DHParser Documentation

Release 0.8

Eckhart Arnold

May 01, 2018

CONTENTS:

CHAPTER

ONE

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. Similarily 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 reStructedText-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 verify that the installation works, you can simply run the "dhparser.py" script and, when asked, chose "3" for the self-test. If the self-test runs through without error, the installation has succeded.

1.1.2 Staring a new DHParser project

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
$ cd experimental/poetry
```

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

```
README.md             - a stub for a readme-file explaiing the project
poetry.ebnf           - a trivial demo grammar for the new project
example.dsl           - an example file written in this grammar
tst_poetry_grammar.py - a python script ("test-script") that re-compiles
                        the grammar (if necessary) and runs the unit tests
grammar_tests/01_test_word.ini     - a demo unit test
grammar_tests/02_test_document.ini - another unit test
```

Now, if you look into the file "example.dsl" you will find that it contains a simple sequence of words, namely "Life is but a walking shadow". In fact, the demo grammar that comes with a newly created project is nothing but simple grammar for sequences of words separated by whitespace. Now, since we alread have unit tests, our first exercise will be to run the unit tests by starting the script "tst_poetry_grammar.py":

```
$ python tst_poetry_grammar.py
```

This will run through the unit-tests in the grammar_tests directory and print their success or failure on the screen. If you check the contents of your project directory after running the script, you might notice that there now exists a new file "poetryCompiler.py" in the project directory. This is an auto-generated compiler-script for our DSL. You can use this script to compile any source file of your DSL, like "example.dsl". Let's try:

```
$ python poetryCompiler.py example.dsl
```

The output is a block of pseudo-XML, looking like this:

```
<document>
  <:ZeroOrMore>
    <WORD>
      <:RegExp>Life</:RegExp>
      <:Whitespace> </:Whitespace>
    </WORD>
    <WORD>
      <:RegExp>is</:RegExp>
      <:Whitespace> </:Whitespace>
```

(continues on next page)

(continued from previous page)

```
    </WORD>
...
```

Now, this does not look too helpful yet, partly, because it is cluttered with all sorts of seemingly superflous pseudo-XML-tags like "<:ZeroOrMore>". However, you might notice that it contains the original sequence of words "Life is but a walkting shadow" in a structured form, where each word is (among other things) surrounded by <WORD>-tags. In fact, the output of the compiler script is a pseudo-XML-representation of the *contrete syntax tree* of our "example.dsl"-document according the grammar specified in "poetry.ebnf" (which we haven't looked into yet, but we will do so soon).

If you see the pseudo-XML on screen, the setup of the new DHParser-project has been successful.

1.1.3 Understanding how compilation of DSL-documents with DHParser works

Generally speaking, the compilation process consists of three stages:

1. Parsing a document. This yields a *concrete syntax tree* (CST) of the document.
2. Transforming. This transforms the CST into the much more concise *abstract syntax tree* (AST) of the document.
3. Compiling. This turns the AST into anything you'd like, for example, an XML-representation or a relational database record.

Now, DHParser can fully automize the generation of a parser from a syntax-description in EBNF-form, like our "poetry.ebnf", but it cannot automize the transformation from the concrete into the abstract syntax tree (which for the sake of brevity we will simply call "AST-Transformation" in the following), and neither can it automize the compilation of the abstract syntax tree into something more useful. Therefore, the AST-Transformation in the autogenerated compile-script is simply left empty, while the compiling stage simply converts the syntax tree into a pseudo-XML-format.

The latter two stages have to be coded into the compile-script by hand, with the support of templates within this script. If the grammar of the DSL is changed - as it will be frequently during the development of a DSL - the parser-part of this script will be regenerated by the testing-script before the unit tests are run. The script will notice if the grammar has changed. This also means that the parser part of this script will be overwritten and should never be edited by hand. The other two stages can and should be edited by hand. Stubs for theses parts of the compile-script will only be generated if the compile-script does not yet exist, that is, on the very first calling of the test-srcipt.

Usually, if you have adjusted the grammar, you will want to run the unit tests anyway. Therefore, the regeneration of the parser-part of the compile-script is triggered by the test-script.

1.1.4 The development workflow for DSLs

When developing a domain specific notation it is recommendable to first develop the grammar and the parser for that notation, then to the abstract syntax tree transformations and finally to implement the compiler. Of course one can always come back and change the grammar later. But in order to avoid revising the AST-transformations and the compiler time and again it helps if the grammar has been worked out before. A bit of interlocking between these steps does not hurt, though.

A resonable workflow for developing the grammar proceeds like this:

1. Set out by writing down a few example documents for your DSL. It is advisable to start with a few simple examples that use only a subset of the intended features of your DSL.
2. Next you sktech a grammar for your DSL that is just rich enough to capture those examples.

3. Right after sketching the grammar you should write test cases for your grammar. The test cases can be small parts or snippets of your example documents. You could also use your example documents as test cases, but usually the test cases should have a smaller granularity to make locating errors easier.
4. Next, you should run the test script. Usually, some test will fail at the first attempt. So you'll keep revising the EBNF-grammar, adjusting and adding test cases until all tests pass.
5. Now it is time to try and compile the example documents. By this time the test-script should have generated the compile-script, which you can be called with the example documents. Don't worry too much about the output, yet. What is important at this stage is merely whether the parser can handle the examples or not. If not, further test cases and adjustments the EBNF grammar will be needed - or revision of the examples in case you decide to use different syntactic constructs.

   If all examples can be parsed, you go back to step one and add further more complex examples, and continue to do so until you have the feeling that you DSL's grammar is rich enough for all intended application cases.

Let's try this with the trivial demo example that comes with creating a new project with the "dhparser.py"-script. Now, you have already seen that the "example.dsl"-document merely contains a simple sequence of words: "Life is but a walking shadow" Now, wouldn't it be nice, if we could end this sequence with a full stop to turn it into a proper sentence. So, open "examples.dsl" with a text editor and add a full stop:

```
Life is but a walking shadow.
```

Now, try to compile "examples.dsl" with the compile-script:

```
$ python poetryCompiler.py example.dsl
example.dsl:1:29: Error: EOF expected; ".\n " found!
```

Since the grammar, obviously, did not allow full stops so far, the parser returns an error message. The error message is pretty self-explanatory in this case. (Often, you will unfortunately find that the error message are somewhat difficult to decipher. In particular, because it so happens that an error the parser complains about is just the consequence of an error made at an earlier location that the parser may not have been able to recognize as such. We will learn more about how to avoid such situations, later.) EOF is actually the name of a parser that captures the end of the file, thus "EOF"! But instead of the expected end of file an, as of now, unparsable construct, namely a full stop followed by a line feed, signified by "n", was found.

Let's have look into the grammar description "poetry.ebnf". We ignore the beginning of the file, in particular all lines starting with "@" as these lines do not represent any grammar rules, but meta rules or so-called "directives" that determine some general characteristics of the grammar, such as whitespace-handling or whether the parser is going to be case-sensitive. Now, there are exactly three rules that make up this grammar:

```
document = ~ { WORD } §EOF
WORD     =  /\w+/~
EOF      =  !/./
```

EBNF-Grammars describe the structure of a domain specific notation in top-down fashion. Thus, the first rule in the grammar describes the comonents out of which a text or document in the domain specific notation is composed as a whole. The following rules then break down the components into even smaller components until, finally, there a only atomic components left which are described be matching rules. Matching rules are rules that do not refer to other rules any more. They consist of string literals or regular expressions that "capture" the sequences of characters which form the atomic components of our DSL. Rules in general always consist of a symbol on the left hand side of a "="-sign (which in this context can be unterstood as a definition signifier) and the definition of the rule on the right hand side.

Note: Traditional parser technology for context-free grammars often distinguishes two phases, *scanning* and *parsing*, where a lexical scanner would take a stream of characters and yield a sequence of tokens and the actual parser would then operate on the stream of tokens. DHParser, however, is an instance of a *scannerless parser* where the functionality of the lexical scanner is seamlessly integrated into the parser. This is done by allowing regular expressions in the definiendum of grammar symbols. The regular expressions do the work of the lexical scanner.

Theoretically, one could do without scanners or regular expressions. Because regular languages are a subset of context-free languages, parsers for context-free languages can do all the work that regular expressions can do. But it makes things easier - and, in the case of DHParser, also faster - to have them.

In our case the text as a whole, conveniently named “document” (any other name would be allowed, too), consists of a leading whitespace, a possibly empty sequence of an arbitrary number of words words ending only if the end of file has been reached. Whitespace in DHParser-grammers is always denoted by a tilde “~”. Thuse the definiens of the rule “document” starts with a “~” on the right hand side of the deifnition sign (“=”). Next, you find the symbol “WORD” enclosed in braces. “WORD”, like any symbol composed of letters in DHParser, refers to another rule further below that defines what words are. The meaning of the braces is that whatever is enclosed by braces may be repeated zero or more times. Thus the expression “{ WORD }” describes a seuqence of arbitrarily many repetitions of WORD, whatever WORD may be. Finally, EOF refers to yet another rule definied further below. We do not yet know what EOF is, but we know that when the sequence of words ends, it must be followed by an EOF. The paragraph sign “§” in front of EOF means that it is absolutely mandatory that the seuqence of WORDs is followed by an EOF. If it doesn’t the program issues an error message. Without the “§”-sign the parser simply would not match, which in itself is not considered an error.

Now, let’s look at our two matching rules. Both of these rules contain regular expressions. If you do not know about regular expressions yet, you should head over to an explanation or tutorial on regular expressions, like https://docs.python.org/3/library/re.html, before continuing, because we are not going to discuss them here. In DHParser-Grammars regular expressions are enclosed by simple forawrd slashes “/”. Everything between two forward slashes is a regular expression as it would be understood by Python’s “re”-module. Thus the rule `WORD = /\w+/~` means that a word consists of a seuqence of letters, numbers or underscores ‘_’ that must be at least one sign long. This is what the regular expression “w+” inside the slashes means. In regular expressions, “w” stands for word-characters and “+” means that the previous character can be repeated one or more times. The tile “~” following the regular expression, we already know. It means that a a word can be followed by whitespace. Strictly speaking that whitespace is part of “WORD” as it is defined here.

Similarly, the EOF (for “end of line”) symbol is defined by a rule that consists of a simple regular expression, namely “.”. The dot in regular expressions means any character. However, the regular expression itself preceded by an exclamations mark “!”. IN DHParser-Grammars, the explanation mark means “not”. Therefore the whole rule means, that *no* character must follow. Since this is true only for the end of file, the parser looking for EOF will only match if the very end of the file has been reached.

Now, what would be the easiest way to allow our sequence of words to be ended like a real sentence with a dot “.”? As always when defining grammars on can think of different choice to implement this requirement in our grammar. One possible solution is to add a dot-literal before the “§EOF”-component at the end of the definition of the “document”-rule. So let’s do that. Change the line where the “document”-rule is defined to:

```
document = ~ { WORD } "." §EOF
```

As you can see, string-literals are simply denoted as strings between inverted commas in DHParser’s variant of the EBNF-Grammar. Now, before we can compile the file “example.dsl”, we will have to regenerate the our parser, because we have changed the grammar. In order to recompile, we simply run the test-script again:

```
$ python tst_poetry_grammar.py
```

But what is that? A whole lot of errormessages? Well, this it not surprising, because we change the grammar, some of our old test-cases fail with the new grammar. So we will have to update our test-cases wird. (Actually, the grammar get’s compiles never the less and we could just ignore the test failures and carry on with compiling our “example.dsl”-file again. But, for this time, we’ll follow good practice and adjust the test cases. So open the test that failed, “grammar_tests/02_test_document.ini”, in the editor and add full stops at the end of the “match”-cases and remove the full stop at the end of the “fail”-case:

```
[match:document]
M1: """This is a sequence of words
    extending over several lines."""
M2: """  This sequence contains leading whitespace."""

[fail:document]
F1: """This test should fail, because neither
    comma nor full have been defined anywhere"""
```

The format of the test-files should be pretty self-explanatory. It is a simple ini-file, where the section markers hold the name of the grammar-rule to be tested which is either preceded by "match" or "fail". "match means" that the following examples should be matched by the grammar-rule. "fail" means they should *not* match. It is just as important that a parser or grammar-rules does not match those strings it should not match as it is that it matches those strings that it should match. The individual test-cases all get a name, in this case M1, M2, F2, but if you prefer more meaningful names this is also possible. (Beware, however, that the names for match-test different from the names for the fail tests for the same rule!). Now, run the test-script again and you'll see that no errors get reported any more.

Finally, we can recompile out "example.dsl"-file, and by its XML output we can tell that it worked:

```
$ python poetryCompiler.py example.dsl
```

So far, we have seen *in nuce* how the development workflow for a building up DSL-grammar goes. Let's take this a step further by adding more capabilities to our grammr.

1.1.5 Extending the example DSL further

A grammar that can only digest single sentences is certainly a rather boring. So we'll extend our grammar a little further so that it can capture paragraphs of sentences. To see, where we are heading, let's first start a new example file, let's call it "macbeth.dsl" and enter the following lines:

```
Life's but a walking shadow, a poor player that struts and frets his hour
upon the stage and then is heard no more. It is a tale told by an idiot,
full of sound and fury, signifying nothing.
```

What have we got, there? We've got a paragraph that consists of several sentences each of which ends with a full stop. The sentences themselves can consist of different parts which a separated by a comma. If, so far, we have got a clear idea (in verbal terms) of the structure of texts in our DSL, we can now try to formulate this in the grammar.

> document = ~ { sentence } §EOF sentence = part {"," part } "." part = { WORD } # a subtle mistake, right here! WORD = /w+/~ # something forgotten, here! EOF = !/./

The most important new part is the grammar rule "sentence". It reads as this: A sentence is a part of a sentence potentially followed by a repeated sequence of a comma and another part of a sentence and ultimately ending with a full stop. (Understandable? If you have ever read Russell's "Introduction to Mathematical Philosophy" you will be used to this kind of prose. Other than that I find the formal definition easier to understand. However, for learning EBNF or any other formalism, it helps in the beginning to translate the meaning of its statements into plain old Englisch.)

There is are two subtle mistakes in this grammar. If you can figure them out just by thinking about it, feel free to correct the grammar right now. (Would you really have noticed the mistakes if they hadn't already been marked in the code above?) For all less intelligent people, like me: Let's be prudent and - since the grammar has become more complex - add a few test cases. This should make it easier to locate any errors. So open up an editor with a new file in the tests subdirectory, say `grammar_tests/03_test_sentence.ini` (Test files should always contain the component "**test_**" in the filename, otherwise they will be overlooked by DHParser's unit testing subsystem) and enter a few test-cases like these:

```
[match:sentence]
M1: """It is a tale told by an idiot,
   full of sound and fury, signifying nothing."""
M2: """Plain old sentence."""

[fail:sentence]
F1: """Ups, a full stop is missing"""
F2: """No commas at the end,."""
```

Again, we recompile the grammar and run the test at the same time by running the testing-script:

```
$ python tst_poetry_grammar.py
Errors found by unit test "03_test_sentence.ini":
Fail test "F2" for parser "sentence" yields match instead of expected failure!
```

Too bad, something went wrong here. But what? Didn't the definition of the rule "sentence" make sure that parts of sentences are, if at all, only be followed by a sequence of a comma *and* another part of a sentence. So, how come that between the last comma and the full stop there is nothing but empty space? Ah, there's the rub! If we look into our grammar, how parts of sentences have been defined, we find that the rule:

```
part = { WORD }
```

definies a part of a sentence as a sequence of *zero* or more WORDs. This means that a string of length zero also counts as a valid part of a sentence. Now in order to avoid this, we could write:

```
part = WORD { WORD }
```

This definition makes sure that there is at least on WORD in a part. Since the case that at least one item is needed occurs rather frequently in grammars, DHParser offers a special syntax for this case:

```
part = { WORD }+
```

(The plus sign "+" must always follow directly after the curly brace "}" without any whitespcae in between, otherwise DHParser won't understannd it.) At this point the worry may arise that the same problem could reoccur at another level, if the rule for WORD would match empty strings as well. Let's quickly add a test case for this to the file `grammar_tests/01_test_word.ini`:

```
[fail:WORD]
F1: two words
F2: ""
```

Thus, we are sure to be warned in case the definition of rule "WORD" matches the empty string. Luckily, it does not do so now. But it might happen that we change this definition later again for some reason, we might have forgotton about this subtlety and introduce the same error again. With a test case we can reduce the risk of such a regression error. This time the tests run through, nicely. So let's try the parser on our new example:

```
$ python poetryCompiler.py macbeth.dsl
macbeth.dsl:1:1: Error: EOF expected; "Life's but" found!
```

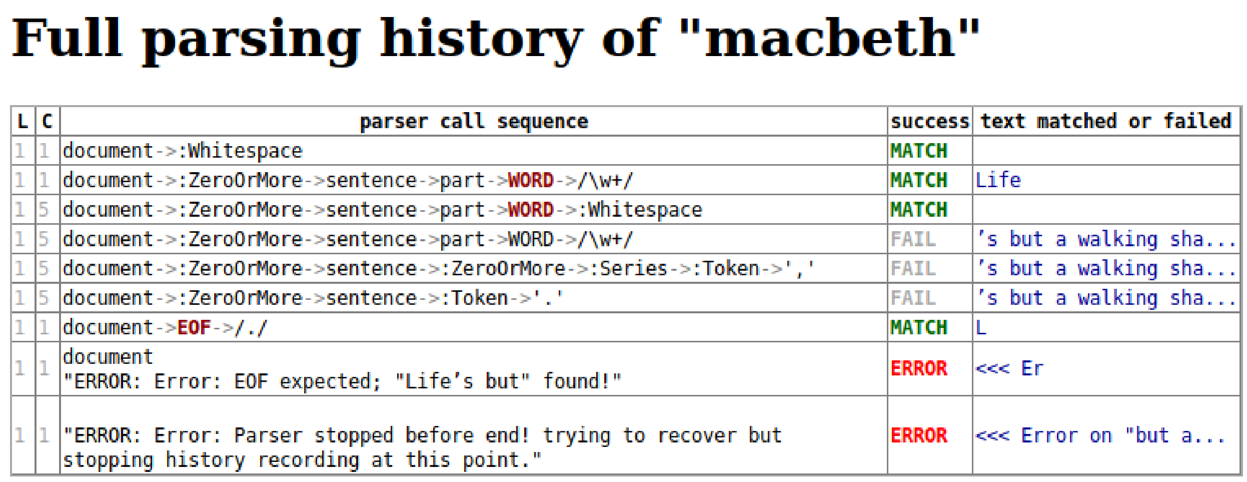
That is strange. Obviously, there is an error right at the beginning (line 1 column 1). But what coul possibly be wrong with the word "Life". Now you might already have guessed what the error is and that the error is not exactly located in the first column of the first line.

Unfortunately, DHParser - like almost any other parser out there - is not always very good at spotting the exact location of an error. Because rules refer to other rules, a rule may fail to parse - or, what is just as bad, succeed to parse when it should indeed fail - as a consequence of an error in the definition of one of the rules it refers to. But this means if the rule for the whole document fails to match, the actual error can be located anywhere in the document! There a different approaches to dealing with this problem. A tool that DHParser offers is to write log-files that document the parsing history. The log-files allow to spot the location, where the parsing error occured. However, you will have to look for the error manually. A good starting point is usually either the end of the parsing process or the point where the parser reached the farthest into the text. In order to receive the parsing history, you need to run the compiler-script again with the debugging option:

```
$ python poetryCompiler.py macbeth.dsl
```

You will receive the same error messages as before. but this time various kinds of debugging information have been written into a new created subdirectory “LOGS”. (Beware that any files in the “LOGS” directory may be overwritten or deleted by any of the DHParser scripts upon the next run! So don’t store any important data there.) The most interesting file in the “LGOS”-directory is the full parser log. We’ll ignore the other files and just open the file “macbech_full_parser.log.html” in an internet-browser. As the parsing history tends to become quite long, this usually takes a while, but luckily not in the case of our short demo example:

```
$ firefox LOGS/macbeth_full_parser.log.html &
```



Full parsing history of "macbeth"

L	C	parser call sequence	success	text matched or failed
1	1	document->:Whitespace	MATCH	
1	1	document->:ZeroOrMore->sentence->part->WORD->/\w+/	MATCH	Life
1	5	document->:ZeroOrMore->sentence->part->WORD->:Whitespace	MATCH	
1	5	document->:ZeroOrMore->sentence->part->WORD->/\w+/	FAIL	's but a walking sha...
1	5	document->:ZeroOrMore->sentence->:ZeroOrMore->:Series->:Token->','	FAIL	's but a walking sha...
1	5	document->:ZeroOrMore->sentence->:Token->'.'	FAIL	's but a walking sha...
1	1	document->EOF->/./	MATCH	L
1	1	document "ERROR: Error: EOF expected; "Life's but" found!"	ERROR	<<< Er
1	1	"ERROR: Error: Parser stopped before end! trying to recover but stopping history recording at this point."	ERROR	<<< Error on "but a...

What you see is a representation of the parsing history. It might look a bit tedious in the beginning, especially the this column that contains the parser call sequence. But it is all very straight forward: For every application of a match rule, there is a row in the table. Typically, match rules are applied at the end of a long sequence of parser calls that is displayed in the third column. You will recognise the parsers that represent rules by their names, e.g. “document”, “sentence” etc. Those parsers that merely represent constructs of the EBNF grammar within a rule do not have a name and are represented by theis type, which always begins with a colon, like “:ZeroOrMore”. Finally, the regular expression or literal parsers are represented by the regular expression pattern or the string literal themselves. (Arguably, it can be confusing that parsers are represented in three different ways in the parer call sequence. I am still figuring out a better way to display the parser call sequence. Any suggestions welcome!) The first two columns display the position in the text in terms of lines and columns. The second but last column, labeled “success” shows wether the last parser in the sequence matched or failed or produced an error. In case of an error, the error message is displayed in the third column as well. In case the parser matched, the last column displays exactly that section of the text that the parser did match. If the parser did not match, the last column displays the text that still lies ahead and has not yet been parsed.

In our concrete example, we can see that the parser “WORD” matches “Life”, but not “Life’s” or “’s”. And this ultimately leads to the failure of the parsing process as a whole. The simplemost solution would be to add the apostrophe to the list of allowed characters in a word by changeing the respective line in the grammar definition to `WORD = /[\w']+/`. Now, before we even change the grammar we first add another test case to capture this kind of error. Since we have decided that “Life’s” should be parsed as a singe word, let’s open the file “grammar_tests/01_test_word.ini” and add the following test:

```
[match:WORD]
M3: Life's
```

To be sure that the new test captures the error we have found you might want to run the script "tst_poetry_grammar.py" and verify that it reports the failure of test "M3" in the suite "01_test_word.ini". After that, change the regular expression for the symbol WORD in the grammar file "poetry.ebnf" as just described. Now both the tests and the compilation of the file "macbeth.dsl" should run through smoothly.

Caution: Depending on the purpose of your DSL, the simple solution of allowing apostrophes within words, might not be what you want. After all "Life's" is but a shorthand for the two word phrase "Life is". Now, whatever alternative solution now comes to your mind, be aware that there are also cases like Irish names, say "O'Dolan" where the apostrophe is actually a part of a word and cases like "don't" which, if expanded, would be two words *not* separated at the position of the apostrophe.

We leave that as an exercise, first to figure out, what different cases for the use of apostrophes in the middle of a word exist. Secondly, to make a reasonable decision which of these should be treated as a single and which as separate words and, finally, if possible, to write a grammar that provides for these cases. These steps are quite typical for the kind of challenges that occur during the design of a DSL for a Digital-Humanities-Project.

1.1.6 Controlling abstract-syntax-tree generation

Compiling the example "macbeth.dsl" with the command `python poetryCompier.py macbeth.dsl`, you might find yourself not being able to avoid the impression that the output is rather verbose. Just looking at the beginning of the output, we find:

```
<document>
    <:ZeroOrMore>
        <sentence>
            <part>
                <WORD>
                    <:RegExp>Life's</:RegExp>
                    <:Whitespace> </:Whitespace>
                </WORD>
                <WORD>
                    <:RegExp>but</:RegExp>
                    <:Whitespace> </:Whitespace>
                </WORD>
...
```

But why do we need to know all those details! Why would we need a ":ZeroOrMore" element inside the "<document>" element, if the "<sentence>"-elements could just as well be direct descendants of the "<document>"-element? Why do we need the information that "Life's" has been captured by a regular expression parser? Wouldn't it suffice to know that the word captured is "Life's"? And is the whitespace really needed at all? If the words in a sequence are separated by definition by whitespace, then it would suffice to have the word without whitespace in our tree, and to add whitespace only later when transforming the tree into some kind of output format. (On the other hand, it might be convenient to have it in the tree never the less…)

Well, the answer to most most of these questions is that what our compilation script yields is more or less the output that the parser yields which in turn is the *concrete syntax tree* of the parsed text. Being a concrete syntax tree it is by its very nature very verbose, because it captures every minute syntactic detail described in the grammar and found in the text, no matter how irrelevant it is, if we are primarily interested in the structure of our text. In order for our tree to become more handy we have to transform it into an *abstract syntax tree* first, which is called thus because it abstracts from all details that deem us irrelevant. Now, which details we consider as irrelevant is almost entirely up to ourselves. And we should think carefully about what features must be included in the abstract syntax tree, because the abstract syntax tree more or less reflects the data model (or is at most one step away from it) with which want to capture our material.

For the sake of our example, let's assume that we are not interested in whitespace and that we want to get rid of all uniformative nodes, i.e. nodes that merely demark syntactic structures but not semantic entities.

DHParser supports the transformation of the concrete syntax tree (CST) into the abstract syntax tree (AST) with a simple technology that (in theory) allows to specify the necessary transformations in an almost delcarative fashion: You simply fill in a Python-dictionary of tag-names with transformation *operators*. Technically, these operators are simply Python-functions. DHParser comes with a rich set of predefined operators. Should these not suffice, you can easily write your own. How does this look like?

```
poetry_AST_transformation_table = {
    "+": remove_empty,
    "document": [],
    "sentence": [],
    "part": [],
    "WORD": [],
    "EOF": [],
    ":Token, :RE": reduce_single_child,
    "*": replace_by_single_child
}
```

You'll find this table in the script `poetryCompiler.py`, which is also the place where you edit the table, because then it is automatically used when compiling your DSL-sources. Now, AST-Transformation works as follows: The whole tree is scanned, starting at the deepest level and applying the specified operators and then working its way upward. This means that the operators specified for "WORD"-nodes will be applied before the operators of "part"-nodes and "sentence"-nodes. This has the advantage that when a particular node is reached the transformations for its descendant nodes have already been applied.

As you can see, the transformation-table contains an entry for every known parser, i.e. "document", "sentence", "part", "WORD", "EOF". (If any of these are missing in the table of your `poetryCompiler.py`, add them now!) In the template you'll also find transformations for two anonymous parsers, i.e. ":Token" and ":RE" as well as some curious entries such as "*" and "+". The latter are considered to be "jokers". The transformations related to the "+"-sign will be applied on any node, before any other transformation is applied. In this case, all empty nodes will be removed first (transformation: `remove_empty`). The "*"-joker contains a list of transformations that will be applied to all those tags that have not been entered explicitly into the transformation table. For example, if the transformation reaches a node with the tag-name ":ZeroOrMore" (i.e. an anonymous node that has been generated by the parser ":ZeroOrmore"), the "*"-joker-operators will be applied. In this case it is just one transformation, namely, `replace_by_single_child` which replaces a node that has but one child by its child. In contrast, the transformation `reduce_single_child` eliminates a single child node by attaching the child's children or content directly to the parent node. We'll see what this means and how this works, briefly.

Caution: Once the compiler-script "xxxxCompiler.py" has been generated, the *only* part that is changed after editing and extending the grammar is the parser-part of this script (i.e. the class derived from class Grammar), because this part is completly auto-generated and can therefore be overwritten safely. The other parts of that script, including the AST-transformation-dictionary, if never changed once it has been generated, because it needs to be filled in by hand by the designer of the DSL and the hand-made changes should not be overwritten. There it is left as it is when regenerating the parser. However, this means, if you add symbols to your grammar later, you will not find them as keys in the AST-transformation-table, but you'll have to add them yourself.

The comments in the compiler-script clearly indicate which parts can be edited by hand safely, i.e. without running the risk of being overwritten, an which cannot.

We can either specify no operator (empty list), a single operator or a list of operators for transforming a node. There is a different between specifying an empty list for a particular tag-name or leaving out a tag-name completly. In the latter case the "*"-joker is applied, in place of the missing list of operators. In the former case only the "+"-joker is applied. If a list of operators is specified, these operator will be applied in sequence one after the other. We also call the list of operators or the single operator if there is only one the *transformation* for a particular tag (or parser name or parser type for that matter).

Because the AST-transfomation works through the table from the inside to the outside, it is reasonable to do the same when designing the AST-transformations, to proceed in the same order. The innermost nodes that concern us are the nodes captured by the <WORD>-parser, or simply, <WORD>-nodes. As we can see, these nodes usually contain a <:RegExp>-node and a <:Whitespace>-node. As the "WORD" parser is defined as a simple regular expresseion with followed by optional whitespace in our grammar, we now that this must always be the case, although the whitespace may occasionally be empty. Thus, we can eliminate the uninformative child nodes by removing whitespace first and the reducing the single left over child node. The respective line in the AST-transformation-table in the compiler-script should be changed as follows:

```
"WORD": [remove_whitespace, reduce_single_child],
```

Running the "poetryCompiler.py"-script on "macbeth.dsl" again, yields:

```
<document>
  <:ZeroOrMore>
    <sentence>
      <part>
        <WORD>Life's</WORD>
        <WORD>but</WORD>
        <WORD>a</WORD>
        <WORD>walking</WORD>
        <WORD>shadow</WORD>
      </part>
      <:Series>
        <:Token>
          <:PlainText>,</:PlainText>
          <:Whitespace> </:Whitespace>
        </:Token>
        <part>
          <WORD>a</WORD>
...
```

It starts to become more readble and concise, but there are sill some oddities. Firstly, the Tokens that deliminate parts of sentences still contain whitespace. Secondly, if several <part>-nodes follow each other in a <sentence>-node, the <part>-nodes after the first one are enclosed by a <:Series>-node or even a cascade of <:ZeroOrMore> and <:Series>-nodes. As for the <:Token>-nodes, have can do the same trick as with the WORD-nodes:

```
":Token": [remove_whitespace, reduce_single_child],
":RE": reduce_single_child,
```

As to the nested structure of the <part>-nodes within the <sentence>-node, this a rather typical case of syntactic artefacts that can be found in concrete syntax trees. It is obviously a consequence of the grammar definition:

```
sentence = part {"," part } "."
```

We'd of course prefer to have flat structure of parts and punctuation marks following each other within the sentence. Since this is a standard case, DHParser includes a special operator to "flatten" nested structures of this kind:

```
"sentence" = [flatten],
```

The `flatten` operator recursively eliminates all intermediary anonymous child nodes. We do not need to do anything in particular for transforming the <part>-node, except that we should explicitly assign an empty operator-list to it, because we do not want the "*" to step in. The reason is that a <part> with a single <WORD> should still be visible as a part a not replaced by the <WORD>-node, because we would like our data model to have has regular a form as possible. (This does of course imply a decision that we have taken on the form of our data model, which would lead too far to discuss here. Suffice it to say that depending on the occasion and purpose, such decisions can also be taken otherwise.)

The only kind of nodes left are the <document>-nodes. In the output of the compiler-script (see above), the <document>-node had a single childe node ":ZeroOrMore". Since this child node does not have any particular semantic meaning it would reasonable to eliminate it and attach its children directly to "document". We could do so by entering `reduce_single_child` in the lost of transformations for "document"-nodes. However, when designing the AST-transformations, it is important not only to consider the concrete output that a particular text yields, but all possible outputs. Therefore, before specifying a transformation, we should also take a careful look at the grammar again, where "document" is defined as follows:

```
document = ~ { sentence } §EOF
```

As we can see a "document"-node may also contain whitespace and an EOF-marker. The reason why we don't find these in the output is that empty nodes have been eliminated by the `remove_empty`-transformation specified in the "+"-joker, before. While EOF is always empty (little exercise: explain why!). But there could be ":Whitespace"-nodes next to the zero or more sentences in the document node, in which case the "reduce_single_child"-operator would do nothing, because there is more than a single child. (We could of course also use the "flatten"-operator, instead. Try this as an exercise.) Test cases help to capture those different scenarios, so adding test cases and examining the output in the test report halp to get a grip on this, if just looking at the grammar strains you imagination too much.

Since we have decided, that we do not want to include whitespace in our data model, we can simply eliminate any whitespace before we apply the `reduce_single_child`-operator, so we change the "document"-entry in the AST-transformation-table as thus:

```
"document": [remove_whitespace, reduce_single_child],
```

Now that everything is set, let's have a look at the result:

```
<document>
  <sentence>
    <part>
      <WORD>Life's</WORD>
      <WORD>but</WORD>
      <WORD>a</WORD>
      <WORD>walking</WORD>
      <WORD>shadow</WORD>
    </part>
    <:Token>,</:Token>
    <part>
      <WORD>a</WORD>
      <WORD>poor</WORD>
      <WORD>player</WORD>
...
```

That is much better. There is but one slight blemish in the output: While all nodes left a named nodes, i.e. nodes associated with a named parser, there are a few anonymous <:Token> nodes. Here is a little exercise: Do away with those <:Token>-nodes by replacing them by something semantically more meaningful. Hint: Add a new symbol "delimiter" in the grammar definition "poetry.ebnf". An alternative strategy to extending the grammar would be to use the `replace_parser` operator. Which of the strategy is the better one? Explain why.

CHAPTER

TWO

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 "redactation 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 receipients. 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.

CHAPTER

THREE

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 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 maschine 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 wirtten 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 a carried out be the computer without any user intervention, i.e. without the need of humans to rewrite or enrich the data during any 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 if 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/snytaxtree.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/ebnf.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

CHAPTER

FOUR

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 across 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 DHParser 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 `PreprocessorToken`-parser.

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], functools.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], functools.partial]]]], functools.partial]

Merges a seuqence 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.

ptype

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=”, ptype=”*)

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 ptype 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 singelton 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’), DHParser.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.parer, ...)`.

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.

errors
list – A list of all errors that occured on this node.

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.

as_sxpr(*src: str = None*, *compact: bool = False*, *showerrors: bool = True*, *indentation: int = 2*) → 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*, *indentation: int = 2*) → 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.

content
Returns content as string, omitting error messages.

init_pos (*pos: int*) → 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.AbstractSet[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 = parse_sxpr('(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 tage 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 *ptype*.

tree_size() → int
Recursively counts the number of nodes in the tree including the root node.

class RootNode(*node: Union[syntaxtree.Node, NoneType] = None*) → syntaxtree.RootNode
TODO: Add Documentation!!!

errors (list): A list of all errors that have occured so far during processing (i.e. parsing, AST-transformation, compiling) of this tree.

error_flag (int): the highest warning or error level of all errors that occurred.

add_error(*node: syntaxtree.Node, error: DHParser.error.Error*) → syntaxtree.RootNode
Adds an Error object to the tree, locating it at a specific node.

collect_errors() → List[DHParser.error.Error]
Returns the list of errors, ordered bv their position.

new_error(*node: syntaxtree.Node, message: str, code: int = 1000*) → syntaxtree.RootNode
Adds an error to this tree, locating it at a specific node. :param pos: The position of the error in the source text :type pos: int :param message: A string with the error message.abs :type message: str :param code: An error code to identify the kind of error :type code: int

parse_sxpr(*sxpr: str*) → syntaxtree.Node
Generates a tree of nodes from an S-expression.

This can - among other things - be used for deserialization of trees that have been serialized with *Node.as_sxpr()* or as a convenient way to generate test data.

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

parse_xml(*xml: str*) → syntaxtree.Node
Generates a tree of nodes from a (Pseudo-)XML-source.

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))'

flatten_xml(*xml: str*) → str
Returns an XML-tree as a one linter without unnecessary whitespace, i.e. only whitespace within leaf-nodes is preserved.

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 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 resault 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 Grammer object is called with a parser that does not exist or if in the course of parsing a parser is reffered 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 accross 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_initializiation__
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.

python__src__
For the purpose of debugging and inspection, this field can take the python src of the concrete grammar class (see *dsl.grammar_provider*).

all_parsers__
A set of all parsers connected to this grammar object

history_tracking__
A flag indicating that the parsing history shall be tracked

whitespace__
A parser for the implicit optional 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`. It can also be place explicitly in the EBNF-Grammar via the "~"-sign.

wsp_left_parser__
The same as `whitespace` for left-adjacent-whitespace.

wsp_right_parser__
The same as `whitespace` for right-adjacent-whitespace.

_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.

tree__
The root-node of the parsing tree. This variable is available for error-reporting already during parsing via

`self.grammar.tree__.add_error`, but it references the full parsing tree only after parsing has been finished.

`_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 Whitespace(*regexp*, *name: str = ”*) → None
An variant of RegExp that signifies through its class name that it is a RegExp-parser for whitespace.

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 cann occur whereever 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 returnd 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(*left: parse.Parser*, *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 overriden 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 overriden 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: DHParser.stringview.StringView*) → Tuple[Union[DHParser.syntaxtree.Node, NoneType], DHParser.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 `context`-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 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: DHParser.syntaxtree.Node, processing_table: Dict[str, Union[typing.Sequence[typing.Callable], typing.Dict[str, typing.Sequence[typing.Callable]]]], key_func: Callable[DHParser.syntaxtree.Node, str] = <function key_tag_name>*) → None
Traverses the snytax 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[DHParser.syntaxtree.Node]*) → bool
: Returns `True` if the current node's parser is a named parser.

replace_by_single_child(*context: List[DHParser.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[DHParser.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[DHParser.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[DHParser.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[DHParser.syntaxtree.Node]*)
: Collapses all sub-nodes of a node by replacing them with the string representation of the node.

merge_children(*context: List[DHParser.syntaxtree.Node], tag_names: Tuple[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[DHParser.syntaxtree.Node], func: Callable*)
: Replaces the content of the node. `func` takes the node's result as an argument an returns the mapped result.

replace_content_by(*context: List[DHParser.syntaxtree.Node], content: str*)
: Replaces the content of the node with the given text content.

apply_if(*context: List[DHParser.syntaxtree.Node], transformation: Callable, condition: Callable*)
: Applies a transformation only if a certain condition is met.

apply_unless (*context: List[DHParser.syntaxtree.Node], transformation: Callable, condition: Callable*)
Applies a transformation if a certain condition is *not* met.

traverse_locally (*context: List[DHParser.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[DHParser.syntaxtree.Node]*) → bool
Returns `True` if the current node's parser is an anonymous parser.

is_whitespace (*context: List[DHParser.syntaxtree.Node]*) → bool
Removes whitespace and comments defined with the `@comment`-directive.

is_empty (*context: List[DHParser.syntaxtree.Node]*) → bool
Returns `True` if the current node's content is empty.

is_expendable (*context: List[DHParser.syntaxtree.Node]*) → bool
Returns `True` if the current node either is a node containing whitespace or an empty node.

is_token (*context: List[DHParser.syntaxtree.Node], tokens: AbstractSet[str] = frozenset()*) → bool
Checks whether the last node in the context has *ptype == TOKEN_PTYPE* and it's 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.

is_one_of (*context: List[DHParser.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[DHParser.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[DHParser.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[DHParser.syntaxtree.Node], condition: Callable = <function is_expendable>*)
Recursively removes all leading child-nodes that fulfill a given condition.

rstrip (*context: List[DHParser.syntaxtree.Node], condition: Callable = <function is_expendable>*)
Recursively removes all leading nodes that fulfill a given condition.

strip (*context: List[DHParser.syntaxtree.Node], condition: Callable = <function is_expendable>*)
Removes leading and trailing child-nodes that fulfill a given condition.

keep_children (*context: List[DHParser.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[DHParser.syntaxtree.Node], condition: Callable*)
Removes all children for which *condition()* returns *True*.

keep_tokens (*context: List[DHParser.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[DHParser.syntaxtree.Node], tag_names: AbstractSet[str]*)
Removes children by tag name.

keep_content (*context: List[DHParser.syntaxtree.Node], regexp: str*)
Removes children depending on their string value.

remove_children_if(*context: List[DHParser.syntaxtree.Node], condition: Callable*)
Removes all children for which *condition()* returns *True*.

remove_nodes(*context: List[DHParser.syntaxtree.Node], tag_names: AbstractSet[str]*)
Removes children by tag name.

remove_content(*context: List[DHParser.syntaxtree.Node], regexp: str*)
Removes children depending on their string value.

remove_tokens(*context: List[DHParser.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[DHParser.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)
```

error_on(*context: List[DHParser.syntaxtree.Node], condition: Callable, error_msg: str = ''*)
Checks for *condition*; adds an error message if condition is not met.

warn_on(*context: List[DHParser.syntaxtree.Node], condition: Callable, warning: str = ''*)
Checks for *condition*; adds an warning 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 an 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 DHParser'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: DHParser.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: DHParser.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(*node_name: str*) → str
Returns the method name for *node_name*, e.g.:

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

set_grammar_name(*grammar_name: str = ", grammar_source: str = "*)
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. Returns *self*.

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: DHParser.parse.Grammar, transformer: Union[typing.Callable[[DHParser.syntaxtree.Node], typing.Any], functools.partial], compiler: compile.Compiler*) → Tuple[[Any, List[DHParser.error.Error]], DHParser.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 DHParser. Usually, what is of interest are the string representations of the error objects. For example:

```
from DHParser 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):
        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[DHParser.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[DHParser.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

DHParser 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 DHParser based Grammar class that can be executed to parse source text conforming to this grammar into contrete 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. DHParser 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 contrete syntax trees.

class EBNFGrammar(*root: DHParser.parse.Parser = None*) → 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] plaintext
           | [flowmarker] regexp
           | [flowmarker] whitespace
           | [flowmarker] oneormore
           | [flowmarker] group
           | [flowmarker] unordered
           | repetition
           | option

flowmarker = "!"  | "&"                         # '!' negative lookahead, '&'␣
↪positive lookahead
```

(continues on next page)

(continued from previous page)

```
           | "-!" | "-&"                          # '-' 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 tacitly.
plaintext  = /`(?:[^"]|\\")*?`/~                # like literal but does not eat␣
↪whitespace
regexp     = /~?\/(?:\\\/|[^\/])*?\/~?/~        # e.g. /\w+/, ~/#.*(?:\n|$)/~
                                                # '~' is a whitespace-marker, if␣
↪present leading or trailing
                                                # whitespace of a regular␣
↪expression will be ignored tacitly.
whitespace = /~/~                               # implicit or default whitespace
list_      = /\w+/~ { "," /\w+/~ }              # comma separated list of symbols,
↪ e.g. BEGIN_LIST, END_LIST,
                                                # BEGIN_QUOTE, END_QUOTE ; see␣
↪CommonMark/markdown.py for an exmaple
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.

Addionally, 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 compilatation, 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: DHParser.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: DHParser.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.

verify_transformation_table(*transtable*)
Checks for symbols that occur in the transformation-table but have never been defined in the grammar. Usually, this kind of inconsistency results from an error like a typo in the transformation table.

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[DHParser.parse.Grammar], Callable[Union[typing.Callable[[DHParser.syntaxtree.Node], typing.Any], functools.partial]]], Callable[DHParser.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, DHParser.parse.Grammar]*, *ast_transformation: Union[typing.Callable[[DHParser.syntaxtree.Node], typing.Any], functools.partial]*, *compiler: DHParser.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'*) → DHParser.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'*) → DHParser.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[DHParser.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 asterix 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 asterix 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 DHParser.testing import runner runner("", globals())

4.3 Supporting Modules Reference

Finally, DHParser comprises a number of "toolkit"-modules which define helpful functions and classes that will are used at different places throughout the other DHParser-modules.

4.3.1 Module `toolkit`

Module `toolkit` contains utility functions that are needed across several of the the other DHParser-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 DHParser 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 abreviated parsing history.

Example:

```
from DHParser 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({'.ast', '.log', '.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: DHParser.syntaxtree.Node, text: DHParser.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 ist 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(*history: List[_ForwardRef('HistoryRecord')]*) → 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(*history: List[_ForwardRef('HistoryRecord')]*) → 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 = True*) → 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 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.

It is recommended to compile this modules with the Cython-compiler for speedup. The modules comes with a `stringview.pxd` that contains some type declarations to fully exploit the potential of the Cython-compiler.

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.

finditer

Executes regex.finditer on the StringView object and returns the iterator of match objects. WARNING: match.end(), match.span() etc. are mapped to the underlying text,

> not the StringView-object!!!

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()*

lstrip

Returns a copy of *self* with leading whitespace removed.

match

Executes *regex.match* on the StringView object and returns the result, which is either a match-object or None. WARNING: match.end(), match.span() etc. are mapped to the underlying text,

> not the StringView-object!!!

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.

rstrip

Returns a copy of *self* with trailing whitespace removed.

search

Executes regex.search on the StringView object and returns the result, which is either a match-object or None. WARNING: match.end(), match.span() etc. are mapped to the underlying text,

> not the StringView-object!!!

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`

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX