Lark Documentation

Erez Shinan

Nov 25, 2020

Overview

CHAPTER 1

Philosophy

Parsers are innately complicated and confusing. They're difficult to understand, difficult to write, and difficult to use. Even experts on the subject can become baffled by the nuances of these complicated state-machines.

Lark's mission is to make the process of writing them as simple and abstract as possible, by following these design principles:

1.1 Design Principles

1. Readability matters
2. Keep the grammar clean and simple
3. Don't force the user to decide on things that the parser can figure out on its own
4. Usability is more important than performance
5. Performance is still very important
6. Follow the Zen of Python, whenever possible and applicable

In accordance with these principles, I arrived at the following design choices:

1.2 Design Choices

1.2.1 1. Separation of code and grammar

Grammars are the de-facto reference for your language, and for the structure of your parse-tree. For any non-trivial language, the conflation of code and grammar always turns out convoluted and difficult to read.

The grammars in Lark are EBNF-inspired, so they are especially easy to read & work with.

1.2.2 2. Always build a parse-tree (unless told not to)

Trees are always simpler to work with than state-machines.

1. Trees allow you to see the "state-machine" visually
2. Trees allow your computation to be aware of previous and future states
3. Trees allow you to process the parse in steps, instead of forcing you to do it all at once.

And anyway, every parse-tree can be replayed as a state-machine, so there is no loss of information.

See this answer in more detail here.

To improve performance, you can skip building the tree for LALR(1), by providing Lark with a transformer (see the JSON example).

1.2.3 3. Earley is the default

The Earley algorithm can accept *any* context-free grammar you throw at it (i.e. any grammar you can write in EBNF, it can parse). That makes it extremely friendly to beginners, who are not aware of the strange and arbitrary restrictions that LALR(1) places on its grammars.

As the users grow to understand the structure of their grammar, the scope of their target language, and their performance requirements, they may choose to switch over to LALR(1) to gain a huge performance boost, possibly at the cost of some language features.

In short, "Premature optimization is the root of all evil."

1.2.4 Other design features

- Automatically resolve terminal collisions whenever possible
- Automatically keep track of line & column numbers

CHAPTER 2

Features

2.1 Main Features

- Earley parser, capable of parsing any context-free grammar
  - Implements SPPF, for efficient parsing and storing of ambiguous grammars.
- LALR(1) parser, limited in power of expression, but very efficient in space and performance (O(n)).
  - Implements a parse-aware lexer that provides a better power of expression than traditional LALR implementations (such as ply).
- EBNF-inspired grammar, with extra features (See: *Grammar Reference*)
- Builds a parse-tree (AST) automagically based on the grammar
- Stand-alone parser generator - create a small independent parser to embed in your project.
- Flexible error handling by using a "puppet parser" mechanism (LALR only)
- Automatic line & column tracking (for both tokens and matched rules)
- Automatic terminal collision resolution
- Standard library of terminals (strings, numbers, names, etc.)
- Unicode fully supported
- Extensive test suite
- MyPy support using type stubs
- Python 2 & Python 3 compatible
- Pure-Python implementation

Read more about the parsers

2.2 Extra features

- Import rules and tokens from other Lark grammars, for code reuse and modularity.
- Support for external regex module (see here)
- Import grammars from Nearley.js (*read more*)
- CYK parser
- Visualize your parse trees as dot or png files (see_example)

2.2.1 Experimental features

- Automatic reconstruction of input from parse-tree (see examples)

2.2.2 Planned features (not implemented yet)

- Generate code in other languages than Python
- Grammar composition
- LALR(k) parser
- Full regexp-collision support using NFAs

CHAPTER 3

Parsers

Lark implements the following parsing algorithms: Earley, LALR(1), and CYK

3.1 Earley

An Earley Parser is a chart parser capable of parsing any context-free grammar at O(n^3), and O(n^2) when the grammar is unambiguous. It can parse most LR grammars at O(n). Most programming languages are LR, and can be parsed at a linear time.

Lark's Earley implementation runs on top of a skipping chart parser, which allows it to use regular expressions, instead of matching characters one-by-one. This is a huge improvement to Earley that is unique to Lark. This feature is used by default, but can also be requested explicitly using `lexer='dynamic'`.

It's possible to bypass the dynamic lexing, and use the regular Earley parser with a traditional lexer, that tokenizes as an independent first step. Doing so will provide a speed benefit, but will tokenize without using Earley's ambiguity-resolution ability. So choose this only if you know why! Activate with `lexer='standard'`

SPPF & Ambiguity resolution

Lark implements the Shared Packed Parse Forest data-structure for the Earley parser, in order to reduce the space and computation required to handle ambiguous grammars.

You can read more about SPPF here

As a result, Lark can efficiently parse and store every ambiguity in the grammar, when using Earley.

Lark provides the following options to combat ambiguity:

1. Lark will choose the best derivation for you (default). Users can choose between different disambiguation strategies, and can prioritize (or demote) individual rules over others, using the rule-priority syntax.
2. Users may choose to receive the set of all possible parse-trees (using ambiguity='explicit'), and choose the best derivation themselves. While simple and flexible, it comes at the cost of space and performance, and so it isn't recommended for highly ambiguous grammars, or very long inputs.
3. As an advanced feature, users may use specialized visitors to iterate the SPPF themselves.

lexer="dynamic_complete"

Earley's "dynamic" lexer uses regular expressions in order to tokenize the text. It tries every possible combination of terminals, but it matches each terminal exactly once, returning the longest possible match.

That means, for example, that when `lexer="dynamic"` (which is the default), the terminal `/a+/`, when given the text `"aa"`, will return one result, `aa`, even though `a` would also be correct.

This behavior was chosen because it is much faster, and it is usually what you would expect.

Setting `lexer="dynamic_complete"` instructs the lexer to consider every possible regexp match. This ensures that the parser will consider and resolve every ambiguity, even inside the terminals themselves. This lexer provides the same capabilities as scannerless Earley, but with different performance tradeoffs.

Warning: This lexer can be much slower, especially for open-ended terminals such as `/.*/`

3.2 LALR(1)

LALR(1) is a very efficient, true-and-tested parsing algorithm. It's incredibly fast and requires very little memory. It can parse most programming languages (For example: Python and Java).

Lark comes with an efficient implementation that outperforms every other parsing library for Python (including PLY)

Lark extends the traditional YACC-based architecture with a *contextual lexer*, which automatically provides feedback from the parser to the lexer, making the LALR(1) algorithm stronger than ever.

The contextual lexer communicates with the parser, and uses the parser's lookahead prediction to narrow its choice of terminals. So at each point, the lexer only matches the subgroup of terminals that are legal at that parser state, instead of all of the terminals. It's surprisingly effective at resolving common terminal collisions, and allows one to parse languages that LALR(1) was previously incapable of parsing.

(If you're familiar with YACC, you can think of it as automatic lexer-states)

This is an improvement to LALR(1) that is unique to Lark.

3.3 CYK Parser

A CYK parser can parse any context-free grammar at O(n^3*|G|).

Its too slow to be practical for simple grammars, but it offers good performance for highly ambiguous grammars.

CHAPTER 4

JSON parser - Tutorial

Lark is a parser - a program that accepts a grammar and text, and produces a structured tree that represents that text. In this tutorial we will write a JSON parser in Lark, and explore Lark's various features in the process.

It has 5 parts.

1. Writing the grammar
2. Creating the parser
3. Shaping the tree
4. Evaluating the tree
5. Optimizing

Knowledge assumed:

- Using Python
- A basic understanding of how to use regular expressions

4.1 Part 1 - The Grammar

Lark accepts its grammars in a format called EBNF. It basically looks like this:

```
rule_name : list of rules and TERMINALS to match
          | another possible list of items
          | etc.

TERMINAL: "some text to match"
```

(*a terminal is a string or a regular expression*)

The parser will try to match each rule (left-part) by matching its items (right-part) sequentially, trying each alternative (In practice, the parser is predictive so we don't have to try every alternative).

How to structure those rules is beyond the scope of this tutorial, but often it's enough to follow one's intuition.

In the case of JSON, the structure is simple: A json document is either a list, or a dictionary, or a string/number/etc.

The dictionaries and lists are recursive, and contain other json documents (or "values").

Let's write this structure in EBNF form:

```
value: dict
     | list
     | STRING
     | NUMBER
     | "true" | "false" | "null"

list : "[" [value ("," value)*] "]"

dict : "{" [pair ("," pair)*] "}"
pair : STRING ":" value
```

A quick explanation of the syntax:

- Parenthesis let us group rules together.
- rule* means *any amount*. That means, zero or more instances of that rule.
- [rule] means *optional*. That means zero or one instance of that rule.

Lark also supports the rule+ operator, meaning one or more instances. It also supports the rule? operator which is another way to say *optional*.

Of course, we still haven't defined "STRING" and "NUMBER". Luckily, both these literals are already defined in Lark's common library:

```
%import common.ESCAPED_STRING   -> STRING
%import common.SIGNED_NUMBER    -> NUMBER
```

The arrow (->) renames the terminals. But that only adds obscurity in this case, so going forward we'll just use their original names.

We'll also take care of the white-space, which is part of the text.

```
%import common.WS
%ignore WS
```

We tell our parser to ignore whitespace. Otherwise, we'd have to fill our grammar with WS terminals.

By the way, if you're curious what these terminals signify, they are roughly equivalent to this:

```
NUMBER : /-?\d+(\.\d+)?([eE][+-]?\d+)?/
STRING : /".*?(?<!\\)"/
%ignore /[ \t\n\f\r]+/
```

Lark will accept this, if you really want to complicate your life :)

You can find the original definitions in common.lark. They don't strictly adhere to json.org - but our purpose here is to accept json, not validate it.

Notice that terminals are written in UPPER-CASE, while rules are written in lower-case. I'll touch more on the differences between rules and terminals later.

4.2 Part 2 - Creating the Parser

Once we have our grammar, creating the parser is very simple.

We simply instantiate Lark, and tell it to accept a "value":

```
from lark import Lark
json_parser = Lark(r"""
    value: dict
         | list
         | ESCAPED_STRING
         | SIGNED_NUMBER
         | "true" | "false" | "null"

    list : "[" [value ("," value)*] "]"

    dict : "{" [pair ("," pair)*] "}"
    pair : ESCAPED_STRING ":" value

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
    %import common.WS
    %ignore WS

    """, start='value')
```

It's that simple! Let's test it out:

```
>>> text = '{"key": ["item0", "item1", 3.14]}'
>>> json_parser.parse(text)
Tree(value, [Tree(dict, [Tree(pair, [Token(STRING, "key"), Tree(value, [Tree(list,␣
↪[Tree(value, [Token(STRING, "item0")]), Tree(value, [Token(STRING, "item1")]),␣
↪Tree(value, [Token(NUMBER, 3.14)])])])])])])
>>> print( _.pretty() )
value
  dict
    pair
      "key"
      value
        list
          value        "item0"
          value        "item1"
          value        3.14
```

As promised, Lark automagically creates a tree that represents the parsed text.

But something is suspiciously missing from the tree. Where are the curly braces, the commas and all the other punctuation literals?

Lark automatically filters out literals from the tree, based on the following criteria:

- Filter out string literals without a name, or with a name that starts with an underscore.
- Keep regexps, even unnamed ones, unless their name starts with an underscore.

Unfortunately, this means that it will also filter out literals like "true" and "false", and we will lose that information. The next section, "Shaping the tree" deals with this issue, and others.

4.3 Part 3 - Shaping the Tree

We now have a parser that can create a parse tree (or: AST), but the tree has some issues:

1. "true", "false" and "null" are filtered out (test it out yourself!)
2. Is has useless branches, like *value*, that clutter-up our view.

I'll present the solution, and then explain it:

```
?value: dict
      | list
      | string
      | SIGNED_NUMBER      -> number
      | "true"             -> true
      | "false"            -> false
      | "null"             -> null

...

string : ESCAPED_STRING
```

1. Those little arrows signify *aliases*. An alias is a name for a specific part of the rule. In this case, we will name the *true/false/null* matches, and this way we won't lose the information. We also alias *SIGNED_NUMBER* to mark it for later processing.
2. The question-mark prefixing *value* ("?value") tells the tree-builder to inline this branch if it has only one member. In this case, *value* will always have only one member, and will always be inlined.
3. We turned the *ESCAPED_STRING* terminal into a rule. This way it will appear in the tree as a branch. This is equivalent to aliasing (like we did for the number), but now *string* can also be used elsewhere in the grammar (namely, in the *pair* rule).

Here is the new grammar:

```
from lark import Lark
json_parser = Lark(r"""
    ?value: dict
          | list
          | string
          | SIGNED_NUMBER      -> number
          | "true"             -> true
          | "false"            -> false
          | "null"             -> null

    list : "[" [value ("," value)*] "]"

    dict : "{" [pair ("," pair)*] "}"
    pair : string ":" value

    string : ESCAPED_STRING

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
    %import common.WS
    %ignore WS

    """, start='value')
```

And let's test it out:

```
>>> text = '{"key": ["item0", "item1", 3.14, true]}'
>>> print( json_parser.parse(text).pretty() )
dict
  pair
    string        "key"
    list
      string        "item0"
      string        "item1"
      number        3.14
      true
```

Ah! That is much much nicer.

4.4 Part 4 - Evaluating the tree

It's nice to have a tree, but what we really want is a JSON object.

The way to do it is to evaluate the tree, using a Transformer.

A transformer is a class with methods corresponding to branch names. For each branch, the appropriate method will be called with the children of the branch as its argument, and its return value will replace the branch in the tree.

So let's write a partial transformer, that handles lists and dictionaries:

```
from lark import Transformer

class MyTransformer(Transformer):
    def list(self, items):
        return list(items)
    def pair(self, key_value):
        k, v = key_value
        return k, v
    def dict(self, items):
        return dict(items)
```

And when we run it, we get this:

```
>>> tree = json_parser.parse(text)
>>> MyTransformer().transform(tree)
{Tree(string, [Token(ANONRE_1, "key")]): [Tree(string, [Token(ANONRE_1, "item0")]),␣
↪Tree(string, [Token(ANONRE_1, "item1")]), Tree(number, [Token(ANONRE_0, 3.14)]),␣
↪Tree(true, [])]}
```

This is pretty close. Let's write a full transformer that can handle the terminals too.

Also, our definitions of list and dict are a bit verbose. We can do better:

```
from lark import Transformer

class TreeToJson(Transformer):
    def string(self, s):
        (s,) = s
        return s[1:-1]
    def number(self, n):
        (n,) = n
        return float(n)
```

(continues on next page)

(continued from previous page)

```
    list = list
    pair = tuple
    dict = dict

    null = lambda self, _: None
    true = lambda self, _: True
    false = lambda self, _: False
```

And when we run it:

```
>>> tree = json_parser.parse(text)
>>> TreeToJson().transform(tree)
{u'key': [u'item0', u'item1', 3.14, True]}
```

Magic!

4.5 Part 5 - Optimizing

4.5.1 Step 1 - Benchmark

By now, we have a fully working JSON parser, that can accept a string of JSON, and return its Pythonic representation.

But how fast is it?

Now, of course there are JSON libraries for Python written in C, and we can never compete with them. But since this is applicable to any parser you would write in Lark, let's see how far we can take this.

The first step for optimizing is to have a benchmark. For this benchmark I'm going to take data from json-generator.com/. I took their default suggestion and changed it to 5000 objects. The result is a 6.6MB sparse JSON file.

Our first program is going to be just a concatenation of everything we've done so far:

```
import sys
from lark import Lark, Transformer

json_grammar = r"""
    ?value: dict
          | list
          | string
          | SIGNED_NUMBER      -> number
          | "true"             -> true
          | "false"            -> false
          | "null"             -> null

    list : "[" [value ("," value)*] "]"

    dict : "{" [pair ("," pair)*] "}"
    pair : string ":" value

    string : ESCAPED_STRING

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
```

(continues on next page)

(continued from previous page)

```
    %import common.WS
    %ignore WS
    """

class TreeToJson(Transformer):
    def string(self, s):
        (s,) = s
        return s[1:-1]
    def number(self, n):
        (n,) = n
        return float(n)

    list = list
    pair = tuple
    dict = dict

    null = lambda self, _: None
    true = lambda self, _: True
    false = lambda self, _: False

json_parser = Lark(json_grammar, start='value', lexer='standard')

if __name__ == '__main__':
    with open(sys.argv[1]) as f:
        tree = json_parser.parse(f.read())
        print(TreeToJson().transform(tree))
```

We run it and get this:

```
$ time python tutorial_json.py json_data > /dev/null

real        0m36.257s
user        0m34.735s
sys         0m1.361s
```

That's unsatisfactory time for a 6MB file. Maybe if we were parsing configuration or a small DSL, but we're trying to handle large amount of data here.

Well, turns out there's quite a bit we can do about it!

4.5.2 Step 2 - LALR(1)

So far we've been using the Earley algorithm, which is the default in Lark. Earley is powerful but slow. But it just so happens that our grammar is LR-compatible, and specifically LALR(1) compatible.

So let's switch to LALR(1) and see what happens:

```
json_parser = Lark(json_grammar, start='value', parser='lalr')
```

```
$ time python tutorial_json.py json_data > /dev/null

real        0m7.554s
user        0m7.352s
sys         0m0.148s
```

Ah, that's much better. The resulting JSON is of course exactly the same. You can run it for yourself and see.

It's important to note that not all grammars are LR-compatible, and so you can't always switch to LALR(1). But there's no harm in trying! If Lark lets you build the grammar, it means you're good to go.

4.5.3 Step 3 - Tree-less LALR(1)

So far, we've built a full parse tree for our JSON, and then transformed it. It's a convenient method, but it's not the most efficient in terms of speed and memory. Luckily, Lark lets us avoid building the tree when parsing with LALR(1).

Here's the way to do it:

```
json_parser = Lark(json_grammar, start='value', parser='lalr',␣
↪transformer=TreeToJson())

if __name__ == '__main__':
    with open(sys.argv[1]) as f:
        print( json_parser.parse(f.read()) )
```

We've used the transformer we've already written, but this time we plug it straight into the parser. Now it can avoid building the parse tree, and just send the data straight into our transformer. The *parse()* method now returns the transformed JSON, instead of a tree.

Let's benchmark it:

```
real        0m4.866s
user        0m4.722s
sys         0m0.121s
```

That's a measurable improvement! Also, this way is more memory efficient. Check out the benchmark table at the end to see just how much.

As a general practice, it's recommended to work with parse trees, and only skip the tree-builder when your transformer is already working.

4.5.4 Step 4 - PyPy

PyPy is a JIT engine for running Python, and it's designed to be a drop-in replacement.

Lark is written purely in Python, which makes it very suitable for PyPy.

Let's get some free performance:

```
$ time pypy tutorial_json.py json_data > /dev/null

real        0m1.397s
user        0m1.296s
sys         0m0.083s
```

PyPy is awesome!

4.5.5 Conclusion

We've brought the run-time down from 36 seconds to 1.1 seconds, in a series of small and simple steps.

Now let's compare the benchmarks in a nicely organized table.

I measured memory consumption using a little script called memusg

I added a few other parsers for comparison. PyParsing and funcparselib fair pretty well in their memory usage (they don't build a tree), but they can't compete with the run-time speed of LALR(1).

These benchmarks are for Lark's alpha version. I already have several optimizations planned that will significantly improve run-time speed.

Once again, shout-out to PyPy for being so effective.

4.6 Afterword

This is the end of the tutorial. I hoped you liked it and learned a little about Lark.

To see what else you can do with Lark, check out the examples.

For questions or any other subject, feel free to email me at erezshin at gmail dot com.

CHAPTER 5

How To Use Lark - Guide

5.1 Work process

This is the recommended process for working with Lark:

1. Collect or create input samples, that demonstrate key features or behaviors in the language you're trying to parse.
2. Write a grammar. Try to aim for a structure that is intuitive, and in a way that imitates how you would explain your language to a fellow human.
3. Try your grammar in Lark against each input sample. Make sure the resulting parse-trees make sense.
4. Use Lark's grammar features to *shape the tree*: Get rid of superfluous rules by inlining them, and use aliases when specific cases need clarification.

- You can perform steps 1-4 repeatedly, gradually growing your grammar to include more sentences.

1. Create a transformer to evaluate the parse-tree into a structure you'll be comfortable to work with. This may include evaluating literals, merging branches, or even converting the entire tree into your own set of AST classes.

Of course, some specific use-cases may deviate from this process. Feel free to suggest these cases, and I'll add them to this page.

5.2 Getting started

Browse the Examples to find a template that suits your purposes.

Read the tutorials to get a better understanding of how everything works. (links in the *main page*)

Use the Cheatsheet (PDF) for quick reference.

Use the reference pages for more in-depth explanations. (links in the *main page*]

5.3 LALR usage

By default Lark silently resolves Shift/Reduce conflicts as Shift. To enable warnings pass `debug=True`. To get the messages printed you have to configure the `logger` beforehand. For example:

```
import logging
from lark import Lark, logger

logger.setLevel(logging.DEBUG)

collision_grammar = '''
start: as as
as: a*
a: "a"
'''
p = Lark(collision_grammar, parser='lalr', debug=True)
```

CHAPTER 6

How to develop Lark - Guide

There are many ways you can help the project:

- Help solve issues
- Improve the documentation
- Write new grammars for Lark's library
- Write a blog post introducing Lark to your audience
- Port Lark to another language
- Help me with code development

If you're interested in taking one of these on, let me know and I will provide more details and assist you in the process.

6.1 Unit Tests

Lark comes with an extensive set of tests. Many of the tests will run several times, once for each parser configuration.

To run the tests, just go to the lark project root, and run the command:

```
python -m tests
```

or

```
pypy -m tests
```

For a list of supported interpreters, you can consult the `tox.ini` file.

You can also run a single unittest using its class and method name, for example:

```
##   test_package test_class_name.test_function_name
python -m tests TestLalrStandard.test_lexer_error_recovering
```

6.1.1 tox

To run all Unit Tests with tox, install tox and Python 2.7 up to the latest python interpreter supported (consult the file tox.ini). Then, run the command `tox` on the root of this project (where the main setup.py file is on).

And, for example, if you would like to only run the Unit Tests for Python version 2.7, you can run the command `tox -e py27`

6.1.2 pytest

You can also run the tests using pytest:

```
pytest tests
```

6.1.3 Using setup.py

Another way to run the tests is using setup.py:

```
python setup.py test
```

CHAPTER 7

Recipes

A collection of recipes to use Lark and its various features

7.1 Use a transformer to parse integer tokens

Transformers are the common interface for processing matched rules and tokens.

They can be used during parsing for better performance.

```
from lark import Lark, Transformer

class T(Transformer):
    def INT(self, tok):
        "Convert the value of `tok` from string to int, while maintaining line number␣
↪& column."
        return tok.update(value=int(tok))

parser = Lark("""
start: INT*
%import common.INT
%ignore " "
""", parser="lalr", transformer=T())

print(parser.parse('3 14 159'))
```

Prints out:

```
Tree(start, [Token(INT, 3), Token(INT, 14), Token(INT, 159)])
```

7.2 Collect all comments with lexer_callbacks

`lexer_callbacks` can be used to interface with the lexer as it generates tokens.

It accepts a dictionary of the form

```
{TOKEN_TYPE: callback}
```

Where callback is of type `f(Token) -> Token`

It only works with the standard and contextual lexers.

This has the same effect of using a transformer, but can also process ignored tokens.

```
from lark import Lark

comments = []

parser = Lark("""
    start: INT*

    COMMENT: /#.*/

    %import common (INT, WS)
    %ignore COMMENT
    %ignore WS
""", parser="lalr", lexer_callbacks={'COMMENT': comments.append})

parser.parse("""
1 2 3  # hello
# world
4 5 6
""")

print(comments)
```

Prints out:

```
[Token(COMMENT, '# hello'), Token(COMMENT, '# world')]
```

Note: We don't have to return a token, because comments are ignored

7.3 CollapseAmbiguities

Parsing ambiguous texts with earley and `ambiguity='explicit'` produces a single tree with `_ambig` nodes to mark where the ambiguity occurred.

However, it's sometimes more convenient instead to work with a list of all possible unambiguous trees.

Lark provides a utility transformer for that purpose:

```
from lark import Lark, Tree, Transformer
from lark.visitors import CollapseAmbiguities

grammar = """
    !start: x y

    !x: "a" "b"
      | "ab"
      | "abc"
```

(continues on next page)

(continued from previous page)

```
    !y: "c" "d"
      | "cd"
      | "d"

"""
parser = Lark(grammar, ambiguity='explicit')

t = parser.parse('abcd')
for x in CollapseAmbiguities().transform(t):
    print(x.pretty())
```

This prints out:

```
start
x
    a
    b
y
    c
    d

start
x     ab
y     cd

start
x     abc
y     d
```

While convenient, this should be used carefully, as highly ambiguous trees will soon create an exponential explosion of such unambiguous derivations.

7.4 Keeping track of parents when visiting

The following visitor assigns a `parent` attribute for every node in the tree.

If your tree nodes aren't unique (if there is a shared Tree instance), the assert will fail.

```
class Parent(Visitor):
    def __default__(self, tree):
        for subtree in tree.children:
            if isinstance(subtree, Tree):
                assert not hasattr(subtree, 'parent')
                subtree.parent = tree
```

CHAPTER 8

Examples for Lark

How to run the examples:

After cloning the repo, open the terminal into the root directory of the project, and run the following:

```
[lark]$ python -m examples.<name_of_example>
```

For example, the following will parse all the Python files in the standard library of your local installation:

```
[lark]$ python -m examples.python_parser
```

8.1 Beginner Examples

8.1.1 Parsing Indentation

A demonstration of parsing indentation ("whitespace significant" language) and the usage of the Indenter class.

Since indentation is context-sensitive, a postlex stage is introduced to manufacture INDENT/DEDENT tokens.

It is crucial for the indenter that the NL_type matches the spaces (and tabs) after the newline.

```
from lark import Lark
from lark.indenter import Indenter

tree_grammar = r"""
    ?start: _NL* tree

    tree: NAME _NL [_INDENT tree+ _DEDENT]

    %import common.CNAME -> NAME
    %import common.WS_INLINE
    %declare _INDENT _DEDENT
    %ignore WS_INLINE
```

(continues on next page)

(continued from previous page)

```
    _NL: /(\r?\n[\t ]*)+/
"""

class TreeIndenter(Indenter):
    NL_type = '_NL'
    OPEN_PAREN_types = []
    CLOSE_PAREN_types = []
    INDENT_type = '_INDENT'
    DEDENT_type = '_DEDENT'
    tab_len = 8

parser = Lark(tree_grammar, parser='lalr', postlex=TreeIndenter())

test_tree = """
a
    b
    c
        d
        e
    f
        g
"""

def test():
    print(parser.parse(test_tree).pretty())

if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.1.2 Handling Ambiguity

A demonstration of ambiguity

This example shows how to use get explicit ambiguity from Lark's Earley parser.

```
import sys
from lark import Lark, tree

grammar = """
    sentence: noun verb noun        -> simple
            | noun verb "like" noun -> comparative

    noun: adj? NOUN
    verb: VERB
    adj: ADJ

    NOUN: "flies" | "bananas" | "fruit"
    VERB: "like" | "flies"
    ADJ: "fruit"

    %import common.WS
    %ignore WS
"""
```

(continues on next page)

(continued from previous page)

```
parser = Lark(grammar, start='sentence', ambiguity='explicit')

sentence = 'fruit flies like bananas'

def make_png(filename):
    tree.pydot__tree_to_png( parser.parse(sentence), filename)

def make_dot(filename):
    tree.pydot__tree_to_dot( parser.parse(sentence), filename)

if __name__ == '__main__':
    print(parser.parse(sentence).pretty())
    # make_png(sys.argv[1])
    # make_dot(sys.argv[1])

# Output:
#
# _ambig
#   comparative
#     noun  fruit
#     verb  flies
#     noun  bananas
#   simple
#     noun
#       fruit
#       flies
#     verb  like
#     noun  bananas
#
# (or view a nicer version at "./fruitflies.png")
```

Total running time of the script: (0 minutes 0.000 seconds)

8.1.3 Lark Grammar

A reference implementation of the Lark grammar (using LALR(1))

```
import lark
from pathlib import Path

examples_path = Path(__file__).parent
lark_path = Path(lark.__file__).parent

parser = lark.Lark.open(lark_path / 'grammars/lark.lark', rel_to=__file__, parser=
↪"lalr")


grammar_files = [
    examples_path / 'advanced/python2.lark',
    examples_path / 'advanced/python3.lark',
    examples_path / 'relative-imports/multiples.lark',
    examples_path / 'relative-imports/multiple2.lark',
    examples_path / 'relative-imports/multiple3.lark',
    examples_path / 'tests/no_newline_at_end.lark',
```

(continues on next page)

(continued from previous page)

```
    examples_path / 'tests/negative_priority.lark',
    examples_path / 'standalone/json.lark',
    lark_path / 'grammars/common.lark',
    lark_path / 'grammars/lark.lark',
    lark_path / 'grammars/unicode.lark',
    lark_path / 'grammars/python.lark',
]

def test():
    for grammar_file in grammar_files:
        tree = parser.parse(open(grammar_file).read())
    print("All grammars parsed successfully")

if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.1.4 Basic calculator

A simple example of a REPL calculator

This example shows how to write a basic calculator with variables.

```
from lark import Lark, Transformer, v_args


try:
    input = raw_input   # For Python2 compatibility
except NameError:
    pass


calc_grammar = """
    ?start: sum
          | NAME "=" sum    -> assign_var

    ?sum: product
        | sum "+" product   -> add
        | sum "-" product   -> sub

    ?product: atom
        | product "*" atom  -> mul
        | product "/" atom  -> div

    ?atom: NUMBER           -> number
         | "-" atom         -> neg
         | NAME             -> var
         | "(" sum ")"

    %import common.CNAME -> NAME
    %import common.NUMBER
    %import common.WS_INLINE

    %ignore WS_INLINE
"""
```

(continues on next page)

(continued from previous page)

```
@v_args(inline=True)    # Affects the signatures of the methods
class CalculateTree(Transformer):
    from operator import add, sub, mul, truediv as div, neg
    number = float

    def __init__(self):
        self.vars = {}

    def assign_var(self, name, value):
        self.vars[name] = value
        return value

    def var(self, name):
        try:
            return self.vars[name]
        except KeyError:
            raise Exception("Variable not found: %s" % name)


calc_parser = Lark(calc_grammar, parser='lalr', transformer=CalculateTree())
calc = calc_parser.parse


def main():
    while True:
        try:
            s = input('> ')
        except EOFError:
            break
        print(calc(s))


def test():
    print(calc("a = 1+2"))
    print(calc("1+a*-3"))


if __name__ == '__main__':
    # test()
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.1.5 Turtle DSL

Implements a LOGO-like toy language for Python's turtle, with interpreter.

```
try:
    input = raw_input   # For Python2 compatibility
except NameError:
    pass

import turtle
```

(continues on next page)

(continued from previous page)

```
from lark import Lark

turtle_grammar = """
    start: instruction+

    instruction: MOVEMENT NUMBER            -> movement
               | "c" COLOR [COLOR]          -> change_color
               | "fill" code_block          -> fill
               | "repeat" NUMBER code_block -> repeat

    code_block: "{" instruction+ "}"

    MOVEMENT: "f"|"b"|"l"|"r"
    COLOR: LETTER+

    %import common.LETTER
    %import common.INT -> NUMBER
    %import common.WS
    %ignore WS
"""

parser = Lark(turtle_grammar)

def run_instruction(t):
    if t.data == 'change_color':
        turtle.color(*t.children)    # We just pass the color names as-is

    elif t.data == 'movement':
        name, number = t.children
        { 'f': turtle.fd,
          'b': turtle.bk,
          'l': turtle.lt,
          'r': turtle.rt, }[name](int(number))

    elif t.data == 'repeat':
        count, block = t.children
        for i in range(int(count)):
            run_instruction(block)

    elif t.data == 'fill':
        turtle.begin_fill()
        run_instruction(t.children[0])
        turtle.end_fill()

    elif t.data == 'code_block':
        for cmd in t.children:
            run_instruction(cmd)
    else:
        raise SyntaxError('Unknown instruction: %s' % t.data)


def run_turtle(program):
    parse_tree = parser.parse(program)
    for inst in parse_tree.children:
        run_instruction(inst)
```

(continues on next page)

(continued from previous page)

```
def main():
    while True:
        code = input('> ')
        try:
            run_turtle(code)
        except Exception as e:
            print(e)

def test():
    text = """
        c red yellow
        fill { repeat 36 {
            f200 l170
        }}
    """
    run_turtle(text)

if __name__ == '__main__':
    # test()
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.1.6 Simple JSON Parser

The code is short and clear, and outperforms every other parser (that's written in Python). For an explanation, check out the JSON parser tutorial at /docs/json_tutorial.md

```
import sys

from lark import Lark, Transformer, v_args

json_grammar = r"""
    ?start: value

    ?value: object
          | array
          | string
          | SIGNED_NUMBER      -> number
          | "true"             -> true
          | "false"            -> false
          | "null"             -> null

    array  : "[" [value ("," value)*] "]"
    object : "{" [pair ("," pair)*] "}"
    pair   : string ":" value

    string : ESCAPED_STRING

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
    %import common.WS

    %ignore WS
"""
```

(continues on next page)

(continued from previous page)

```
class TreeToJson(Transformer):
    @v_args(inline=True)
    def string(self, s):
        return s[1:-1].replace('\\"', '"')

    array = list
    pair = tuple
    object = dict
    number = v_args(inline=True)(float)

    null = lambda self, _: None
    true = lambda self, _: True
    false = lambda self, _: False


### Create the JSON parser with Lark, using the Earley algorithm
# json_parser = Lark(json_grammar, parser='earley', lexer='standard')
# def parse(x):
#     return TreeToJson().transform(json_parser.parse(x))

### Create the JSON parser with Lark, using the LALR algorithm
json_parser = Lark(json_grammar, parser='lalr',
                   # Using the standard lexer isn't required, and isn't usually␣
↪recommended.
                   # But, it's good enough for JSON, and it's slightly faster.
                   lexer='standard',
                   # Disabling propagate_positions and placeholders slightly improves␣
↪speed
                   propagate_positions=False,
                   maybe_placeholders=False,
                   # Using an internal transformer is faster and more memory efficient
                   transformer=TreeToJson())
parse = json_parser.parse


def test():
    test_json = '''
        {
            "empty_object" : {},
            "empty_array"  : [],
            "booleans"     : { "YES" : true, "NO" : false },
            "numbers"      : [ 0, 1, -2, 3.3, 4.4e5, 6.6e-7 ],
            "strings"      : [ "This", [ "And" , "That", "And a \\"b" ] ],
            "nothing"      : null
        }
    '''

    j = parse(test_json)
    print(j)
    import json
    assert j == json.loads(test_json)


if __name__ == '__main__':
    # test()
```

(continues on next page)

(continued from previous page)

```
    with open(sys.argv[1]) as f:
        print(parse(f.read()))
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2 Advanced Examples

8.2.1 LALR's contextual lexer

Demonstrates the power of LALR's contextual lexer on a toy configuration language.

The tokens NAME and VALUE match the same input. A standard lexer would arbitrarily choose one over the other, which would lead to a (confusing) parse error. However, due to the unambiguous structure of the grammar, Lark's LALR(1) algorithm knows which one of them to expect at each point during the parse. The lexer then only matches the tokens that the parser expects. The result is a correct parse, something that is impossible with a regular lexer.

Another approach is to discard a lexer altogether and use the Earley algorithm. It will handle more cases than the contextual lexer, but at the cost of performance. See examples/conf_earley.py for an example of that approach.

```
from lark import Lark

parser = Lark(r"""
        start: _NL? section+
        section: "[" NAME "]" _NL item+
        item: NAME "=" VALUE? _NL
        VALUE: /./+

        %import common.CNAME -> NAME
        %import common.NEWLINE -> _NL
        %import common.WS_INLINE
        %ignore WS_INLINE
    """, parser="lalr")


sample_conf = """
[bla]
a=Hello
this="that",4
empty=
"""

print(parser.parse(sample_conf).pretty())
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.2 Templates

This example shows how to use Lark's templates to achieve cleaner grammars

```
from lark import Lark

grammar = r"""
start: list | dict
```

(continues on next page)

(continued from previous page)

```
list: "[" _seperated{atom, ","} "]"
dict: "{" _seperated{key_value, ","} "}"
key_value: atom ":" atom

_seperated{x, sep}: x (sep x)*  // Define a sequence of 'x sep x sep x ...'

atom: NUMBER | ESCAPED_STRING

%import common (NUMBER, ESCAPED_STRING, WS)
%ignore WS
"""


parser = Lark(grammar)

print(parser.parse('[1, "a", 2]'))
print(parser.parse('{"a": 2, "b": 6}'))
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.3 Earley's dynamic lexer

Demonstrates the power of Earley's dynamic lexer on a toy configuration language

Using a lexer for configuration files is tricky, because values don't have to be surrounded by delimiters. Using a standard lexer for this just won't work.

In this example we use a dynamic lexer and let the Earley parser resolve the ambiguity.

Another approach is to use the contextual lexer with LALR. It is less powerful than Earley, but it can handle some ambiguity when lexing and it's much faster. See examples/conf_lalr.py for an example of that approach.

```
from lark import Lark

parser = Lark(r"""
        start: _NL? section+
        section: "[" NAME "]" _NL item+
        item: NAME "=" VALUE? _NL
        VALUE: /./+

        %import common.CNAME -> NAME
        %import common.NEWLINE -> _NL
        %import common.WS_INLINE
        %ignore WS_INLINE
    """, parser="earley")

def test():
    sample_conf = """
[bla]

a=Hello
this="that",4
empty=
"""
```

(continues on next page)

(continued from previous page)

```
    r = parser.parse(sample_conf)
    print (r.pretty())

if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.4 Error handling with a puppet

This example demonstrates error handling using a parsing puppet in LALR

When the parser encounters an UnexpectedToken exception, it creates a parsing puppet with the current parse-state, and lets you control how to proceed step-by-step. When you've achieved the correct parse-state, you can resume the run by returning True.

```
from lark import Token

from _json_parser import json_parser

def ignore_errors(e):
    if e.token.type == 'COMMA':
        # Skip comma
        return True
    elif e.token.type == 'SIGNED_NUMBER':
        # Try to feed a comma and retry the number
        e.puppet.feed_token(Token('COMMA', ','))
        e.puppet.feed_token(e.token)
        return True

    # Unhandled error. Will stop parse and raise exception
    return False


def main():
    s = "[0 1, 2,, 3,,, 4, 5 6 ]"
    res = json_parser.parse(s, on_error=ignore_errors)
    print(res)      # prints [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]

main()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.5 Reconstruct a JSON

Demonstrates the experimental text-reconstruction feature

The Reconstructor takes a parse tree (already filtered from punctuation, of course), and reconstructs it into correct text, that can be parsed correctly. It can be useful for creating "hooks" to alter data before handing it to other parsers. You can also use it to generate samples from scratch.

```
import json

from lark import Lark
```

(continues on next page)

(continued from previous page)

```
from lark.reconstruct import Reconstructor

from _json_parser import json_grammar

test_json = '''
    {
        "empty_object" : {},
        "empty_array"  : [],
        "booleans"     : { "YES" : true, "NO" : false },
        "numbers"      : [ 0, 1, -2, 3.3, 4.4e5, 6.6e-7 ],
        "strings"      : [ "This", [ "And" , "That", "And a \\"b" ] ],
        "nothing"      : null
    }
'''

def test_earley():

    json_parser = Lark(json_grammar, maybe_placeholders=False)
    tree = json_parser.parse(test_json)

    new_json = Reconstructor(json_parser).reconstruct(tree)
    print (new_json)
    print (json.loads(new_json) == json.loads(test_json))


def test_lalr():

    json_parser = Lark(json_grammar, parser='lalr', maybe_placeholders=False)
    tree = json_parser.parse(test_json)

    new_json = Reconstructor(json_parser).reconstruct(tree)
    print (new_json)
    print (json.loads(new_json) == json.loads(test_json))

test_earley()
test_lalr()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.6 Custom lexer

Demonstrates using a custom lexer to parse a non-textual stream of data

You can use a custom lexer to tokenize text when the lexers offered by Lark are too slow, or not flexible enough.

You can also use it (as shown in this example) to tokenize streams of objects.

```
from lark import Lark, Transformer, v_args
from lark.lexer import Lexer, Token

class TypeLexer(Lexer):
    def __init__(self, lexer_conf):
        pass

    def lex(self, data):
        for obj in data:
```

(continues on next page)

(continued from previous page)

```
            if isinstance(obj, int):
                yield Token('INT', obj)
            elif isinstance(obj, (type(''), type(u''))):
                yield Token('STR', obj)
            else:
                raise TypeError(obj)

parser = Lark("""
        start: data_item+
        data_item: STR INT*

        %declare STR INT
        """, parser='lalr', lexer=TypeLexer)


class ParseToDict(Transformer):
    @v_args(inline=True)
    def data_item(self, name, *numbers):
        return name.value, [n.value for n in numbers]

    start = dict


def test():
    data = ['alice', 1, 27, 3, 'bob', 4, 'carrie', 'dan', 8, 6]

    print(data)

    tree = parser.parse(data)
    res = ParseToDict().transform(tree)

    print('-->')
    print(res) # prints {'alice': [1, 27, 3], 'bob': [4], 'carrie': [], 'dan': [8, 6]}


if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.7 Transform a Forest

This example demonstrates how to subclass `TreeForestTransformer` to directly transform a SPPF.

```
from lark import Lark
from lark.parsers.earley_forest import TreeForestTransformer, handles_ambiguity,␣
↪Discard

class CustomTransformer(TreeForestTransformer):

    @handles_ambiguity
    def sentence(self, trees):
        return next(tree for tree in trees if tree.data == 'simple')

    def simple(self, children):
```

(continues on next page)

(continued from previous page)

```
        children.append('.')
        return self.tree_class('simple', children)

    def adj(self, children):
        raise Discard()

    def __default_token__(self, token):
        return token.capitalize()

grammar = """
    sentence: noun verb noun        -> simple
            | noun verb "like" noun -> comparative

    noun: adj? NOUN
    verb: VERB
    adj: ADJ

    NOUN: "flies" | "bananas" | "fruit"
    VERB: "like" | "flies"
    ADJ: "fruit"

    %import common.WS
    %ignore WS
"""

parser = Lark(grammar, start='sentence', ambiguity='forest')
sentence = 'fruit flies like bananas'
forest = parser.parse(sentence)

tree = CustomTransformer(resolve_ambiguity=False).transform(forest)
print(tree.pretty())

# Output:
#
# simple
#   noun  Flies
#   verb  Like
#   noun  Bananas
#   .
#
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.8 Simple JSON Parser

The code is short and clear, and outperforms every other parser (that's written in Python). For an explanation, check out the JSON parser tutorial at /docs/json_tutorial.md

(this is here for use by the other examples)

```
import sys

from lark import Lark, Transformer, v_args

json_grammar = r"""
    ?start: value
```

(continues on next page)

(continued from previous page)

```
    ?value: object
          | array
          | string
          | SIGNED_NUMBER      -> number
          | "true"             -> true
          | "false"            -> false
          | "null"             -> null

    array  : "[" [value ("," value)*] "]"
    object : "{" [pair ("," pair)*] "}"
    pair   : string ":" value

    string : ESCAPED_STRING

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
    %import common.WS

    %ignore WS
"""


class TreeToJson(Transformer):
    @v_args(inline=True)
    def string(self, s):
        return s[1:-1].replace('\\"', '"')

    array = list
    pair = tuple
    object = dict
    number = v_args(inline=True)(float)

    null = lambda self, _: None
    true = lambda self, _: True
    false = lambda self, _: False


### Create the JSON parser with Lark, using the LALR algorithm
json_parser = Lark(json_grammar, parser='lalr',
                   # Using the standard lexer isn't required, and isn't usually␣
↪recommended.
                   # But, it's good enough for JSON, and it's slightly faster.
                   lexer='standard',
                   # Disabling propagate_positions and placeholders slightly improves␣
↪speed
                   propagate_positions=False,
                   maybe_placeholders=False,
                   # Using an internal transformer is faster and more memory efficient
                   transformer=TreeToJson())
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.9 Custom SPPF Prioritizer

This example demonstrates how to subclass `ForestVisitor` to make a custom SPPF node prioritizer to be used in conjunction with `TreeForestTransformer`.

Our prioritizer will count the number of descendants of a node that are tokens. By negating this count, our prioritizer will prefer nodes with fewer token descendants. Thus, we choose the more specific parse.

```
from lark import Lark
from lark.parsers.earley_forest import ForestVisitor, TreeForestTransformer

class TokenPrioritizer(ForestVisitor):

    def visit_symbol_node_in(self, node):
        # visit the entire forest by returning node.children
        return node.children

    def visit_packed_node_in(self, node):
        return node.children

    def visit_symbol_node_out(self, node):
        priority = 0
        for child in node.children:
            # Tokens do not have a priority attribute
            # count them as -1
            priority += getattr(child, 'priority', -1)
        node.priority = priority

    def visit_packed_node_out(self, node):
        priority = 0
        for child in node.children:
            priority += getattr(child, 'priority', -1)
        node.priority = priority

    def on_cycle(self, node, path):
        raise Exception("Oops, we encountered a cycle.")

grammar = """
start: hello " " world | hello_world
hello: "Hello"
world: "World"
hello_world: "Hello World"
"""

parser = Lark(grammar, parser='earley', ambiguity='forest')
forest = parser.parse("Hello World")

print("Default prioritizer:")
tree = TreeForestTransformer(resolve_ambiguity=True).transform(forest)
print(tree.pretty())

forest = parser.parse("Hello World")

print("Custom prioritizer:")
tree = TreeForestTransformer(resolve_ambiguity=True, prioritizer=TokenPrioritizer()).
↪transform(forest)
print(tree.pretty())
```

(continues on next page)

(continued from previous page)

```
# Output:
#
# Default prioritizer:
# start
#   hello Hello
#
#   world World
#
# Custom prioritizer:
# start
#   hello_world   Hello World
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.10 Compile Python to Bytecode

A toy example that compiles Python directly to bytecode, without generating an AST. It currently only works for very very simple Python code.

It requires the 'bytecode' library. You can get it using

```
$ pip install bytecode
```

```
from lark import Lark, Transformer, v_args
from lark.indenter import Indenter

from bytecode import Instr, Bytecode

class PythonIndenter(Indenter):
    NL_type = '_NEWLINE'
    OPEN_PAREN_types = ['LPAR', 'LSQB', 'LBRACE']
    CLOSE_PAREN_types = ['RPAR', 'RSQB', 'RBRACE']
    INDENT_type = '_INDENT'
    DEDENT_type = '_DEDENT'
    tab_len = 8


@v_args(inline=True)
class Compile(Transformer):
    def number(self, n):
        return [Instr('LOAD_CONST', int(n))]
    def string(self, s):
        return [Instr('LOAD_CONST', s[1:-1])]
    def var(self, n):
        return [Instr('LOAD_NAME', n)]

    def arith_expr(self, a, op, b):
        # TODO support chain arithmetic
        assert op == '+'
        return a + b + [Instr('BINARY_ADD')]

    def arguments(self, args):
        return args

    def funccall(self, name, args):
```

(continues on next page)

(continued from previous page)

```
        return name + args + [Instr('CALL_FUNCTION', 1)]

    @v_args(inline=False)
    def file_input(self, stmts):
        return sum(stmts, []) + [Instr("RETURN_VALUE")]

    def expr_stmt(self, lval, rval):
        # TODO more complicated than that
        name ,= lval
        assert name.name == 'LOAD_NAME' # XXX avoid with another layer of abstraction
        return rval + [Instr("STORE_NAME", name.arg)]

    def __default__(self, *args):
        assert False, args


python_parser3 = Lark.open('python3.lark', rel_to=__file__, start='file_input',
                           parser='lalr', postlex=PythonIndenter(),
                           transformer=Compile(), propagate_positions=False)

def compile_python(s):
    insts = python_parser3.parse(s+"\n")
    return Bytecode(insts).to_code()

code = compile_python("""
a = 3
b = 5
print("Hello World!")
print(a+(b+2))
print((a+b)+2)
""")
exec(code)
# -- Output --
# Hello World!
# 10
# 10
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.11 Grammar-complete Python Parser

A fully-working Python 2 & 3 parser (but not production ready yet!)

This example demonstrates usage of the included Python grammars

```
import sys
import os, os.path
from io import open
import glob, time

from lark import Lark
from lark.indenter import Indenter

# __path__ = os.path.dirname(__file__)

class PythonIndenter(Indenter):
```

(continues on next page)

(continued from previous page)

```
    NL_type = '_NEWLINE'
    OPEN_PAREN_types = ['LPAR', 'LSQB', 'LBRACE']
    CLOSE_PAREN_types = ['RPAR', 'RSQB', 'RBRACE']
    INDENT_type = '_INDENT'
    DEDENT_type = '_DEDENT'
    tab_len = 8

kwargs = dict(rel_to=__file__, postlex=PythonIndenter(), start='file_input')

python_parser2 = Lark.open('python2.lark', parser='lalr', **kwargs)
python_parser3 = Lark.open('python3.lark',parser='lalr', **kwargs)
python_parser2_earley = Lark.open('python2.lark', parser='earley', lexer='standard',␣
↪**kwargs)

try:
    xrange
except NameError:
    chosen_parser = python_parser3
else:
    chosen_parser = python_parser2


def _read(fn, *args):
    kwargs = {'encoding': 'iso-8859-1'}
    with open(fn, *args, **kwargs) as f:
        return f.read()

def _get_lib_path():
    if os.name == 'nt':
        if 'PyPy' in sys.version:
            return os.path.join(sys.prefix, 'lib-python', sys.winver)
        else:
            return os.path.join(sys.prefix, 'Lib')
    else:
        return [x for x in sys.path if x.endswith('%s.%s' % sys.version_info[:2])][0]

def test_python_lib():
    path = _get_lib_path()

    start = time.time()
    files = glob.glob(path+'/*.py')
    for f in files:
        print( f )
        chosen_parser.parse(_read(os.path.join(path, f)) + '\n')

    end = time.time()
    print( "test_python_lib (%d files), time: %s secs"%(len(files), end-start) )

def test_earley_equals_lalr():
    path = _get_lib_path()

    files = glob.glob(path+'/*.py')
    for f in files:
        print( f )
        tree1 = python_parser2.parse(_read(os.path.join(path, f)) + '\n')
        tree2 = python_parser2_earley.parse(_read(os.path.join(path, f)) + '\n')
        assert tree1 == tree2
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    test_python_lib()
    # test_earley_equals_lalr()
    # python_parser3.parse(_read(sys.argv[1]) + '\n')
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.12 Example-Driven Error Reporting

A demonstration of example-driven error reporting with the Earley parser (See also: error_reporting_lalr.py)

```
from lark import Lark, UnexpectedInput

from _json_parser import json_grammar   # Using the grammar from the json_parser␣
↪example

json_parser = Lark(json_grammar)

class JsonSyntaxError(SyntaxError):
    def __str__(self):
        context, line, column = self.args
        return '%s at line %s, column %s.\n\n%s' % (self.label, line, column, context)

class JsonMissingValue(JsonSyntaxError):
    label = 'Missing Value'

class JsonMissingOpening(JsonSyntaxError):
    label = 'Missing Opening'

class JsonMissingClosing(JsonSyntaxError):
    label = 'Missing Closing'

class JsonMissingComma(JsonSyntaxError):
    label = 'Missing Comma'

class JsonTrailingComma(JsonSyntaxError):
    label = 'Trailing Comma'


def parse(json_text):
    try:
        j = json_parser.parse(json_text)
    except UnexpectedInput as u:
        exc_class = u.match_examples(json_parser.parse, {
            JsonMissingOpening: ['{"foo": ]}',
                                 '{"foor": }}',
                                 '{"foo": }'],
            JsonMissingClosing: ['{"foo": [}',
                                 '{',
                                 '{"a": 1',
                                 '[1'],
            JsonMissingComma: ['[1 2]',
                               '[false 1]',
```

(continues on next page)

(continued from previous page)

```
                                '["b" 1]',
                                '{"a":true 1:4}',
                                '{"a":1 1:4}',
                                '{"a":"b" 1:4}'],
            JsonTrailingComma: ['[,]',
                                '[1,]',
                                '[1,2,]',
                                '{"foo":1,}',
                                '{"foo":false,"bar":true,}']
        }, use_accepts=True)
        if not exc_class:
            raise
        raise exc_class(u.get_context(json_text), u.line, u.column)


def test():
    try:
        parse('{"example1": "value"')
    except JsonMissingClosing as e:
        print(e)

    try:
        parse('{"example2": ] ')
    except JsonMissingOpening as e:
        print(e)


if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.13 Example-Driven Error Reporting

A demonstration of example-driven error reporting with the LALR parser (See also: error_reporting_earley.py)

```
from lark import Lark, UnexpectedInput

from _json_parser import json_grammar   # Using the grammar from the json_parser␣
↪example

json_parser = Lark(json_grammar, parser='lalr')

class JsonSyntaxError(SyntaxError):
    def __str__(self):
        context, line, column = self.args
        return '%s at line %s, column %s.\n\n%s' % (self.label, line, column, context)

class JsonMissingValue(JsonSyntaxError):
    label = 'Missing Value'

class JsonMissingOpening(JsonSyntaxError):
    label = 'Missing Opening'

class JsonMissingClosing(JsonSyntaxError):
```

(continues on next page)

(continued from previous page)

```
    label = 'Missing Closing'

class JsonMissingComma(JsonSyntaxError):
    label = 'Missing Comma'

class JsonTrailingComma(JsonSyntaxError):
    label = 'Trailing Comma'


def parse(json_text):
    try:
        j = json_parser.parse(json_text)
    except UnexpectedInput as u:
        exc_class = u.match_examples(json_parser.parse, {
            JsonMissingOpening: ['{"foo": ]}',
                                 '{"foor": }}',
                                 '{"foo": }'],
            JsonMissingClosing: ['{"foo": [}',
                                 '{',
                                 '{"a": 1',
                                 '[1'],
            JsonMissingComma: ['[1 2]',
                               '[false 1]',
                               '["b" 1]',
                               '{"a":true 1:4}',
                               '{"a":1 1:4}',
                               '{"a":"b" 1:4}'],
            JsonTrailingComma: ['[,]',
                                '[1,]',
                                '[1,2,]',
                                '{"foo":1,}',
                                '{"foo":false,"bar":true,}']
        }, use_accepts=True)
        if not exc_class:
            raise
        raise exc_class(u.get_context(json_text), u.line, u.column)


def test():
    try:
        parse('{"example1": "value"')
    except JsonMissingClosing as e:
        print(e)

    try:
        parse('{"example2": ] ')
    except JsonMissingOpening as e:
        print(e)


if __name__ == '__main__':
    test()
```

Total running time of the script: (0 minutes 0.000 seconds)

8.2.14 Syntax Highlighting

This example shows how to write a syntax-highlighted editor with Qt and Lark

Requirements:

PyQt5==5.10.1 QScintilla==2.10.4

```
import sys
import textwrap

from PyQt5.Qt import *  # noqa

from PyQt5.Qsci import QsciScintilla
from PyQt5.Qsci import QsciLexerCustom

from lark import Lark


class LexerJson(QsciLexerCustom):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.create_parser()
        self.create_styles()

    def create_styles(self):
        deeppink = QColor(249, 38, 114)
        khaki = QColor(230, 219, 116)
        mediumpurple = QColor(174, 129, 255)
        mediumturquoise = QColor(81, 217, 205)
        yellowgreen = QColor(166, 226, 46)
        lightcyan = QColor(213, 248, 232)
        darkslategrey = QColor(39, 40, 34)

        styles = {
            0: mediumturquoise,
            1: mediumpurple,
            2: yellowgreen,
            3: deeppink,
            4: khaki,
            5: lightcyan
        }

        for style, color in styles.items():
            self.setColor(color, style)
            self.setPaper(darkslategrey, style)
            self.setFont(self.parent().font(), style)

        self.token_styles = {
            "COLON": 5,
            "COMMA": 5,
            "LBRACE": 5,
            "LSQB": 5,
            "RBRACE": 5,
            "RSQB": 5,
            "FALSE": 0,
            "NULL": 0,
            "TRUE": 0,
```

(continues on next page)

(continued from previous page)

```
            "STRING": 4,
            "NUMBER": 1,
        }

    def create_parser(self):
        grammar = '''
            anons: ":" "{" "}" "," "[" "]"
            TRUE: "true"
            FALSE: "false"
            NULL: "NULL"
            %import common.ESCAPED_STRING -> STRING
            %import common.SIGNED_NUMBER  -> NUMBER
            %import common.WS
            %ignore WS
        '''

        self.lark = Lark(grammar, parser=None, lexer='standard')
        # All tokens: print([t.name for t in self.lark.parser.lexer.tokens])

    def defaultPaper(self, style):
        return QColor(39, 40, 34)

    def language(self):
        return "Json"

    def description(self, style):
        return {v: k for k, v in self.token_styles.items()}.get(style, "")

    def styleText(self, start, end):
        self.startStyling(start)
        text = self.parent().text()[start:end]
        last_pos = 0

        try:
            for token in self.lark.lex(text):
                ws_len = token.pos_in_stream - last_pos
                if ws_len:
                    self.setStyling(ws_len, 0)    # whitespace

                token_len = len(bytearray(token, "utf-8"))
                self.setStyling(
                    token_len, self.token_styles.get(token.type, 0))

                last_pos = token.pos_in_stream + token_len
        except Exception as e:
            print(e)


class EditorAll(QsciScintilla):

    def __init__(self, parent=None):
        super().__init__(parent)

        # Set font defaults
        font = QFont()
        font.setFamily('Consolas')
        font.setFixedPitch(True)
```

(continues on next page)

(continued from previous page)

```
        font.setPointSize(8)
        font.setBold(True)
        self.setFont(font)

        # Set margin defaults
        fontmetrics = QFontMetrics(font)
        self.setMarginsFont(font)
        self.setMarginWidth(0, fontmetrics.width("000") + 6)
        self.setMarginLineNumbers(0, True)
        self.setMarginsForegroundColor(QColor(128, 128, 128))
        self.setMarginsBackgroundColor(QColor(39, 40, 34))
        self.setMarginType(1, self.SymbolMargin)
        self.setMarginWidth(1, 12)

        # Set indentation defaults
        self.setIndentationsUseTabs(False)
        self.setIndentationWidth(4)
        self.setBackspaceUnindents(True)
        self.setIndentationGuides(True)

        # self.setFolding(QsciScintilla.CircledFoldStyle)

        # Set caret defaults
        self.setCaretForegroundColor(QColor(247, 247, 241))
        self.setCaretWidth(2)

        # Set selection color defaults
        self.setSelectionBackgroundColor(QColor(61, 61, 52))
        self.resetSelectionForegroundColor()

        # Set multiselection defaults
        self.SendScintilla(QsciScintilla.SCI_SETMULTIPLESELECTION, True)
        self.SendScintilla(QsciScintilla.SCI_SETMULTIPASTE, 1)
        self.SendScintilla(
            QsciScintilla.SCI_SETADDITIONALSELECTIONTYPING, True)

        lexer = LexerJson(self)
        self.setLexer(lexer)


EXAMPLE_TEXT = textwrap.dedent("""\
        {
            "_id": "5b05ffcbcf8e597939b3f5ca",
            "about": "Excepteur consequat commodo esse voluptate aute aliquip ad sint␣
↪deserunt commodo eiusmod irure. Sint aliquip sit magna duis eu est culpa aliqua␣
↪excepteur ut tempor nulla. Aliqua ex pariatur id labore sit. Quis sit ex aliqua␣
↪veniam exercitation laboris anim adipisicing. Lorem nisi reprehenderit ullamco␣
↪labore qui sit ut aliqua tempor consequat pariatur proident.",
            "address": "665 Malbone Street, Thornport, Louisiana, 243",
            "age": 23,
            "balance": "$3,216.91",
            "company": "BULLJUICE",
            "email": "elisekelley@bulljuice.com",
            "eyeColor": "brown",
            "gender": "female",
            "guid": "d3a6d865-0f64-4042-8a78-4f53de9b0707",
            "index": 0,
```

(continues on next page)

(continued from previous page)

```
            "isActive": false,
            "isActive2": true,
            "latitude": -18.660714,
            "longitude": -85.378048,
            "name": "Elise Kelley",
            "phone": "+1 (808) 543-3966",
            "picture": "http://placehold.it/32x32",
            "registered": "2017-09-30T03:47:40 -02:00",
            "tags": [
                "et",
                "nostrud",
                "in",
                "fugiat",
                "incididunt",
                "labore",
                "nostrud"
            ]
        }\
    """)

def main():
    app = QApplication(sys.argv)
    ex = EditorAll()
    ex.setWindowTitle(__file__)
    ex.setText(EXAMPLE_TEXT)
    ex.resize(800, 600)
    ex.show()
    sys.exit(app.exec_())


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 9

Grammar Reference

9.1 Definitions

A **grammar** is a list of rules and terminals, that together define a language.

Terminals define the alphabet of the language, while rules define its structure.

In Lark, a terminal may be a string, a regular expression, or a concatenation of these and other terminals.

Each rule is a list of terminals and rules, whose location and nesting define the structure of the resulting parse-tree.

A **parsing algorithm** is an algorithm that takes a grammar definition and a sequence of symbols (members of the alphabet), and matches the entirety of the sequence by searching for a structure that is allowed by the grammar.

9.1.1 General Syntax and notes

Grammars in Lark are based on EBNF syntax, with several enhancements.

EBNF is basically a short-hand for common BNF patterns.

Optionals are expanded:

```
a b? c    ->    (a c | a b c)
```

Repetition is extracted into a recursion:

```
a: b*    ->    a: _b_tag
              _b_tag: (_b_tag b)?
```

And so on.

Lark grammars are composed of a list of definitions and directives, each on its own line. A definition is either a named rule, or a named terminal, with the following syntax, respectively:

```
  rule: <EBNF EXPRESSION>
      | etc.

  TERM: <EBNF EXPRESSION>   // Rules aren't allowed
```

Comments start with `//` and last to the end of the line (C++ style)

Lark begins the parse with the rule 'start', unless specified otherwise in the options.

Names of rules are always in lowercase, while names of terminals are always in uppercase. This distinction has practical effects, for the shape of the generated parse-tree, and the automatic construction of the lexer (aka tokenizer, or scanner).

9.2 Terminals

Terminals are used to match text into symbols. They can be defined as a combination of literals and other terminals.

Syntax:

```
<NAME> [. <priority>] : <literals-and-or-terminals>
```

Terminal names must be uppercase.

Literals can be one of:

- `"string"`
- `/regular expression+/`
- `"case-insensitive string"i`
- `/re with flags/imulx`
- Literal range: `"a".."z"`, `"1".."9"`, etc.

Terminals also support grammar operators, such as `|`, `+`, `*` and `?`.

Terminals are a linear construct, and therefore may not contain themselves (recursion isn't allowed).

9.2.1 Templates

Templates are expanded when preprocessing the grammar.

Definition syntax:

```
  my_template{param1, param2, ...}: <EBNF EXPRESSION>
```

Use syntax:

```
some_rule: my_template{arg1, arg2, ...}
```

Example:

```
_separated{x, sep}: x (sep x)*  // Define a sequence of 'x sep x sep x ...'

num_list: "[" _separated{NUMBER, ","} "]"   // Will match "[1, 2, 3]" etc.
```

9.2.2 Priority

Terminals can be assigned priority only when using a lexer (future versions may support Earley's dynamic lexing).

Priority can be either positive or negative. If not specified for a terminal, it defaults to 1.

Highest priority terminals are always matched first.

9.2.3 Regexp Flags

You can use flags on regexps and strings. For example:

```
SELECT: "select"i     //# Will ignore case, and match SELECT or Select, etc.
MULTILINE_TEXT: /.+/s
SIGNED_INTEGER: /
    [+-]?  # the sign
    (0|[1-9][0-9]*)  # the digits
 /x
```

Supported flags are one of: `imslux`. See Python's regex documentation for more details on each one.

Regexps/strings of different flags can only be concatenated in Python 3.6+

Notes for when using a lexer:

When using a lexer (standard or contextual), it is the grammar-author's responsibility to make sure the literals don't collide, or that if they do, they are matched in the desired order. Literals are matched according to the following precedence:

1. Highest priority first (priority is specified as: TERM.number: ...)
2. Length of match (for regexps, the longest theoretical match is used)
3. Length of literal / pattern definition
4. Name

Examples:

```
IF: "if"
INTEGER : /[0-9]+/
INTEGER2 : ("0".."9")+           //# Same as INTEGER
DECIMAL.2: INTEGER? "." INTEGER  //# Will be matched before INTEGER
WHITESPACE: (" " | /\t/ )+
SQL_SELECT: "select"i
```

9.2.4 Regular expressions & Ambiguity

Each terminal is eventually compiled to a regular expression. All the operators and references inside it are mapped to their respective expressions.

For example, in the following grammar, `A1` and `A2`, are equivalent:

```
A1: "a" | "b"
A2: /a|b/
```

This means that inside terminals, Lark cannot detect or resolve ambiguity, even when using Earley.

For example, for this grammar:

```
start           : (A | B)+
A               : "a" | "ab"
B               : "b"
```

We get this behavior:

```
>>> p.parse("ab")
Tree(start, [Token(A, 'a'), Token(B, 'b')])
```

This is happening because Python's regex engine always returns the first matching option.

If you find yourself in this situation, the recommended solution is to use rules instead.

Example:

```
>>> p = Lark("""start: (a | b)+
...              !a: "a" | "ab"
...              !b: "b"
...              """, ambiguity="explicit")
>>> print(p.parse("ab").pretty())
_ambig
  start
    a   ab
  start
    a   a
    b   b
```

9.3 Rules

Syntax:

```
<name> : <items-to-match>  [-> <alias> ]
       | ...
```

Names of rules and aliases are always in lowercase.

Rule definitions can be extended to the next line by using the OR operator (signified by a pipe: `|`).

An alias is a name for the specific rule alternative. It affects tree construction.

Each item is one of:

- `rule`
- `TERMINAL`
- `"string literal"` or `/regexp literal/`
- `(item item ..)` - Group items
- `[item item ..]` - Maybe. Same as `(item item ..)?`, but when `maybe_placeholders=True`, generates `None` if there is no match.
- `item?` - Zero or one instances of item ("maybe")
- `item*` - Zero or more instances of item

- `item+` - One or more instances of item
- `item ~ n` - Exactly *n* instances of item
- `item ~ n..m` - Between *n* to *m* instances of item (not recommended for wide ranges, due to performance issues)

Examples:

```
hello_world: "hello" "world"
mul: (mul "*")? number     //# Left-recursion is allowed and encouraged!
expr: expr operator expr
    | value                //# Multi-line, belongs to expr

four_words: word ~ 4
```

9.3.1 Priority

Rules can be assigned priority only when using Earley (future versions may support LALR as well).

Priority can be either positive or negative. In not specified for a terminal, it's assumed to be 1 (i.e. the default).

9.4 Directives

9.4.1 %ignore

All occurrences of the terminal will be ignored, and won't be part of the parse.

Using the `%ignore` directive results in a cleaner grammar.

It's especially important for the LALR(1) algorithm, because adding whitespace (or comments, or other extraneous elements) explicitly in the grammar, harms its predictive abilities, which are based on a lookahead of 1.

Syntax:

```
%ignore <TERMINAL>
```

Examples:

```
%ignore " "

COMMENT: "#" /[^\n]/*
%ignore COMMENT
```

9.4.2 %import

Allows one to import terminals and rules from lark grammars.

When importing rules, all their dependencies will be imported into a namespace, to avoid collisions. It's not possible to override their dependencies (e.g. like you would when inheriting a class).

Syntax:

```
%import <module>.<TERMINAL>
%import <module>.<rule>
%import <module>.<TERMINAL> -> <NEWTERMINAL>
%import <module>.<rule> -> <newrule>
%import <module> (<TERM1>, <TERM2>, <rule1>, <rule2>)
```

If the module path is absolute, Lark will attempt to load it from the built-in directory (which currently contains `common.lark`, `python.lark`, and `unicode.lark`).

If the module path is relative, such as `.path.to.file`, Lark will attempt to load it from the current working directory. Grammars must have the `.lark` extension.

The rule or terminal can be imported under another name with the `->` syntax.

Example:

```
%import common.NUMBER

%import .terminals_file (A, B, C)

%import .rules_file.rulea -> ruleb
```

Note that `%ignore` directives cannot be imported. Imported rules will abide by the `%ignore` directives declared in the main grammar.

9.4.3 %declare

Declare a terminal without defining it. Useful for plugins.

CHAPTER 10

Tree Construction Reference

Lark builds a tree automatically based on the structure of the grammar, where each rule that is matched becomes a branch (node) in the tree, and its children are its matches, in the order of matching.

For example, the rule `node: child1 child2` will create a tree node with two children. If it is matched as part of another rule (i.e. if it isn't the root), the new rule's tree node will become its parent.

Using `item+` or `item*` will result in a list of items, equivalent to writing `item item item ...`

Using `item?` will return the item if it matched, or nothing.

If `maybe_placeholders=False` (the default), then `[]` behaves like `()?`.

If `maybe_placeholders=True`, then using `[item]` will return the item if it matched, or the value `None`, if it didn't.

10.1 Terminals

Terminals are always values in the tree, never branches.

Lark filters out certain types of terminals by default, considering them punctuation:

- Terminals that won't appear in the tree are:
  - Unnamed literals (like `"keyword"` or `"+"`)
  - Terminals whose name starts with an underscore (like `_DIGIT`)
- Terminals that *will* appear in the tree are:
  - Unnamed regular expressions (like `/[0-9]/`)
  - Named terminals whose name starts with a letter (like `DIGIT`)

Note: Terminals composed of literals and other terminals always include the entire match without filtering any part.

Example:

```
start:  PNAME pname

PNAME:  "(" NAME ")"
pname:  "(" NAME ")"

NAME:   /\w+/
%ignore /\s+/
```

Lark will parse "(Hello) (World)" as:

```
start
    (Hello)
    pname World
```

Rules prefixed with `!` will retain all their literals regardless.

Example:

```
    expr: "(" expr ")"
        | NAME+

    NAME: /\w+/

    %ignore " "
```

Lark will parse "((hello world))" as:

```
expr
    expr
        expr
            "hello"
            "world"
```

The brackets do not appear in the tree by design. The words appear because they are matched by a named terminal.

10.2 Shaping the tree

Users can alter the automatic construction of the tree using a collection of grammar features.

- Rules whose name begins with an underscore will be inlined into their containing rule.

Example:

```
    start: "(" _greet ")"
    _greet: /\w+/ /\w+/
```

Lark will parse "(hello world)" as:

```
start
    "hello"
    "world"
```

- Rules that receive a question mark (?) at the beginning of their definition, will be inlined if they have a single child, after filtering.

Example:

```
    start: greet greet
    ?greet: "(" /\w+/ ")"
          | /\w+/ /\w+/
```

Lark will parse “hello world (planet)” as:

```
start
    greet
        "hello"
        "world"
    "planet"
```

- Rules that begin with an exclamation mark will keep all their terminals (they won’t get filtered).

```
    !expr: "(" expr ")"
         | NAME+
    NAME: /\w+/
    %ignore " "
```

Will parse “((hello world))” as:

```
expr
  (
  expr
    (
    expr
      hello
      world
    )
  )
```

Using the ! prefix is usually a “code smell”, and may point to a flaw in your grammar design.

- Aliases - options in a rule can receive an alias. It will be then used as the branch name for the option, instead of the rule name.

Example:

```
    start: greet greet
    greet: "hello"
         | "world" -> planet
```

Lark will parse “hello world” as:

```
start
    greet
    planet
```

CHAPTER 11

API Reference

11.1 Lark

class lark.**Lark**(*grammar*, ***options*)

Main interface for the library.

It's mostly a thin wrapper for the many different parsers, and for the tree constructor.

Parameters

- **grammar** – a string or file-object containing the grammar spec (using Lark's ebnf syntax)
- **options** – a dictionary controlling various aspects of Lark.

Example

```
>>> Lark(r'''start: "foo" ''')
Lark(...)
```

=== General Options ===

start The start symbol. Either a string, or a list of strings for multiple possible starts (Default: "start")

debug Display debug information and extra warnings. Use only when debugging (default: False) When used with Earley, it generates a forest graph as "sppf.png", if 'dot' is installed.

transformer Applies the transformer to every parse tree (equivalent to applying it after the parse, but faster)

propagate_positions Propagates (line, column, end_line, end_column) attributes into all tree branches.

maybe_placeholders When True, the `[]` operator returns `None` when not matched.

When `False`, `[]` behaves like the `?` operator, and returns no value at all. (default= `False`. Recommended to set to `True`)

cache Cache the results of the Lark grammar analysis, for x2 to x3 faster loading. LALR only for now.

- When `False`, does nothing (default)
- When `True`, caches to a temporary file in the local directory
- When given a string, caches to the path pointed by the string

regex When True, uses the `regex` module instead of the stdlib `re`.

g_regex_flags Flags that are applied to all terminals (both regex and strings)

keep_all_tokens Prevent the tree builder from automagically removing "punctuation" tokens (default: False)

tree_class Lark will produce trees comprised of instances of this class instead of the default `lark.Tree`.

=== Algorithm Options ===

parser Decides which parser engine to use. Accepts "earley" or "lalr". (Default: "earley"). (there is also a "cyk" option for legacy)

lexer Decides whether or not to use a lexer stage

- "auto" (default): Choose for me based on the parser
- "standard": Use a standard lexer
- "contextual": Stronger lexer (only works with parser="lalr")
- "dynamic": Flexible and powerful (only with parser="earley")
- "dynamic_complete": Same as dynamic, but tries *every* variation of tokenizing possible.

ambiguity Decides how to handle ambiguity in the parse. Only relevant if parser="earley"

- "resolve": The parser will automatically choose the simplest derivation (it chooses consistently: greedy for tokens, non-greedy for rules)
- "explicit": The parser will return all derivations wrapped in "_ambig" tree nodes (i.e. a forest).
- "forest": The parser will return the root of the shared packed parse forest.

=== Misc. / Domain Specific Options ===

postlex Lexer post-processing (Default: None) Only works with the standard and contextual lexers.

priority How priorities should be evaluated - auto, none, normal, invert (Default: auto)

lexer_callbacks Dictionary of callbacks for the lexer. May alter tokens during lexing. Use with caution.

use_bytes Accept an input of type `bytes` instead of `str` (Python 3 only).

edit_terminals A callback for editing the terminals before parse.

import_paths A List of either paths or loader functions to specify from where grammars are imported

source_path Override the source of from where the grammar was loaded. Useful for relative imports and unconventional grammar loading

=== End Options ===

save(*f*)
Saves the instance into the given file object

Useful for caching and multiprocessing.

classmethod load(*f*)
Loads an instance from the given file object

Useful for caching and multiprocessing.

classmethod open(*grammar_filename*, *rel_to=None*, ***options*)
Create an instance of Lark with the grammar given by its filename

If `rel_to` is provided, the function will find the grammar filename in relation to it.

Example

```
>>> Lark.open("grammar_file.lark", rel_to=__file__, parser="lalr")
Lark(...)
```

parse(*text*, *start=None*, *on_error=None*)
Parse the given text, according to the options provided.

Parameters

- **text** (*str*) – Text to be parsed.
- **start** (*str, optional*) – Required if Lark was given multiple possible start symbols (using the start option).
- **on_error** (*function, optional*) – if provided, will be called on UnexpectedToken error. Return true to resume parsing. LALR only. See examples/error_puppet.py for an example of how to use on_error.

Returns If a transformer is supplied to `__init__`, returns whatever is the result of the transformation. Otherwise, returns a Tree instance.

11.1.1 Using Unicode character classes with `regex`

Python's builtin `re` module has a few persistent known bugs and also won't parse advanced regex features such as character classes. With `pip install lark-parser[regex]`, the `regex` module will be installed alongside lark and can act as a drop-in replacement to `re`.

Any instance of Lark instantiated with `regex=True` will use the `regex` module instead of `re`.

For example, we can use character classes to match PEP-3131 compliant Python identifiers:

```
from lark import Lark
>>> g = Lark(r"""
                    ?start: NAME
                    NAME: ID_START ID_CONTINUE*
                    ID_START: /[\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}_]+/
                    ID_CONTINUE: ID_START | /[\p{Mn}\p{Mc}\p{Nd}\p{Pc}·]+/
                """, regex=True)

>>> g.parse('')
''
```

11.2 Tree

class lark.**Tree**(*data*, *children*, *meta=None*)
The main tree class.

Creates a new tree, and stores "data" and "children" in attributes of the same name. Trees can be hashed and compared.

Parameters

- **data** – The name of the rule or alias
- **children** – List of matched sub-rules and terminals
- **meta** – Line & Column numbers (if `propagate_positions` is enabled). meta attributes: line, column, start_pos, end_line, end_column, end_pos

pretty(*indent_str=' '*)
Returns an indented string representation of the tree.

Great for debugging.

iter_subtrees()
Depth-first iteration.

Iterates over all the subtrees, never returning to the same node twice (Lark's parse-tree is actually a DAG).

find_pred(*pred*)
Returns all nodes of the tree that evaluate pred(node) as true.

find_data(*data*)
Returns all nodes of the tree whose data equals the given data.

iter_subtrees_topdown()
Breadth-first iteration.

Iterates over all the subtrees, return nodes in order like pretty() does.

11.3 Token

class lark.**Token**
A string with meta-information, that is produced by the lexer.

When parsing text, the resulting chunks of the input that haven't been discarded, will end up in the tree as Token instances. The Token class inherits from Python's `str`, so normal string comparisons and operations will work as expected.

type
Name of the token (as specified in grammar)

value
Value of the token (redundant, as `token.value == token` will always be true)

pos_in_stream
The index of the token in the text

line
The line of the token in the text (starting with 1)

column
The column of the token in the text (starting with 1)

end_line
The line where the token ends

end_column
The next column after the end of the token. For example, if the token is a single character with a column value of 4, end_column will be 5.

end_pos
the index where the token ends (basically `pos_in_stream + len(token)`)

11.4 Transformer, Visitor & Interpreter

See *Transformers & Visitors*.

11.5 ForestVisitor, ForestTransformer, & TreeForestTransformer

See *Working with the SPPF*.

11.6 UnexpectedInput

class `lark.exceptions.`**UnexpectedInput**
UnexpectedInput Error.

Used as a base class for the following exceptions:

- `UnexpectedToken`: The parser received an unexpected token
- `UnexpectedCharacters`: The lexer encountered an unexpected string

After catching one of these exceptions, you may call the following helper methods to create a nicer error message.

get_context(*text*, *span=40*)
Returns a pretty string pinpointing the error in the text, with span amount of context characters around it.

Note: The parser doesn't hold a copy of the text it has to parse, so you have to provide it again

---

match_examples(*parse_fn*, *examples*, *token_type_match_fallback=False*, *use_accepts=False*)
Allows you to detect what's wrong in the input text by matching against example errors.

Given a parser instance and a dictionary mapping some label with some malformed syntax examples, it'll return the label for the example that bests matches the current error. The function will iterate the dictionary until it finds a matching error, and return the corresponding value.

For an example usage, see *examples/error_reporting_lalr.py*

Parameters

- **parse_fn** – parse function (usually `lark_instance.parse`)
- **examples** – dictionary of `{'example_string': value}`.
- **use_accepts** – Recommended to call this with `use_accepts=True`. The default is `False` for backwards compatibility.

class `lark.exceptions.`**UnexpectedToken**(*token*, *expected*, *considered_rules=None*, *state=None*, *puppet=None*, *token_history=None*)
When the parser throws UnexpectedToken, it instantiates a puppet with its internal state. Users can then interactively set the puppet to the desired puppet state, and resume regular parsing.

see: *ParserPuppet*.

class lark.exceptions.**UnexpectedCharacters**(*seq*, *lex_pos*, *line*, *column*, *allowed=None*, *considered_tokens=None*, *state=None*, *token_history=None*)

11.7 ParserPuppet

class lark.parsers.lalr_puppet.**ParserPuppet**(*parser*, *parser_state*, *lexer_state*)

ParserPuppet gives you advanced control over error handling when parsing with LALR.

For a simpler, more streamlined interface, see the `on_error` argument to `Lark.parse()`.

feed_token(*token*)

Feed the parser with a token, and advance it to the next state, as if it received it from the lexer.

Note that `token` has to be an instance of `Token`.

pretty()

Print the output of `choices()` in a way that's easier to read.

choices()

Returns a dictionary of token types, matched to their action in the parser.

Only returns token types that are accepted by the current state.

Updated by `feed_token()`.

resume_parse()

Resume parsing from the current puppet state.

CHAPTER 12

Transformers & Visitors

Transformers & Visitors provide a convenient interface to process the parse-trees that Lark returns.

They are used by inheriting from the correct class (visitor or transformer), and implementing methods corresponding to the rule you wish to process. Each method accepts the children as an argument. That can be modified using the `v_args` decorator, which allows one to inline the arguments (akin to `*args`), or add the tree `meta` property as an argument.

See: visitors.py

12.1 Visitor

Visitors visit each node of the tree, and run the appropriate method on it according to the node's data.

They work bottom-up, starting with the leaves and ending at the root of the tree.

There are two classes that implement the visitor interface:

- `Visitor`: Visit every node (without recursion)
- `Visitor_Recursive`: Visit every node using recursion. Slightly faster.

Example:

```
class IncreaseAllNumbers(Visitor):
    def number(self, tree):
        assert tree.data == "number"
        tree.children[0] += 1

IncreaseAllNumbers().visit(parse_tree)
```

class lark.visitors.**Visitor**

Tree visitor, non-recursive (can handle huge trees).

Visiting a node calls its methods (provided by the user via inheritance) according to `tree.data`

visit(*tree*)
Visits the tree, starting with the leaves and finally the root (bottom-up)

visit_topdown(*tree*)
Visit the tree, starting at the root, and ending at the leaves (top-down)

__default__(*tree*)
Default function that is called if there is no attribute matching `tree.data`

Can be overridden. Defaults to doing nothing.

class lark.visitors.**Visitor_Recursive**
Bottom-up visitor, recursive.

Visiting a node calls its methods (provided by the user via inheritance) according to `tree.data`

Slightly faster than the non-recursive version.

visit(*tree*)
Visits the tree, starting with the leaves and finally the root (bottom-up)

visit_topdown(*tree*)
Visit the tree, starting at the root, and ending at the leaves (top-down)

__default__(*tree*)
Default function that is called if there is no attribute matching `tree.data`

Can be overridden. Defaults to doing nothing.

12.2 Interpreter

class lark.visitors.**Interpreter**
Interpreter walks the tree starting at the root.

Visits the tree, starting with the root and finally the leaves (top-down)

For each tree node, it calls its methods (provided by user via inheritance) according to `tree.data`.

Unlike `Transformer` and `Visitor`, the Interpreter doesn't automatically visit its sub-branches. The user has to explicitly call `visit`, `visit_children`, or use the `@visit_children_decor`. This allows the user to implement branching and loops.

Example:

```
class IncreaseSomeOfTheNumbers(Interpreter):
    def number(self, tree):
        tree.children[0] += 1

    def skip(self, tree):
        # skip this subtree. don't change any number node inside it.
        pass

    IncreaseSomeOfTheNumbers().visit(parse_tree)
```

12.3 Transformer

class lark.visitors.**Transformer**(*visit_tokens=True*)
Transformers visit each node of the tree, and run the appropriate method on it according to the node's data.

Calls its methods (provided by the user via inheritance) according to `tree.data`. The returned value replaces the old one in the structure.

They work bottom-up (or depth-first), starting with the leaves and ending at the root of the tree. Transformers can be used to implement map & reduce patterns. Because nodes are reduced from leaf to root, at any point the callbacks may assume the children have already been transformed (if applicable).

`Transformer` can do anything `Visitor` can do, but because it reconstructs the tree, it is slightly less efficient. It can be used to implement map or reduce patterns.

All these classes implement the transformer interface:

- `Transformer` - Recursively transforms the tree. This is the one you probably want.
- `Transformer_InPlace` - Non-recursive. Changes the tree in-place instead of returning new instances
- `Transformer_InPlaceRecursive` - Recursive. Changes the tree in-place instead of returning new instances

Parameters **`visit_tokens`** (*`bool, optional`*) – Should the transformer visit tokens in addition to rules. Setting this to `False` is slightly faster. Defaults to `True`. (For processing ignored tokens, use the `lexer_callbacks` options)

NOTE: A transformer without methods essentially performs a non-memoized deepcopy.

`transform`(*tree*)
Transform the given tree, and return the final result

`__mul__`(*other*)
Chain two transformers together, returning a new transformer.

`__default__`(*data*, *children*, *meta*)
Default function that is called if there is no attribute matching `data`

Can be overridden. Defaults to creating a new copy of the tree node (i.e. `return Tree(data, children, meta)`)

`__default_token__`(*token*)
Default function that is called if there is no attribute matching `token.type`

Can be overridden. Defaults to returning the token as-is.

Example:

```
from lark import Tree, Transformer

class EvalExpressions(Transformer):
    def expr(self, args):
            return eval(args[0])

t = Tree('a', [Tree('expr', ['1+2'])])
print(EvalExpressions().transform( t ))

# Prints: Tree(a, [3])
```

Example:

```
class T(Transformer):
    INT = int
    NUMBER = float
    def NAME(self, name):
```

(continues on next page)

(continued from previous page)

```
        return lookup_dict.get(name, name)

T(visit_tokens=True).transform(tree)
```

class lark.visitors.**Transformer_NonRecursive**(*visit_tokens=True*)
Same as Transformer but non-recursive.

Like Transformer, it doesn't change the original tree.

Useful for huge trees.

class lark.visitors.**Transformer_InPlace**(*visit_tokens=True*)
Same as Transformer, but non-recursive, and changes the tree in-place instead of returning new instances

Useful for huge trees. Conservative in memory.

class lark.visitors.**Transformer_InPlaceRecursive**(*visit_tokens=True*)
Same as Transformer, recursive, but changes the tree in-place instead of returning new instances

12.4 v_args

lark.visitors.**v_args**(*inline=False*, *meta=False*, *tree=False*, *wrapper=None*)
A convenience decorator factory for modifying the behavior of user-supplied visitor methods.

By default, callback methods of transformers/visitors accept one argument - a list of the node's children.

`v_args` can modify this behavior. When used on a transformer/visitor class definition, it applies to all the callback methods inside it.

`v_args` can be applied to a single method, or to an entire class. When applied to both, the options given to the method take precedence.

Parameters

- **inline** (*bool, optional*) – Children are provided as `*args` instead of a list argument (not recommended for very long lists).
- **meta** (*bool, optional*) – Provides two arguments: `children` and `meta` (instead of just the first)
- **tree** (*bool, optional*) – Provides the entire tree as the argument, instead of the children.
- **wrapper** (*function, optional*) – Provide a function to decorate all methods.

Example

```
@v_args(inline=True)
class SolveArith(Transformer):
    def add(self, left, right):
        return left + right


class ReverseNotation(Transformer_InPlace):
    @v_args(tree=True)
    def tree_node(self, tree):
        tree.children = tree.children[::-1]
```

12.5 Discard

class lark.visitors.**Discard**

When raising the Discard exception in a transformer callback, that node is discarded and won't appear in the parent.

CHAPTER 13

Working with the SPPF

When parsing with Earley, Lark provides the `ambiguity='forest'` option to obtain the shared packed parse forest (SPPF) produced by the parser as an alternative to it being automatically converted to a tree.

Lark provides a few tools to facilitate working with the SPPF. Here are some things to consider when deciding whether or not to use the SPPF.

Pros

- Efficient storage of highly ambiguous parses
- Precise handling of ambiguities
- Custom rule prioritizers
- Ability to handle infinite ambiguities
- Directly transform forest -> object instead of forest -> tree -> object

Cons

- More complex than working with a tree
- SPPF may contain nodes corresponding to rules generated internally
- Loss of Lark grammar features:
  - Rules starting with '_' are not inlined in the SPPF
  - Rules starting with '?' are never inlined in the SPPF
  - All tokens will appear in the SPPF

13.1 SymbolNode

class lark.parsers.earley_forest.**SymbolNode**(*s*, *start*, *end*)
A Symbol Node represents a symbol (or Intermediate LR0).

Symbol nodes are keyed by the symbol (s). For intermediate nodes s will be an LR0, stored as a tuple of (rule, ptr). For completed symbol nodes, s will be a string representing the non-terminal origin (i.e. the left hand side of the rule).

The children of a Symbol or Intermediate Node will always be Packed Nodes; with each Packed Node child representing a single derivation of a production.

Hence a Symbol Node with a single child is unambiguous.

Variables

- **s** – A Symbol, or a tuple of (rule, ptr) for an intermediate node.
- **start** – The index of the start of the substring matched by this symbol (inclusive).
- **end** – The index of the end of the substring matched by this symbol (exclusive).
- **is_intermediate** – True if this node is an intermediate node.
- **priority** – The priority of the node's symbol.

is_ambiguous
Returns True if this node is ambiguous.

children
Returns a list of this node's children sorted from greatest to least priority.

13.2 PackedNode

class lark.parsers.earley_forest.**PackedNode**(*parent*, *s*, *rule*, *start*, *left*, *right*)
A Packed Node represents a single derivation in a symbol node.

Variables

- **rule** – The rule associated with this node.
- **parent** – The parent of this node.
- **left** – The left child of this node. `None` if one does not exist.
- **right** – The right child of this node. `None` if one does not exist.
- **priority** – The priority of this node.

children
Returns a list of this node's children.

13.3 ForestVisitor

class lark.parsers.earley_forest.**ForestVisitor**
An abstract base class for building forest visitors.

This class performs a controllable depth-first walk of an SPPF. The visitor will not enter cycles and will backtrack if one is encountered. Subclasses are notified of cycles through the `on_cycle` method.

Behavior for visit events is defined by overriding the `visit*node*` functions.

The walk is controlled by the return values of the `visit*node_in` methods. Returning a node(s) will schedule them to be visited. The visitor will begin to backtrack if no nodes are returned.

visit_token_node(*node*)
Called when a `Token` is visited. `Token` nodes are always leaves.

visit_symbol_node_in(*node*)
Called when a symbol node is visited. Nodes that are returned will be scheduled to be visited. If `visit_intermediate_node_in` is not implemented, this function will be called for intermediate nodes as well.

visit_symbol_node_out(*node*)
Called after all nodes returned from a corresponding `visit_symbol_node_in` call have been visited. If `visit_intermediate_node_out` is not implemented, this function will be called for intermediate nodes as well.

visit_packed_node_in(*node*)
Called when a packed node is visited. Nodes that are returned will be scheduled to be visited.

visit_packed_node_out(*node*)
Called after all nodes returned from a corresponding `visit_packed_node_in` call have been visited.

on_cycle(*node*, *path*)
Called when a cycle is encountered.

Parameters

- **node** – The node that causes a cycle.
- **path** – The list of nodes being visited: nodes that have been entered but not exited. The first element is the root in a forest visit, and the last element is the node visited most recently. `path` should be treated as read-only.

get_cycle_in_path(*node*, *path*)
A utility function for use in `on_cycle` to obtain a slice of `path` that only contains the nodes that make up the cycle.

13.4 ForestTransformer

class lark.parsers.earley_forest.**ForestTransformer**
The base class for a bottom-up forest transformation. Most users will want to use `TreeForestTransformer` instead as it has a friendlier interface and covers most use cases.

Transformations are applied via inheritance and overriding of the `transform*node` methods.

`transform_token_node` receives a `Token` as an argument. All other methods receive the node that is being transformed and a list of the results of the transformations of that node's children. The return value of these methods are the resulting transformations.

If `Discard` is raised in a node's transformation, no data from that node will be passed to its parent's transformation.

transform(*root*)
Perform a transformation on an SPPF.

transform_symbol_node(*node*, *data*)
Transform a symbol node.

transform_intermediate_node(*node*, *data*)
Transform an intermediate node.

transform_packed_node(*node*, *data*)
Transform a packed node.

transform_token_node(*node*)
Transform a `Token`.

13.5 TreeForestTransformer

class lark.parsers.earley_forest.**TreeForestTransformer**(*tree_class=<class 'lark.tree.Tree'>*, *prioritizer=<lark.parsers.earley_forest.ForestSumVisitor object>*, *resolve_ambiguity=True*)

A `ForestTransformer` with a tree `Transformer`-like interface. By default, it will construct a tree.

Methods provided via inheritance are called based on the rule/symbol names of nodes in the forest.

Methods that act on rules will receive a list of the results of the transformations of the rule's children. By default, trees and tokens.

Methods that act on tokens will receive a token.

Alternatively, methods that act on rules may be annotated with `handles_ambiguity`. In this case, the function will receive a list of all the transformations of all the derivations of the rule. By default, a list of trees where each tree.data is equal to the rule name or one of its aliases.

Non-tree transformations are made possible by override of `__default__`, `__default_token__`, and `__default_ambig__`.

Note: Tree shaping features such as inlined rules and token filtering are not built into the transformation. Positions are also not propagated.

Parameters

- **tree_class** – The tree class to use for construction
- **prioritizer** – A `ForestVisitor` that manipulates the priorities of nodes in the SPPF.
- **resolve_ambiguity** – If True, ambiguities will be resolved based on priorities.

__default__(*name*, *data*)
Default operation on tree (for override).

Returns a tree with name with data as children.

__default_ambig__(*name*, *data*)
Default operation on ambiguous rule (for override).

Wraps data in an '_ambig_' node if it contains more than one element.

__default_token__(*node*)
Default operation on `Token` (for override).

Returns `node`.

13.6 handles_ambiguity

lark.parsers.earley_forest.**handles_ambiguity**(*func*)

Decorator for methods of subclasses of `TreeForestTransformer`. Denotes that the method should receive a list of transformed derivations.

CHAPTER 14

Importing grammars from Nearley

Lark comes with a tool to convert grammars from Nearley, a popular Earley library for Javascript. It uses Js2Py to convert and run the Javascript postprocessing code segments.

14.1 Requirements

1. Install Lark with the `nearley` component:

```
pip install lark-parser[nearley]
```

1. Acquire a copy of the nearley codebase. This can be done using:

```
git clone https://github.com/Hardmath123/nearley
```

14.2 Usage

Here's an example of how to import nearley's calculator example into Lark:

```
git clone https://github.com/Hardmath123/nearley
python -m lark.tools.nearley nearley/examples/calculator/arithmetic.ne main nearley >␣
↪ncalc.py
```

You can use the output as a regular python module:

```
>>> import ncalc
>>> ncalc.parse('sin(pi/4) ^ e')
0.38981434460254655
```

The Nearley converter also supports an experimental converter for newer JavaScript (ES6+), using the `--es6` flag:

```
git clone https://github.com/Hardmath123/nearley
python -m lark.tools.nearley nearley/examples/calculator/arithmetic.ne main nearley --
↪es6 > ncalc.py
```

14.3 Notes

- Lark currently cannot import templates from Nearley
- Lark currently cannot export grammars to Nearley

These might get added in the future, if enough users ask for them.

Lark is a modern parsing library for Python. Lark can parse any context-free grammar.

Lark provides:

- Advanced grammar language, based on EBNF
- Three parsing algorithms to choose from: Earley, LALR(1) and CYK
- Automatic tree construction, inferred from your grammar
- Fast unicode lexer with regexp support, and automatic line-counting

CHAPTER 15

Install Lark

```
$ pip install lark-parser
```

CHAPTER 16

Syntax Highlighting

- Sublime Text & TextMate
- Visual Studio Code (Or install through the vscode plugin system)
- Intellij & PyCharm
- Vim
- Atom

CHAPTER 17

Resources

- *Philosophy*
- *Features*
- Examples
- Online IDE
- Tutorials
  - How to write a DSL - Implements a toy LOGO-like language with an interpreter
  - *JSON parser - Tutorial* - Teaches you how to use Lark
  - Unofficial
    - Program Synthesis is Possible - Creates a DSL for Z3
- Guides
  - *How To Use Lark - Guide*
  - *How to develop Lark - Guide*
- Reference
  - *Grammar Reference*
  - *Tree Construction Reference*
  - *Transformers & Visitors*
  - *Working with the SPPF*
  - *API Reference*
  - *Importing grammars from Nearley*
  - Cheatsheet (PDF)
- Discussion
  - Gitter

- Forum (Google Groups)

Index

Symbols

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V