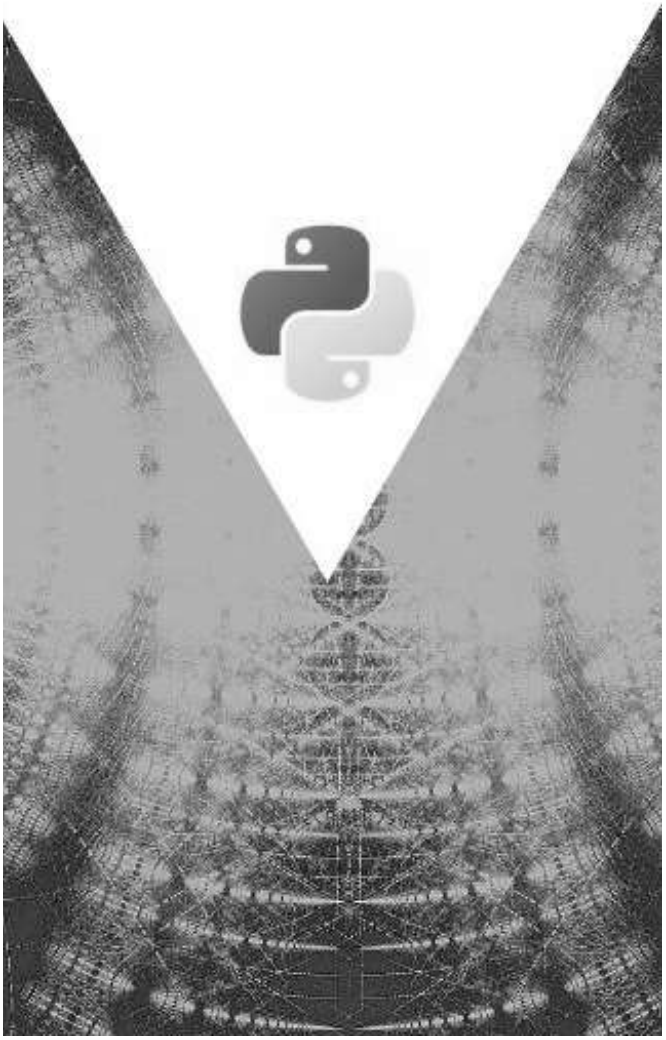
Parsing In Python: Tools And Libraries



Written by
Gabriele Tomassetti
in Parsing

- a tool or library to generate a parser: for example ANTLR, that you can use to build parsers for any language

In other cases you are out of luck.

Building Your Own Custom Parser By Hand

You may need to pick the second option if you have particular needs. Both in the sense that the language you need to parse cannot be parsed with traditional parser generators, or you have specific requirements that you cannot satisfy using a typical parser generator. For instance, because you need the best possible performance or a deep integration between different components.

A Tool Or Library To Generate A Parser

We are going to see:

- tools that can generate parsers usable from Python (and possibly from other languages)
- Python libraries to build parsers

Tools that can be used to generate the code for a parser are called **parser generators** or **compiler compiler**. Libraries that create parsers are known as **parser combinators**.

Parser generators (or parser combinators) are not trivial: you need some time to learn how to use them and not all types of parser generators are suitable for all kinds of languages. That is why we have prepared a list of the best known of them, with a short introduction for each of them. We are also concentrating on one target language: Python. This also means that (usually) the parser itself will be written in Python.

programmers we have prepared a short explanation for terms and concepts that you may encounter searching for a parser. We are not trying to give you formal explanations, but practical ones.

Structure Of A Parser

A parser is usually composed of two parts: a *lexer*, also known as *scanner* or *tokenizer*, and the proper parser. Not all parsers adopt this two-steps schema: some parsers do not depend on a lexer. They are called *scannerless parsers*.

A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser scans the tokens and produces the parsing result.

Then the lexer finds a '+' symbol, which corresponds to a second token of type *PLUS*, and lastly it finds another token of type *NUM*.

The parser will typically combine the tokens produced by the lexer and group them.

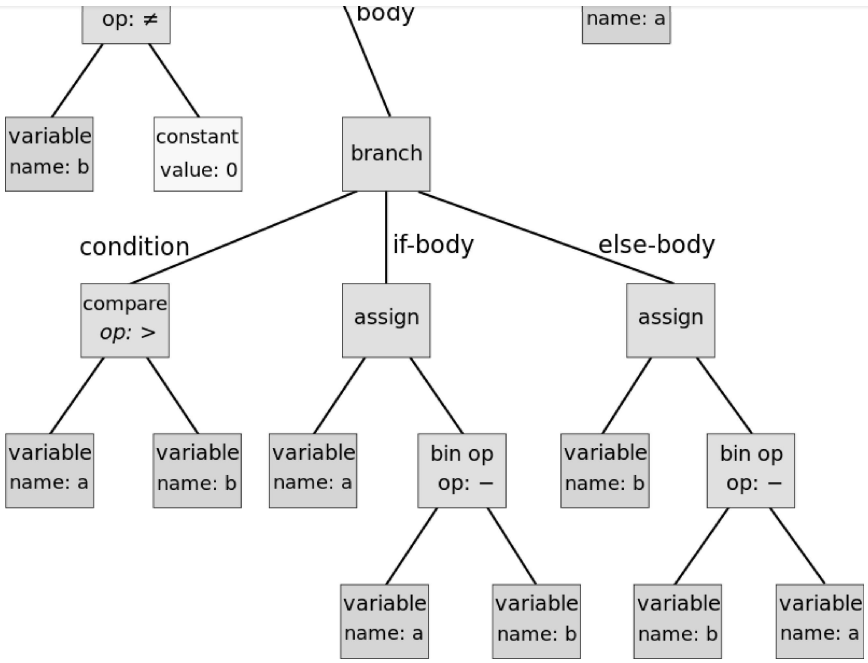
The definitions used by lexers or parser are called *rules* or *productions*. A lexer rule will specify that a sequence of digits correspond to a token of type *NUM*, while a parser rule will specify that a sequence of tokens of type *NUM, PLUS, NUM* corresponds to an expression.

Scannerless parsers are different because they process directly the original text, instead of processing a list of tokens produced by a lexer.

It is now typical to find suites that can generate both a lexer and parser. In the past it was instead more common to combine two different tools: one to

- they are both **trees**: there is a root representing the whole piece of code parsed. Then there are smaller subtrees representing portions of code that become smaller until single tokens appear in the tree
- the difference is the level of abstraction: the parse tree contains all the tokens which appeared in the program and possibly a set of intermediate rules. The AST instead is a polished version of the parse tree where the information that could be derived or is not important to understand the piece of code is removed

In the AST some information is lost, for instance comments and grouping symbols (parentheses) are not represented. Things like comments are superfluous for a program and grouping symbols are implicitly defined by the structure of the tree.



Sometimes you may want to start producing a parse tree and then derive from it an AST. This can make sense because the parse tree is easier to produce for the parser (it is a direct representation of the parsing process) but the AST is simpler and easier to process by the following steps. By following steps we mean all the operations that you may want to perform on the

the "if" keyword, followed by a left parenthesis, an expression, a right parenthesis and a statement.

A rule could reference other rules or token types. In the example of the if statement, the keyword "if", the left and the right parenthesis were token types, while expression and statement were references to other rules.

The most used format to describe grammars is the **Backus-Naur Form (BNF)**, which also has many variants, including the **Extended Backus-Naur Form**. The Extended variant has the advantage of including a simple way to denote repetitions. A typical rule in a Backus-Naur grammar looks like this:

```
<symbol> ::= __expression__
```

support for left-recursive rules. This means that a rule could start with a reference to itself. This reference could be also indirect.

Consider for example arithmetic operations. An addition could be described as two expression(s) separated by the plus (+) symbol, but an expression could also contain other additions.

```
 addition       ::= expression '+' expression
 multiplication ::= expression '*' expression
 // an expression could be an addition or a
multiplication or a number
 expression     ::= addition | multiplication |//
a number
```

This description also match multiple additions like 5 + 4 + 3. That is because it can be interpreted as expression (5) (‘+’) expression(4+3). And then 4 + 3 itself can be divided in its two components.

and *context-free language*s. We could give you the formal definition according to the Chomsky hierarchy of languages, but it would not be that useful. Let's look at some practical aspects instead.

A regular language can be defined by a series of regular expressions, while a context-free one need something more. A simple rule of thumb is that if a grammar of a language has recursive elements it is not a regular language. For instance, as we said elsewhere, HTML is not a regular language. In fact, most programming languages are context-free languages.

Usually to a kind of language correspond the same kind of grammar. That is to say there are regular grammars and context-free grammars that corresponds respectively to regular and context-free languages. But to complicate matters, there is a

input, a CFG will be ambiguous and thus wrong. Instead with PEG the first applicable choice will be chosen, and this automatically solve some ambiguities.

Another difference is that PEG use scannerless parsers: they do not need a separate lexer, or lexical analysis phase.

Traditionally both PEG and some CFG have been unable to deal with left-recursive rules, but some tools have found workarounds for this. Either by modifying the basic parsing algorithm, or by having the tool automatically rewrite a left-recursive rule in a non recursive way. Either of these ways has downsides: either by making the generated parser less intelligible or by worsen its performance. However, in practical terms, the advantages of easier and quicker development outweigh the drawbacks.

additional ready-to-use classes, such Listeners or Visitors. Some tools instead offer the chance to embed code inside the grammar to be executed every time the specific rule is matched.

Usually you need a runtime library and/or program to use the generated parser.

Context Free

Let's see the tools that generate Context Free parsers.

ANTLR

ANTLR is a great parser generator written in Java that can also generate parsers for Python and many other languages. There is also a beta version for TypeScript from the same guy that makes the *optimized C#*

manipulate or interact with the elements of the tree, while the second is useful when you just have to do something when a rule is matched.

The typical grammar is divided in two parts: lexer rules and parser rules. The division is implicit, since all the rules starting with an uppercase letter are lexer rules, while the ones starting with a lowercase letter are parser rules. Alternatively lexer and parser grammars can be defined in separate files.

```
1.  grammar simple;
2.
3.  basic   : NAME ':' NAME ;
4.
5.  NAME    : [a-zA-Z]* ;
6.
7.  COMMENT : '/*' .*? '*/' -> skip ;
```

Python, implementing Earley & LALR(1) and an easy interface

Lark is a parser generator that works as a library. You write the grammar in a string or a file and then use it as an argument to dynamically generate the parser. Lark can use two algorithms: Earley is used when you need to parse all grammars and LALR when you need speed. Earley can parse also ambiguous grammars. Lark offers the chance to automatically solve the ambiguity by choosing the simplest option or reporting all options.

Lark grammars are written in an EBNF format. They cannot include actions. This means that they are clean and readable, but also that you have to traverse the resulting tree yourself. Although there is a function that can help with that if you use the LALR algorithm. On the positive side you can also use specific notations in the grammar to automatically

```
                        | product "/" item -> div

                    ?item: NUMBER -> number
                         | "-" item -> neg
                         | "(" sum ")"

                    %import common.NUMBER
                    %ignore WS
            ''', start='sum')
```

Lark comes with a tool to convert Nearley grammars in its own format. It also includes a useful function to transform the tree generated by the parser in an image.

It has a sufficient documentation, with examples and tutorials available. There is also a small reference.

but they are harder to read than a traditional grammar.

```
// from the documentation
class ExprParser(lrparsing.Grammar):
    #
    # Put Tokens we don't want to re-type in a TokenRegistry.
    #
    class T(lrparsing.TokenRegistry):
        integer = Token(re="[0-9]+")
        integer["key"] = "I'm a mapping!"
        ident = Token(re="[A-Za-z_][A-Za-z_0-9]*")
    #
    # Grammar rules.
    #
    expr = Ref("expr")                  # Forward reference
    call = T.ident + '(' + List(expr, ',') + ')'
    atom = T.ident | T.integer | Token('(') + expr + ')' | call
    expr = Prio(                        # If ambiguous choose atom 1st, ...
```

Lrparsing also provide some basic functions to print parsing tree and grammar rules for debugging purposes.

The documentation is really good: it explains everything you need to know about the library and it also provide some guidance on creating good grammars (eg. solving ambiguities). There are also quite complex example grammars, like one for SQLite.

PLY

> *PLY doesn't try to do anything more or less than provide the basic lex/yacc functionality. In other words, it's not a large parsing framework or a component of some larger system.*

```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'+'
t_MINUS   = r'-'
t_TIMES   = r'*'
t_DIVIDE  = r'/'
t_LPAREN  = r'('
t_RPAREN  = r')'

# A regular expression rule with some
```

```
        t.value[0])
39.         t.lexer.skip(1)
40.
41.   # Build the lexer
42.   lexer = lex.lex()
```

The documentation is extensive, clear, with abundant examples and explanations of parsing concepts. All that you need, if you can get pass the '90 looks.

There is a port for RPython called RPLY.

PlyPlus

> *Plyplus is a general-purpose parser built on top of PLY (LALR(1)), and written in Python. Plyplus features a modern design, and focuses on simplicity without losing power.*

```
// from the documentation
start: add;

// Rules
?add: (add add_symbol)? mul;
?mul: (mul mul_symbol)? atom;
// rules preceded by @ will not appear in the tree
@atom: neg | number | '(' add ')';
neg: '-' atom;

// Tokens
number: '[d.]+';
mul_symbol: '*' | '/';
add_symbol: '+' | '-';

WS: '[ t]+' (%ignore);
```

PlyPlus include a function to draw an image of a parse tree based upon pydot and graphviz. PlyPlus has unique features, too. It allows you to select nodes in the AST using selectors similar to the CSS selectors used in web development. For instance, if you want to

Pyleri

Python Left-Right Parser (pyleri) is part of a family of similar parser generators for JavaScript, Python, C, Go and Java.

A grammar for Pyleri must be defined in Python expressions that are part of a class. Once it is defined, the grammar can be exported as a file defining the grammar in Python or any other supported language. For example, you can define the grammar in Python, export it to JacaScript and then use the JavaScript version of pyleri to run it. You cannot do the inverse, i.e., you cannot create a grammar in JavaScript and export it to Python. So, even if you want to use another language, it is better to create the grammar in Python and then export it to that language.

Apart from this interesting feature, Pyleri is a simple and easy to use tool.

In practical terms there are two kinds of parsing rules: simple and combination of simple ones. The simple ones are essentially tokens created with regular expressions, while the complex ones are created using ready-to-use parsing functions (e.g., Sequence to parse a sequence of elements).

So, it is a cross between a parser generator and a parser combinator. However, it is more powerful that a traditional parser combinator and can also generate a parse tree. Another neat feature is that it provide a property `expecting`, that list the elements that it can accept at that particular position. This is very useful if you are building auto-completion functionality.

This mixture of simplicity of syntax and powerful features can quite attractive for people that something powerful, but are not used to a traditional

> *parser with backtracking and memoization (a.k.a. pacrat parser). Arpeggio grammars are based on PEG formalism.*

The documentation defines Arpeggio as a parser interpreter, since parser are generated dynamically from a grammar. In any case it does not work any different from many other Python parser generators. A peculiarity of Arpeggio is that you can define a grammar in a textual PEG format or using Python expressions. Actually, there are two dialects of PEGs, one with a cleaner Python-like syntax and the other the traditional PEG one.

Arpeggio generate a simple parse tree, but it supports the use of a visitor. The visitor can also include a second action to perform after all the tree nodes have been processed. This is used for post-processing, for instance it can be used to deal with symbol reference.

```
8.
9.   [..]
10.
11.   def main(debug=False):
12.       # First we will make a parser - an
instance of the CVS parser model.
13.       # Parser model is given in the form
of python constructs therefore we
14.       # are using ParserPython class.
15.       # Skipping of whitespace will be done
only for tabs and spaces. Newlines
16.       # have semantics in csv files. They
are used to separate records.
17.    parser = ParserPython(csvfile, ws='t ',
debug=debug)
18.
19.    [..]
```

There are a couple of options for debugging: verbose and informative ouput and the generation of DOT files of the parser. The DOT files can be used for creating a visualization of the parser, but you will have to call

> *Canopy is a parser compiler targeting Java, JavaScript, Python and Ruby. It takes a file describing a parsing expression grammar and compiles it into a parser module in the target language. The generated parsers have no runtime dependency on Canopy itself.*

It also provides easy access to the parse tree nodes.

A Canopy grammar has the neat feature of using actions annotation to use custom code in the parser. In practical terms. you just write the name of a function next to a rule and then you implement the function in your source code.

```
1.  // the actions are prepended by %
2.  grammar Maps
3.    map     <-  "{" string ":" value "}"
    %make_map
```

Parsimonious

> *Parsimonious aims to be the fastest arbitrary-lookahead parser written in pure Python—and the most usable. It's based on parsing expression grammars (PEGs), which means you feed it a simplified sort of EBNF notation.*

Parsimonious is a no-nonsense tool designed for speed and low usage of RAM. It is also a no-documentation tool, there are not even complete examples. Actually the short README file explain the basics and redirect you to Docstring for more specific information.

```
5.      italic_text = "'" text "'"
6.      text        = ~"[A-Z 0-9]*"i
7.      """)
```

pyPEG

> *pyPEG is a plain and simple intrinsic parser interpreter framework for Python version 2.7 and 3.x*

PyPEG is a framework to parse and compose text. Which means that you define a grammar in a syntax as powerful as PEG, but you do it in Python code. And then you use this grammar to parse and/or compose a text based upon that grammar. Obviously if you compose a text you have to provide the data yourself. In this case it works as a template system.

```
8.
9.  class Parameters(Namespace):
10. grammar = optional(csl(Parameter))
11.
12. class Instruction(str):
13. grammar = word, ";"
14.
15. block = "{", maybe_some(Instruction), "}"
16. class Function(List):
17. grammar = attr("typing", Type), name(),
18.         "(", attr("parms", Parameters),
    ")", block
19.
20. f = parse("int f(int a, long b) {
    do_this; do_that; }", Function)
```

PyPEG does not produce a standard tree, but a structure based upon the defined grammar. Look at what happens for the previous example.

```
1.  # execute the example
2.  >>> f.name
3.  Symbol('f')
```

outputs memoizing (Packrat) PEG parsers in Python.

TatSu is the successor of Grako, another parser generator tool, and it has a good level of compatibility with it. It can create a parser dynamically from a grammar or compiling into a Python module.

TatSu generate PEG parsers, but grammars are defined in a variant of EBNF. Though the order of rules matters as it is usual for PEG grammars. So it is actually a sort of cross between the two. This variant includes support for dealing with associativity and simplifying the generated tree or model (more on that later). Support for left-recursive rule is present, but experimental.

```
1.  // TatSu example grammar from the
    tutorial
2.  @@grammar::CALC
```

```
21.         ;
22.
23.   factor
24.         =
25.         | '(' expression ')'
26.         | number
27.         ;
28.
29.   number
30.         =
31.         /d+/
```

TatSu grammars cannot include actions, that can be defined in a separate Python class. Instead you have to annotate the grammar if you want to use an object model in place of semantic actions. An object model is a way to separate the parsing process from the entity that is parsed. In practical terms instead of doing something when a certain rule is matched you do something when a certain object is defined. This object may be defined by more than one rule.

```
2.
3.  class CalcWalker(NodeWalker):
4.      def walk_object(self, node):
5.          return node
6.
7.      def walk__add(self, node):
8.          return self.walk(node.left) +
    self.walk(node.right)
9.
10.     def walk__subtract(self, node):
11.         return self.walk(node.left) -
    self.walk(node.right)
12.
13.     def walk__multiply(self, node):
14.         return self.walk(node.left) *
    self.walk(node.right)
15.
16.     def walk__divide(self, node):
17.         return self.walk(node.left) /
    self.walk(node.right)
18.
19.  def parse_and_walk_model():
20.     grammar =
    open('grammars/calc_model.ebnf').read()
21.
22.     parser = tatsu.compile(grammar,
    asmodel=True)
23.     model = parser.parse('3 + 5 * ( 10 -
```

graphical representation of the tree using pygraphviz. ANLTR grammar may have to be manually adapted to respect PEG constraints.

The documentation is complete: it shows all the features, provide examples and even has basic introduction to parsing concepts, like AST.

Waxeye

> *Waxeye is a parser generator based on parsing expression grammars (PEGs). It supports C, Java, Javascript, Python, Ruby and Scheme.*

Waxeye can facilitate the creation of an AST by defining nodes in the grammar that will not be included in the generated tree. That is quite useful,

```
6.
7.  prod  <- unary *([*/] ws unary)
8.
9.  unary <= '-' ws unary
10.        | :'(' ws sum :')' ws
11.        | num
12.
13. num   <- +[0-9] ?('.' +[0-9]) ws
14.
15. ws    <: *[ tnr]
```

A particular feature of Waxeye is that it provides some help to compose different grammars together and then it facilitates modularity. For instance, you could create a common grammar for identifiers, that are usually similar in many languages.

Waxeye has a great documentation in the form of a manual that explains basic concepts and how to use the tool for all the languages it supports. There are a few example grammars.

Some readers have pointed us to funcparserlib, but we decided to not include it because it has been unmantained for a few years.

Parsec.py, Parsy and Pyparsing

> *A universal Python parser combinator library inspired by Parsec library of Haskell.*

That is basically the extent of the documentation on Parsec.py. Though there are a couple of examples. If you already know how to use the original Parsec library or one of its many clones you can try to use it. It does not look bad, but the lack of documentation is a problem for new users.

“industrial-strength” parser combinator such Parsec (the original one), but it has a few nice features. For instance, you can create a generator function to create a parser. It now requires Python 3.3 or later, which should only be a problem for people stuck with Python 2.

The project now has ample documentation, examples and a tutorial. The following example comes from the documentation and shows how to parse a date.

```
# from the documentation
# parsing a date
from parsy import string, regex
from datetime import date
ddmmyy = regex(r'[0-9]{2}').map(int).sep_by(string("-"), min=3, max=3).combine(
                    lambda d, m, y: date(2000 + y, m, d))
ddmmyy.parse('06-05-14')
```

Pyparsing is as equally powerful as a traditional parser combinator, it works a bit differently and this lack in the proper documentation makes it frustrating.

However, if you take the time to learn on its own, the following example shows that can be easy to use.

```
1.  # example from the documentation
2.  # define grammar
3.  greet = Word( alphas ) + "," + Word(
    alphas ) + "!"
4.
5.  # input string
6.  hello = "Hello, World!"
7.
8.  # parse input string
9.  print hello, "->", greet.parseString(
    hello )
```

Python the best choice is to rely on your own Python interpreter.

The standard reference implementation of Python, known as CPython, include a few modules to access its internals for parsing: tokenize, parser and ast. You may also be able to use the parser in the PyPy interpreter.

Parsing with Regular Expressions and The Like

Usually you resort to parsing libraries and tools when regular expression are not enough. However, there is a good library for Python than can extend the life and usefulness of regular expressions or using elements of similar complexity.

The basic idea is that you define regular expressions, the patterns in which they can combine and the functions that are called when an expression or pattern is found. You must define functions in Python, but expressions and pattern can be defined in Yaml, JSON or Python.

In this example from the documentation expressions and patterns are defined in Yaml.

```
1.  Color:
2.      Basic Color:
3.          Expression:
    (Red|Orange|Yellow|Green|Blue|Violet|Brown
    |Black)
4.          Matches: Orange | Green
5.          Non-Matches: White
6.          Groups:
7.            - Color
8.
9.  Time:
10.     Basic Time:
11.         Expression: ([0-9]|[1][0-2]) s?
```

```
    # Angle brackets detonate expression
groups
    # Multiple expressions in one group are
combined together
```

An example function in Python for the pattern.

```
from datetime import time
def color_time(Color=None, Time=None):
    Color, Hour, Period = Color[0],
int(Time[0]), Time[1]
    if Period == 'pm':
        Hour += 12
    Time = time(hour=Hour)

    return Color, Time

functions = {
   'BasicColorTime' : color_time,
}
```

The file that puts everything together.

Parsing Binary Data: Construct

> *Instead of writing imperative code to parse a piece of data, you declaratively define a data structure that describes your data. As this data structure is not code, you can use it in one direction to parse data into Pythonic objects, and in the other direction, to build objects into binary data.*

And that is it: Construct. You could parse binary data even with some parser generators (e.g. ANTLR), but Constuct make it much easier. It is a sort of DSL combined with a parser combinator to parse binary formats. It gives you a bunch of fields to manage binary data: apart from the obvious ones (e.g. float,

```
        ULInt16("height"),
        [..]
        If(lambda ctx: ctx["flags"]
["global_color_table"],
            Array(lambda ctx: 2**
(ctx["flags"]["global_color_table_bpp"] +
1),
                Struct("palette",
                    ULInt8("R"),
                    ULInt8("G"),
                    ULInt8("B")
                )
            )
        )
    )

    gif_header = Struct("gif_header",
        Const("signature", b"GIF"),
        Const("version", b"89a"),
    )

    [..]

    gif_file = Struct("gif_file",
        gif_header,
        gif_logical_screen,
        [..]
```

Summary

Any programming language has a different community with its peculiarities. These differences remain even when we compare the same interests across the languages. For instance, when we compare parsers tools we can see how Java and Python developers live in a different world.

The parsing tools and libraries for Python for the most part use very readable grammars and are simple to use. But the most interesting thing is that they cover a very wide spectrum of competence and use cases. There seems to be an uninterrupted line of tools available from regular expression, passing through Reparse to end with TatSu and ANTLR.

Update: since we originally wrote this article we have also wrote a tutorial on building languages using textX and VSCode. You can find it here:

Quick Domain-Specific Languages in Python with textX

More on Parsing

ANTLR

Application modernization

Code processing

Consulting

Domain specific languages

Editors

Jetbrains MPS