammar or the file name of a compiler suite containing the grammar
- **grammar_source** – File name or string representation of the EBNF code of the grammar

Returns (bool): True, if the source text of the grammar is different from the source from which the grammar class was generated

PreprocessorFactoryFunc

alias of `typing.Callable`

ParserFactoryFunc

alias of `typing.Callable`

TransformerFactoryFunc

alias of `typing.Callable`

CompilerFactoryFunc

alias of `typing.Callable`

4.2.2 Module `dsl`

Module `dsl` contains various functions to support the compilation of domain specific languages based on an EBNF-grammar.

exception GrammarError (*errors, grammar_src*)

Raised when (already) the grammar of a domain specific language (DSL) contains errors.

exception CompilationError (*errors, dsl_text, dsl_grammar, AST, result*)

Raised when a string or file in a domain specific language (DSL) contains errors.

load_compiler_suite (*compiler_suite: str*) → Tuple[[Callable[Union[typing.Callable[[str], typing.Union[str, typing.Tuple[str, typing.Union[typing.Callable[[int], int], functools.partial]]], functools.partial]], Callable[DHParse.parse.Grammar], Callable[Union[typing.Callable[[DHParse.syntaxtree.Node], typing.Any], functools.partial]], Callable[DHParse.compile.Compiler]]]

Extracts a compiler suite from file or string *compiler_suite* and returns it as a tuple (preprocessor, parser, ast, compiler).

Returns 4-tuple (preprocessor function, parser class, ast transformer function, compiler class)

compiledSL (*text_or_file*: *str*, *preprocessor*: *Union[typing.Callable[[str], typing.Union[str, typing.Tuple[str, typing.Union[typing.Callable[[int], int], functools.partial]]], functools.partial]*, *dsl_grammar*: *Union[str, DHParse.parse.Grammar]*, *ast_transformation*: *Union[typing.Callable[[DHParse.syntaxtree.Node], typing.Any], functools.partial]*, *compiler*: *DHParse.compile.Compiler*) → *Any*

Compiles a text in a domain specific language (DSL) with an EBNF-specified grammar. Returns the compiled text or raises a compilation error.

Raises `CompilationError` if any errors occurred during compilation

raw_compileEBNF (*ebnf_src*: *str*, *branding*='DSL') → *DHParse.ebnf.EBNFCompiler*

Compiles an EBNF grammar file and returns the compiler object that was used and which can now be queried for the result as well as skeleton code for preprocessor, transformer and compiler objects.

Parameters

- **ebnf_src** (*str*) – Either the file name of an EBNF grammar or the EBNF grammar itself as a string.
- **branding** (*str*) – Branding name for the compiler suite source code.

Returns An instance of class `ebnf.EBNFCompiler`

Raises `CompilationError` if any errors occurred during compilation

compileEBNF (*ebnf_src*: *str*, *branding*='DSL') → *str*

Compiles an EBNF source file and returns the source code of a compiler suite with skeletons for preprocessor, transformer and compiler.

Parameters

- **ebnf_src** (*str*) – Either the file name of an EBNF grammar or the EBNF grammar itself as a string.
- **branding** (*str*) – Branding name for the compiler suite source code.

Returns The complete compiler suite skeleton as Python source code.

Raises `CompilationError` if any errors occurred during compilation

grammar_provider (*ebnf_src*: *str*, *branding*='DSL') → *DHParse.parse.Grammar*

Compiles an EBNF grammar and returns a grammar-parser provider function for that grammar.

Parameters

- **ebnf_src** (*str*) – Either the file name of an EBNF grammar or the EBNF grammar itself as a string.
- **branding** (*str* or *bool*) – Branding name for the compiler suite source code.

Returns A provider function for a grammar object for texts in the language defined by *ebnf_src*.

compile_on_disk (*source_file*: *str*, *compiler_suite*=", *extension*='.xml') → *Iterable[DHParse.error.Error]*

Compiles the a source file with a given compiler and writes the result to a file.

If no *compiler_suite* is given it is assumed that the source file is an EBNF grammar. In this case the result will be a Python script containing a parser for that grammar as well as the skeletons for a preprocessor, AST transformation table, and compiler. If the Python script already exists only the parser name in the script will be

updated. (For this to work, the different names need to be delimited section marker blocks.). `compile_on_disk()` returns a list of error messages or an empty list if no errors occurred.

Parameters

- **source_file** (*str*) – The file name of the source text to be compiled.
- **compiler_suite** (*str*) – The file name of the compiler suite (usually ending with ‘Compiler.py’), with which the source file shall be compiled. If this is left empty, the source file is assumed to be an EBNF-Grammar that will be compiled with the internal EBNF-Compiler.
- **extension** (*str*) – The result of the compilation (if successful) is written to a file with the same name but a different extension than the source file. This parameter sets the extension.

Returns A (potentially empty) list of error or warning messages.

recompile_grammar (*ebnf_filename*, *force=False*) → bool

Re-compiles an EBNF-grammar if necessary, that is, if either no corresponding ‘XXXXCompiler.py’-file exists or if that file is outdated.

Parameters

- **ebnf_filename** (*str*) – The filename of the ebnf-source of the grammar. In case this is a directory and not a file, all files within this directory ending with .ebnf will be compiled.
- **force** (*bool*) – If False (default), the grammar will only be recompiled if it has been changed.

4.2.3 Module testing

Module `testing` contains support for unit-testing domain specific languages. Tests for arbitrarily small components of the Grammar can be written into test files with ini-file syntax in order to test whether the parser matches or fails as expected. It can also be tested whether it produces an expected concrete or abstract syntax tree. Usually, however, unexpected failure to match a certain string is the main cause of trouble when constructing a context free Grammar.

unit_from_configfile (*config_filename*)

Reads grammar unit tests contained in a file in config file (.ini) syntax.

Parameters **config_filename** (*str*) – A config file containing Grammar unit-tests

Returns A dictionary representing the unit tests.

unit_from_json (*json_filename*)

Reads grammar unit tests from a json file.

unit_from_file (*filename*)

Reads a grammar unit test from a file. The format of the file is determined by the ending of its name.

get_report (*test_unit*)

Returns a text-report of the results of a grammar unit test. The report lists the source of all tests as well as the error messages, if a test failed or the abstract-syntax-tree (AST) in case of success.

If an asterisk has been appended to the test name then the concrete syntax tree will also be added to the report in this particular case.

The purpose of the latter is to help constructing and debugging of AST-Transformations. It is better to switch the CST-output on and off with the asterisk marker when needed than to output the CST for all tests which would unnecessarily bloat the test reports.

grammar_unit (*test_unit*, *parser_factory*, *transformer_factory*, *report=True*, *verbose=False*)

Unit tests for a grammar-parser and ast transformations.

grammar_suite (*directory*, *parser_factory*, *transformer_factory*, *fn_patterns*=[*'*test*'*], *ignore_unknown_filetypes*=*False*, *report*=*True*, *verbose*=*True*)
 Runs all grammar unit tests in a directory. A file is considered a test unit, if it has the word “test” in its name.

reset_unit (*test_unit*)
 Resets the tests in *test_unit* by removing all results and error messages.

runner (*test_classes*, *namespace*)
 Runs all or some selected Python unit tests found in the namespace. To run all tests in a module, call `runner("", globals())` from within that module.

Unit-Tests are either classes, the name of which starts with “Test” and methods, the name of which starts with “test” contained in such classes or functions, the name of which starts with “test”.

Parameters

- **tests** – Either a string or a list of strings that contains the names of test or test classes. Each test and, in the case of a test class, all tests within the test class will be run.
- **namespace** – The namespace for running the test, usually `globals()` should be used.

Example

```
class TestSomething()
    def setup(self): pass
    def teardown(self): pass
    def test_something(self): pass

if __name__ == "__main__": from DHParse.testing import runner runner("", globals())
```

4.3 Supporting Modules Reference

Finally, DHParse comprises a number of “toolkit”-modules which define helpful functions and classes that will be used at different places throughout the other DHParse-modules.

4.3.1 Module `toolkit`

Module `toolkit` contains utility functions that are needed across several of the other DHParse-Modules or that are just very generic so that they are best defined in a toolkit-module.

escape_re (*strg*: *str*) → *str*
 Returns the string with all regular expression special characters escaped.

escape_control_characters (*strg*: *str*) → *str*
 Replace all control characters (e.g.) in a string by their backslashed representation.

is_filename (*strg*: *str*) → *bool*
 Tries to guess whether string *s* is a file name.

lstrip_docstring (*docstring*: *str*) → *str*
 Strips leading whitespace from a docstring.

load_if_file (*text_or_file*) → *str*
 Reads and returns content of a text-file if parameter *text_or_file* is a file name (i.e. a single line string), otherwise (i.e. if *text_or_file* is a multi-line string) *text_or_file* is returned.

is_python_code (*text_or_file: str*) → bool

Checks whether 'text_or_file' is python code or the name of a file that contains python code.

md5 (**txt*)

Returns the md5-checksum for *txt*. This can be used to test if some piece of text, for example a grammar source file, has changed.

expand_table (*compact_table: Dict*) → Dict

Expands a table by separating keywords that are tuples or strings containing comma separated words into single keyword entries with the same values. Returns the expanded table. Example: >>> expand_table({'a, b': 1, ('d','e','f'):5, "c":3}) {'a': 1, 'b': 1, 'd': 5, 'e': 5, 'f': 5, 'c': 3}

compile_python_object (*python_src, catch_obj_regex=""*)

Compiles the python source code and returns the (first) object the name of which is matched by *catch_obj_regex*. If *catch_obj* is the empty string, the namespace dictionary will be returned.

smart_list (*arg: Union[typing.Iterable, typing.Any]*) → Union[typing.Sequence, typing.Set]

Returns the argument as list, depending on its type and content.

If the argument is a string, it will be interpreted as a list of comma separated values, trying ';', ',', ' ' as possible delimiters in this order, e.g. >>> smart_list('1; 2, 3; 4') ['1', '2, 3', '4'] >>> smart_list('2, 3') ['2', '3'] >>> smart_list('a b cd') ['a', 'b', 'cd']

If the argument is a collection other than a string, it will be returned as is, e.g. >>> smart_list((1, 2, 3)) (1, 2, 3) >>> smart_list({1, 2, 3}) {1, 2, 3}

If the argument is another iterable than a collection, it will be converted into a list, e.g. >>> smart_list(i for i in {1,2,3}) [1, 2, 3]

Finally, if none of the above is true, the argument will be wrapped in a list and returned, e.g. >>> smart_list(125) [125]

sane_parser_name (*name*) → bool

Checks whether given name is an acceptable parser name. Parser names must not be preceded or succeeded by a double underscore '__'!

4.3.2 Module log

Module `log` contains logging and debugging support for the parsing process.

For logging functionality, the global variable `LOGGING` is defined which contains the name of a directory where log files shall be placed. By setting its value to the empty string `""` logging can be turned off.

To read the directory name function `LOGS_DIR()` should be called rather than reading the variable `LOGGING`. `LOGS_DIR()` makes sure the directory exists and raises an error if a file with the same name already exists.

For debugging of the parsing process, the parsing history can be logged and written to an html-File.

For ease of use module `log` defines a context-manager `logging` to which either `False` (turn off logging), a log directory name or `True` for the default logging directory is passed as argument. The other components of `DHParse` check whether logging is on and write log files in the the logging directory accordingly. Usually, this will be concrete and abstract syntax trees as well as the full and abbreviated parsing history.

Example:

```
from DHParse import compile_source, logging

with logging("LOGS"):
    result, errors, ast = compile_source(source, preprocessor, grammar,
                                       transformer, compiler)
```

log_dir() → str

Creates a directory for log files (if it does not exist) and returns its path.

WARNING: Any files in the log dir will eventually be overwritten. Don't use a directory name that could be the name of a directory for other purposes than logging.

Returns name of the logging directory

logging (dirname='LOGS')

Context manager. Log files within this context will be stored in directory `dirname`. Logging is turned off if name is empty.

Parameters `dirname` – the name for the log directory or the empty string to turn logging of

is_logging() → bool

→ True, if logging is turned on.

logfile_basename (filename_or_text, function_or_class_or_instance) → str

Generates a reasonable logfile-name (without extension) based on the given information.

clear_logs (logfile_types=frozenset({'log', 'ast', 'cst'}))

Removes all logs from the log-directory and removes the log-directory if it is empty.

class HistoryRecord (call_stack: List[_ForwardRef('Parser')], node: DHParse.syntaxtree.Node, text: DHParse.stringview.StringView) → None

Stores debugging information about one completed step in the parsing history.

A parsing step is “completed” when the last one of a nested sequence of parser-calls returns. The call stack including the last parser call will be frozen in the `HistoryRecord`- object. In addition a reference to the generated leaf node (if any) will be stored and the result status of the last parser call, which is either MATCH, FAIL (i.e. no match) or ERROR.

class Snapshot (line, column, stack, status, text)

column

Alias for field number 1

line

Alias for field number 0

stack

Alias for field number 2

status

Alias for field number 3

text

Alias for field number 4

as_csv_line() → str

Returns history record formatted as a csv table row.

as_html_tr() → str

Returns history record formatted as an html table row.

as_tuple() → log.Snapshot

Returns history record formatted as a snapshot tuple.

static last_match() → Union[_ForwardRef('HistoryRecord'), NoneType]

Returns the last match from the parsing-history. :param history: the parsing-history as a list of `HistoryRecord` objects

Returns the history record of the last match or none if either history is empty or no parser could match

static most_advanced_match () → Union[_ForwardRef('HistoryRecord'), NoneType]

Returns the closest-to-the-end-match from the parsing-history. :param history: the parsing-history as a list of HistoryRecord objects

Returns the history record of the closest-to-the-end-match or none if either history is empty or no parser could match

log_ST (syntax_tree, log_file_name)

Writes an S-expression-representation of the *syntax_tree* to a file, if logging is turned on.

log_parsing_history (grammar, log_file_name: str = "", html: bool = False) → None

Writes a log of the parsing history of the most recently parsed document.

Parameters

- **grammar** (Grammar) – The Grammar object from which the parsing history shall be logged.
- **log_file_name** (str) – The (base-)name of the log file to be written. If no name is given (default), then the class name of the grammar object will be used.
- **html** (bool) – If true (default), the log will be output as html-Table, otherwise as plain test. (Browsers might take a few seconds or minutes to display the table for long histories.)

4.3.3 Module `stringview`

stringview.py - a string class where slices are views not copies as

with the standard Python strings.

stringview.pxd - declarations for the cython Python to C compiler to speed up handling of StringViews.

Copyright 2016 by Eckhart Arnold (arnold@badw.de) Bavarian Academy of Sciences an Humanities (badw.de)

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

StringView provides string-slicing without copying. Slicing Python-strings always yields copies of a segment of the original string. See: <https://mail.python.org/pipermail/python-dev/2008-May/079699.html> However, this becomes costly (in terms of space and as a consequence also time) when parsing longer documents. Unfortunately, Python’s *memoryview* does not work for unicode strings. Hence, the StringView class.

class StringView

A rudimentary StringView class, just enough for the use cases in parse.py. The difference between a StringView and the python builtin strings is that StringView-objects do slicing without copying, i.e. slices are just a view on a section of the sliced string.

count ()

Returns the number of non-overlapping occurrences of substring *sub* in StringView *S*[start:end]. Optional arguments start and end are interpreted as in slice notation.

find()

Returns the lowest index in *S* where substring *sub* is found, such that *sub* is contained within *S*[start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns -1 on failure.

index()

Converts an index for a string watched by a *StringView* object to an index relative to the string view object, e.g.: >>> import re >>> sv = *StringView*('xxIxx')[2:3] >>> match = sv.match(re.compile('I')) >>> match.end() 3 >>> sv.index(match.end()) 1

indices()

Converts indices for a string watched by a *StringView* object to indices relative to the string view object. See also: *sv_index()*

match()

Executes *regex.match* on the *StringView* object and returns the result, which is either a match-object or *None*.

replace()

Returns a string where *old* is replaced by *new*.

rfind()

Returns the highest index in *S* where substring *sub* is found, such that *sub* is contained within *S*[start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns -1 on failure.

search()

Executes *regex.search* on the *StringView* object and returns the result, which is either a match-object or *None*.

split()

Returns a list of the words in *self*, using *sep* as the delimiter string. If *sep* is not specified or is *None*, any whitespace string is a separator and empty strings are removed from the result.

startswith()

Return *True* if *S* starts with the specified prefix, *False* otherwise. With optional *start*, test *S* beginning at that position. With optional *end*, stop comparing *S* at that position. prefix can also be a tuple of strings to try.

strip()

Returns a copy of the *StringView self* with leading and trailing whitespace removed.

4.3.4 Module `versionnumber`

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

c

compile, [25](#)

d

dsl, [30](#)

e

ebnf, [28](#)

error, [26](#)

l

log, [34](#)

p

parse, [12](#)

preprocess, [8](#)

s

stringview, [36](#)

syntaxtree, [9](#)

t

testing, [32](#)

toolkit, [33](#)

transform, [21](#)

v

versionnumber, [37](#)

Symbols

`_dirty_flag` (Compiler attribute), 25
`_dirty_flag__` (Grammar attribute), 15
`_parent` (Node attribute), 10
`_reversed__` (Grammar attribute), 15

A

`add_error()` (Node method), 10
`adjust_error_locations()` (in module `error`), 27
`all_parsers__` (Grammar attribute), 15
`AllOf` (class in `parse`), 19
`Alternative` (class in `parse`), 19
`apply()` (Forward method), 21
`apply()` (Parser method), 13
`apply()` (ParserBase method), 9
`apply()` (RE method), 17
`apply_if()` (in module `transform`), 23
`apply_unless()` (in module `transform`), 23
`ApplyFunc` (Parser attribute), 13
`as_csv_line()` (HistoryRecord method), 35
`as_html_tr()` (HistoryRecord method), 35
`as_sxpr()` (Node method), 10
`as_tuple()` (HistoryRecord method), 35
`as_xml()` (Node method), 11
`assemble_parser()` (EBNFCompiler method), 30
`assert_condition()` (in module `transform`), 24
`attributes` (Node attribute), 10, 11

C

`call_stack__` (Grammar attribute), 15
`Capture` (class in `parse`), 20
`chain_preprocessors()` (in module `preprocess`), 8
`children` (Node attribute), 10
`clear_logs()` (in module `log`), 35
`collapse()` (in module `transform`), 23
`collect_errors()` (Node method), 11
`column` (HistoryRecord.Snapshot attribute), 35
`combined_mandatory()` (Series static method), 19
`COMMENT__` (Grammar attribute), 14
`CompilationError`, 30
`compile` (module), 25
`compile()` (Compiler method), 25

`compile_on_disk()` (in module `dsl`), 31
`compile_python_object()` (in module `toolkit`), 34
`compile_source()` (in module `compile`), 26
`compileDSL()` (in module `dsl`), 31
`compileEBNF()` (in module `dsl`), 31
`Compiler` (class in `compile`), 25
`CompilerError`, 25
`CompilerFactoryFunc` (in module `ebnf`), 30
`ConditionFunc` (in module `transform`), 21
`content` (Node attribute), 10, 11
`context` (Compiler attribute), 25
`count()` (StringView method), 36
`create_main_parser()` (RE method), 17
`create_main_parser()` (Token method), 17
`current_symbols` (EBNFCompiler attribute), 29

D

`deferred_taks` (EBNFCompiler attribute), 29
`definitions` (EBNFCompiler attribute), 29
`directives` (EBNFCompiler attribute), 29
`document__` (Grammar attribute), 15
`document_lbreaks__` (Grammar attribute), 15
`document_length__` (Grammar attribute), 15
`dsl` (module), 30

E

`ebnf` (module), 28
`EBNFCompiler` (class in `ebnf`), 29
`EBNFCompilerError`, 29
`EBNFGrammar` (class in `ebnf`), 28
`error` (module), 26
`error_flag` (Node attribute), 10
`errors` (Node attribute), 10, 11
`escape_control_characters()` (in module `toolkit`), 33
`escape_re()` (in module `toolkit`), 33
`expand_table()` (in module `toolkit`), 34

F

`fallback_compiler()` (Compiler method), 25
`find()` (StringView method), 36
`flatten()` (in module `transform`), 24
`flatten_sxpr()` (in module `syntaxtree`), 12

Forward (class in parse), 21

G

gen_compiler_skeleton() (EBNFCompiler method), 30
gen_preprocessor_skeleton() (EBNFCompiler method), 30
gen_transformer_skeleton() (EBNFCompiler method), 30
get_report() (in module testing), 32
Grammar (class in parse), 13
grammar (Parser attribute), 13
grammar() (ParserBase method), 9
grammar_changed() (in module ebnf), 30
grammar_name (Compiler attribute), 25
grammar_provider() (in module dsl), 31
grammar_source (Compiler attribute), 25
grammar_suite() (in module testing), 32
grammar_unit() (in module testing), 32
GrammarError, 30

H

has_content() (in module transform), 24
has_errors() (in module error), 27
has_parent() (in module transform), 24
history__ (Grammar attribute), 16
history_tracking__ (Grammar attribute), 15
HistoryRecord (class in log), 35
HistoryRecord.Snapshot (class in log), 35

I

index() (StringView method), 37
indices() (StringView method), 37
init_pos() (Node method), 11
is_anonymous() (in module transform), 23
is_empty() (in module transform), 23
is_error() (in module error), 27
is_expendable() (in module transform), 23
is_filename() (in module toolkit), 33
is_logging() (in module log), 35
is_named() (in module transform), 22
is_one_of() (in module transform), 23
is_python_code() (in module toolkit), 34
is_token() (in module transform), 23
is_warning() (in module error), 27
is_whitespace() (in module transform), 23

K

keep_children() (in module transform), 24
keep_children_if() (in module transform), 24
keep_content() (in module transform), 24
keep_nodes() (in module transform), 24
keep_tokens() (in module transform), 24
KeyFunc (in module transform), 21

L

last_match() (HistoryRecord static method), 35
last_rb__loc__ (Grammar attribute), 15
left_recursion_handling__ (Grammar attribute), 16
len (Node attribute), 10
line (HistoryRecord.Snapshot attribute), 35
line_col() (in module error), 27
linebreaks() (in module error), 27
load_compiler_suite() (in module dsl), 30
load_if_file() (in module toolkit), 33
log (module), 34
log_dir() (in module log), 34
log_parsing_history() (in module log), 36
log_ST() (in module log), 36
logfile_basename() (in module log), 35
logging() (in module log), 35
Lookahead (class in parse), 20
Lookbehind (class in parse), 20
lstrip() (in module transform), 24
lstrip_docstring() (in module toolkit), 33

M

make_token() (in module preprocess), 8
match() (StringView method), 37
md5() (in module toolkit), 34
memoization__ (Grammar attribute), 16
merge_children() (in module transform), 23
method_name() (Compiler static method), 25
mixin_comment() (in module parse), 17
mock_syntax_tree() (in module syntaxtree), 12
MockParser (class in syntaxtree), 9
most_advanced_match() (HistoryRecord static method), 36
moving_forward__ (Grammar attribute), 16

N

name (ParserBase attribute), 9
NegativeLookahead (class in parse), 20
NegativeLookbehind (class in parse), 20
nil_preprocessor() (in module preprocess), 8
Node (class in syntaxtree), 9
non_terminal() (EBNFCompiler method), 30

O

offsets (SourceMap attribute), 8
OneOrMore (class in parse), 18
only_errors() (in module error), 27
Option (class in parse), 17

P

parse (module), 12
Parser (class in parse), 12
parser (Node attribute), 10

parser_initialization__ (Grammar attribute), 15
 ParserBase (class in syntaxtree), 9
 ParserFactoryFunc (in module ebnf), 30
 pick() (Node method), 11
 Pop (class in parse), 20
 pos (Node attribute), 10, 11
 positions (SourceMap attribute), 8
 preprocess (module), 8
 PreprocessorFactoryFunc (in module ebnf), 30
 PreprocessorToken (class in parse), 16
 prettyprint_tokenized() (in module preprocess), 8
 propagate_error_flags() (Compiler static method), 26
 ptype (ParserBase attribute), 9
 push_rollback__() (Grammar method), 16

R

raw_compileEBNF() (in module dsl), 31
 RE (class in parse), 16
 re_flags (EBNFCompiler attribute), 29
 recompile_grammar() (in module dsl), 32
 recursion_locations__ (Grammar attribute), 16
 recursive (EBNFCompiler attribute), 29
 reduce_single_child() (in module transform), 23
 RegExp (class in parse), 16
 remove_children_if() (in module transform), 24
 remove_content() (in module transform), 24
 remove_nodes() (in module transform), 24
 remove_tokens() (in module transform), 24
 replace() (StringView method), 37
 replace_by_single_child() (in module transform), 23
 replace_content() (in module transform), 23
 replace_or_reduce() (in module transform), 23
 replace_parser() (in module transform), 23
 repr (ParserBase attribute), 9
 reset() (Alternative method), 19
 reset() (Parser method), 13
 reset() (ParserBase method), 9
 reset_unit() (in module testing), 33
 result (Node attribute), 10, 11
 Retrieve (class in parse), 20
 retrieve_and_match() (Retrieve method), 20
 reversed__ (Grammar attribute), 16
 rfind() (StringView method), 37
 rollback__ (Grammar attribute), 15
 rollback_to__() (Grammar method), 16
 root (EBNFCompiler attribute), 29
 root__ (Grammar attribute), 15
 rstrip() (in module transform), 24
 rules (EBNFCompiler attribute), 29
 runner() (in module testing), 33

S

sane_parser_name() (in module toolkit), 34
 search() (StringView method), 37

select() (Node method), 11
 select_by_tag() (Node method), 11
 Series (class in parse), 18
 set() (Forward method), 21
 set_grammar_name() (Compiler method), 26
 sign() (NegativeLookahead method), 20
 sign() (NegativeLookbehind method), 20
 smart_list() (in module toolkit), 34
 SomeOf (class in parse), 19
 source_map() (in module preprocess), 8
 SourceMap (class in preprocess), 8
 split() (StringView method), 37
 stack (HistoryRecord.Snapshot attribute), 35
 startswith() (StringView method), 37
 status (HistoryRecord.Snapshot attribute), 35
 StringView (class in stringview), 36
 stringview (module), 36
 strip() (in module transform), 24
 strip() (StringView method), 37
 strip_tokens() (in module preprocess), 8
 structure (Node attribute), 12
 symbols (EBNFCompiler attribute), 29
 Synonym (class in parse), 17
 syntaxtree (module), 9

T

tag_name (Node attribute), 9, 12
 testing (module), 32
 text (HistoryRecord.Snapshot attribute), 35
 Token (class in parse), 17
 tokenized_to_original_mapping() (in module preprocess), 8
 toolkit (module), 33
 transform (module), 21
 transformation_factory() (in module transform), 21
 TransformationDict (in module transform), 21
 TransformationProc (in module transform), 21
 TransformerFactoryFunc (in module ebnf), 30
 traverse() (in module transform), 22
 traverse_locally() (in module transform), 23
 tree_size() (Node method), 12

U

unit_from_configfile() (in module testing), 32
 unit_from_file() (in module testing), 32
 unit_from_json() (in module testing), 32
 UnknownParserError, 13
 Unordered() (in module parse), 20

V

variables (EBNFCompiler attribute), 29
 variables__ (Grammar attribute), 15
 versionnumber (module), 37

W

`with_source_mapping()` (in module `preprocess`), 9

`WSP__` (Grammar attribute), 14

`wsp_left_parser__` (Grammar attribute), 15

`wsp_right_parser__` (Grammar attribute), 15

`wspL__` (Grammar attribute), 14

`wspR__` (Grammar attribute), 15

Z

`ZeroOrMore` (class in `parse`), 18

`ZombieParser` (class in `syntaxtree`), 9