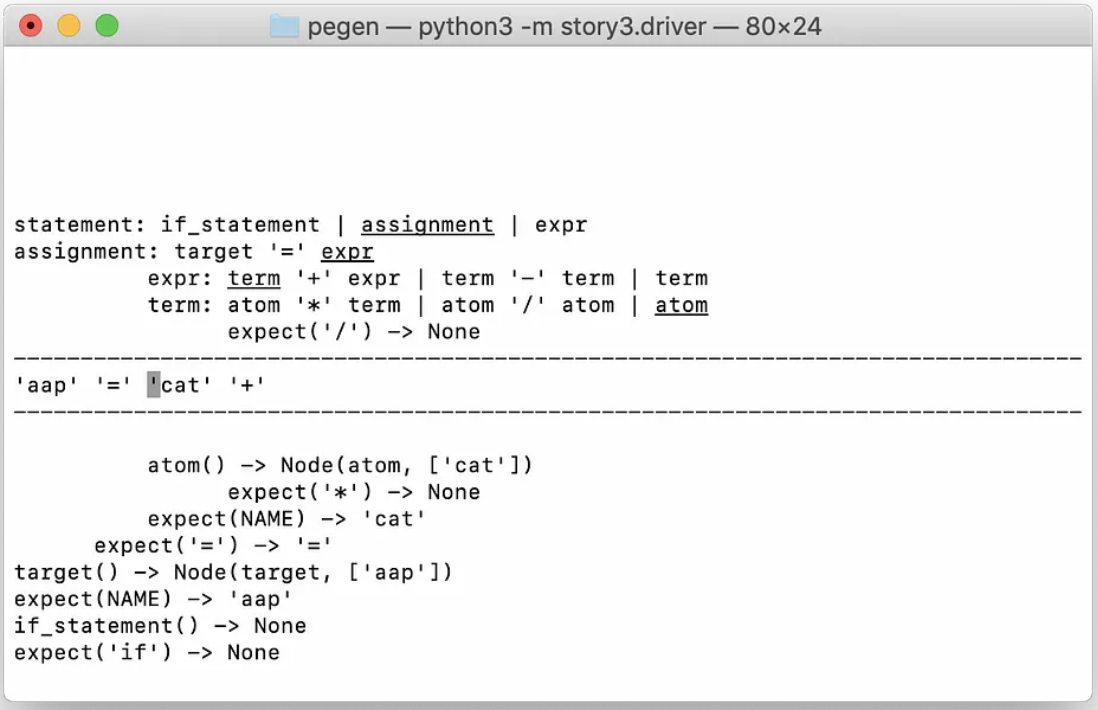
PEG Parsing Series Overview

Guido van Rossum · Follow
1 min read · Sep 1, 2019

My series of blog posts about PEG parsing keeps expanding. Instead of updating each part to link to all other parts, here's the table of content:

1. PEG Parsers
2. Building a PEG Parser
3. Generating a PEG Parser
4. Visualizing PEG Parsing
5. Left-recursive PEG Grammars
6. Adding Actions to a PEG Grammar
7. A Meta-Grammar for PEG Parsers
8. Implementing PEG Features
9. PEG at the Core Developer Sprint

A video of a talk I gave about this topic at North Bay Python is up on YouTube: Writing a PEG parser for fun and profit



Python Parsing

Update: April 2, 2020. In case you are wondering what's happening, we now have PEP 617 up, which proposes to replace the current parser in CPython with a PEG-based parser.

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

More from Guido van Rossum

PEG Parsers

Guido van Rossum · Follow
7 min read · Jul 21, 2019

Some years ago someone asked whether it would make sense to switch Python to a PEG parser. (Or a PEG grammar; I don't recall exactly what was said by whom, or when.) I looked into it a bit and wasn't sure what to think, so I dropped the subject. Recently I've learned more about PEG (Parsing Expression Grammars), and I now think it's an interesting alternative to the home-grown parser generator that I developed 30 years ago when I started working on Python. (That parser generator, dubbed "pgen", was just about the first piece of code I wrote for Python.)

[This is part 1 of my PEG series. See the Series Overview for the rest.]

The reason I'm now interested in PEG parsing is that I've become somewhat annoyed with pgen's limitations. It uses a variant of LL(1) parsing that I cooked up myself — I didn't like grammar rules that could produce the empty string, so I disallowed that, thereby simplifying the algorithm for producing parsing tables somewhat. I also invented my own EBNF-like grammar notation, which I still like very much.

Here are some of the issues with pgen that annoy me. The "1" in the LL(1) moniker implies that it uses only a single token lookahead, and this limits our ability of writing nice grammar rules. For example, a Python statement can be an

expression or an assignment (or other things, but those all start with a dedicated keyword like `if` or `def`). We'd like to write this syntax as follows using the pgen notation. (Note that this example describes a toy language that is a tiny subset of Python, as is traditional in writing about language design.)

```
statement: assignment | expr | if_statement
expr: expr '+' term | expr '-' term | term
term: term '*' atom | term '/' atom | atom
atom: NAME | NUMBER | '(' expr ')'
assignment: target '=' expr
target: NAME
if_statement: 'if' expr ':' statement
```

A few words about the notation: `NAME` and `NUMBER` are tokens and predefined outside the grammar. Strings in quotes like `'+'` or `'if'` are are also tokens. (I should talk about tokens some other time.) Grammar rules start with a rule name followed by `:`, followed by one or more alternatives separated by `|`.

The problem is that if you write the grammar like this, the parser does not work, and pgen will complain. One of the issues is that some rules (`expr` and `term`) are left-recursive, and pgen isn't smart enough to do the right thing here. This is typically solved by rewriting those rules, for example (leaving the other rules unchanged):

```
expr: term ('+' term | '-' term)*
term: atom ('*' atom | '/' atom)*
```

This reveals a few bits of pgen's EBNF capabilities: you can nest alternatives inside parentheses, and you can create repetitions by placing `*` after an item, so the rule for `expr` here means "it's a term, followed by zero or more repetitions of plus followed by a term or minus followed by a term". This grammar accepts the same language as the first version, but it doesn't reflect the intent of the language designer as well — in particular, it doesn't show that the operators are left-binding, which is important when you are trying to generate code.

But there's another annoying problem in this toy language (and in Python). Because of the single-token lookahead, the parser cannot determine whether it is

looking at the start of an expression or an assignment. At the beginning of a statement, the parser needs to decide what alternative for `statement` it is seeing from the first token it sees. (Why? This is how pgen's parsing automation works.) Let's say our program is this:

```
answer = 42
```

This program is tokenized into three tokens: `NAME` (with value `answer`), `'='`, and `NUMBER` (with value `42`). The only lookahead we have at the start of the program is the first token, `NAME`. The rule we are trying to satisfy at this point is `statement` (the grammar's start symbol). This rule has three alternatives: `expr`, `assignment`, and `if_statement`. We can rule out `if_statement`, because the lookahead token isn't `'if'`. But both `expr` and `assignment` (can) start with a `NAME` token, and because of this pgen rejects our grammar as being ambiguous.

(That's not entirely correct, since technically the *grammar* isn't ambiguous per se; but we'll ignore this because I don't know if there's a better word. And how does pgen decide this? It computes something called the FIRST set for each grammar rule, and it complains if the FIRST sets of the choices at any given point overlap.)

So couldn't we solve this annoyance by giving the parser a larger lookahead buffer? For our toy example, a second lookahead token would be enough, since in this grammar the second token of an assignment must be `'='`. But in a more realistic language like Python, you may need an unlimited lookahead buffer, since the stuff to the left of the `'='` token may be arbitrarily complex, for example:

```
table[index + 1].name.first = 'Steven'
```

That's already ten tokens before we encounter the `'='` token, and I could cook up arbitrary long examples if challenged. What we've done to solve this in pgen is to change the grammar to accept some illegal programs, adding an extra check in a later pass that generates a `SyntaxError` if it finds an invalid left-hand side for an assignment. For our toy language, this comes down to writing the following:

```
statement: assignment_or_expr | if_statement
assignment_or_expr: expr ['=' expr]
```

(The square brackets indicate an optional part.) And then in a subsequent compiler pass (say, when generating bytecode) we check whether there's an `'='` or not, and if there is, we check that the left-hand side follows the syntax for `target`.

There's a similar annoyance around keyword arguments in function calls. We would *like* to write something like this (again, a simplified version of Python's call syntax):

```
call: atom '(' arguments ')'
arguments: arg (',' arg)*
arg: posarg | kwarg
posarg: expr
kwarg: NAME '=' expr
```

But the single-token lookahead can't to tell the parser whether a `NAME` at the start of an argument is the beginning of `posarg` (since `expr` may start with `NAME`) or the beginning of `kwarg`. Again, Python's current parser solves this by essentially stating

```
arg: expr ['=' expr]
```

and then sorting out the cases in a subsequent compiler pass. (We even got this slightly wrong and allowed things like `foo((a)=1)`, giving it the same meaning as `foo(a=1)`, until we fixed it in Python 3.8.)

So how does a PEG parser solve these annoyances? By using an infinite lookahead buffer! The typical implementation of a PEG parser uses something called "packrat parsing", which not only loads the entire program in memory before parsing it, but also allows the parser to backtrack arbitrarily. While the term PEG primarily refers to the *grammar* notation, the *parsers* generated from PEG

grammars are typically recursive-descent parsers with unlimited backtracking, and packrat parsing makes this efficient by memoizing the rules already matched for each position.

This makes everything easy, but of course there's a cost: memory. Thirty years ago, I had a good reason to use a parsing technology with a single token lookahead: memory was expensive. LL(1) parsing (and other technologies like LALR(1), made famous by YACC) uses a state machine and a stack (a "push-down automaton") to construct a parse tree efficiently.

Fortunately the computers on which CPython runs have a lot more memory than 30 years ago, and keeping the entire file in memory really isn't much of a burden any more. For example, the largest non-test file in the stdlib that I could find is `_pydecimal.py`, which clocks in at around 223 kilobytes. In a Gigabyte world, that's essentially nothing. And that's what led me to have another look at parsing technology.

But there's another thing about CPython's current parser that bugs me. Compilers are complicated things, and CPython's is no exception: while the output of the pgen-generated parser is a parse tree, this parse tree is not directly used as the input to the code generator: first it is transformed to an *abstract syntax tree* (AST), and then that AST is compiled into bytecode. (There's more to it, but that's not my focus here.)

Why not compile from the parse tree? That is how it originally worked, but about 15 years ago we found that the compiler was complicated by the structure of the parse tree, and we introduced a separate AST, and a separate translation phase from parse tree to AST. As Python evolves, the AST is more stable than the parse tree, so this reduces the opportunity for bugs in the compiler.

The AST is also easier to work with for third-party code that wants to inspect Python code, and is exposed through the popular `ast` module. This module also lets you construct AST nodes from scratch and modify existing AST nodes, and you can compile the new nodes to bytecode. The latter has enabled an entire cottage industry of language extensions for Python. (The parse tree is *also* exposed to Python users, via the `parser` module, but it is much more cumbersome to work with; therefore it has gone out of style in favor of the `ast` module.)

My idea now, putting these things together, is to see if we can create a new parser for CPython that uses PEG and packrat parsing to construct the AST directly during parsing, thereby skipping the intermediate parse tree construction, possibly *saving* memory despite using an infinite lookahead buffer. I'm not there yet, but I have a prototype that can compile a subset of Python into an AST at about the same speed as CPython's current parser. It uses more memory, however, and I expect that extending the subset to the full language will slow down the PEG parser. But I also haven't done anything to optimize it, so I am hopeful.

A final advantage of switching to PEG is that it provides more flexibility for future evolution of the language. In the past it's been said that pgen's LL(1) restrictions have helped Python's grammar stay simple. That may well be so, but we have plenty of other processes in place to prevent unchecked growth of the language (mainly the PEP process, aided by very strict backwards compatibility requirements and the new governance structure). So I'm not worried.

I have a lot more to write about PEG parsing and my specific implementation, but I'll write about that in a later post, after I've cleaned up the code.

Python Parsing

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

Member-only story

Building a PEG Parser

Guido van Rossum · Follow

8 min read · Jul 28, 2019

Inspired by only a partial understanding of PEG parsing I decided to build one. The result may not be a great general-purpose PEG parser generator — there are already many of those (e.g. TatSu is written in Python and generates Python code) — but it was a good way to learn about PEG, and it furthers my goal of replacing CPython's parser with one built from a PEG grammar.

[This is part 2 of my PEG series. See the Series Overview for the rest.]

In this section I lay the groundwork for understanding how the generated parser works, by showing a simple hand-written parser.

(By the way, as an experiment, I'm not sprinkling links all over my writings. If there's something you don't understand, just Google it. :-)

The most common way of PEG parsing uses a recursive descent parser with unlimited backtracking. Take the toy grammar from last week's article:

```
statement: assignment | expr | if_statement
expr: expr '+' term | expr '-' term | term
term: term '*' atom | term '/' atom | atom
atom: NAME | NUMBER | '(' expr ')'
assignment: target '=' expr
target: NAME
```

```
if_statement: 'if' expr ':' statement
```

A super-abstract recursive descent parser for this language would define a function for each symbol that tries to call the functions corresponding to the alternatives. For example, for `statement` we'd have this function:

```
def statement():
    if assignment():
        return True
    if expr():
        return True
    if if_statement():
        return True
    return False
```

Of course this is too simplistic: it leaves out essential details about the parser's input and output.

Let's start with the input side. Classic parsers use a separate tokenizer which breaks the input (a text file or string) into a series of tokens, such as keywords, identifiers (names), numbers and operators. PEG parsers (like other modern parsers such as ANTLR) often unify tokenizing and parsing, but for my project I chose to keep the separate tokenizer.

Tokenizing Python is complicated enough that I don't want to reimplement it using PEG's formalism. For example, you have to keep track of indentation (this requires a stack inside the tokenizer), and the handling of newlines in Python is interesting (they are significant except inside matching brackets). The many types of string quotes also cause some complexity. In short, I have no beef with Python's existing tokenizer, so I want to keep it. (Aside: CPython has two tokenizers —an internal one used by the parser, written in C, and the standard library one, which is a faithful reimplementation in pure Python. This is helpful for my project.)

Classic tokenizers typically have a simple interface whereby you call a function, e.g. `get_token()`, which returns the next token in the input, consuming the input a few characters at a time. The `tokenize` module simplifies this even further: its basic API is a generator which yields one token at a time. Each token is a `TypeInfo`

object which has several fields, the most important ones of which indicate the *type* of the token (e.g. `NAME`, `NUMBER`, `STRING`), and its *string* value, meaning the string of characters comprising the token (e.g. `abc`, `42`, or `"hello world"`). There are also additional fields that give the coordinates of the token in the input file, which is useful for error reporting.

A special token type is `ENDMARKER`, which indicates that the end of the input file has been reached. The generator terminates if you ignore this and try to get the next token.

But I digress. How do we implement unlimited backtracking? Backtracking requires you to be able to remember a position in the source code and re-parse from that point. The tokenizer API doesn't allow us to reset its input pointer, but it's easy to capture the stream of tokens in an array and replay it from there, so that's what we do. (You could also do this using `itertools.tee()`, but based on warnings in the docs that's probably less efficient in our case.)

I suppose you could just first tokenize the entire input into a Python list and then use that as the parser input, but that would mean if there's an invalid token near the end of the file (such as a string with a missing closing quote) and there's also a syntax error earlier in the file, you would get an error message about the bad token first. I would find that a poor user experience, since the syntax error could actually be the root cause for the bad string. So my design tokenizes on demand, and the list becomes a lazy list.

The basic API is very simple. The `Tokenizer` object encapsulates the array of tokens and the position in that array. It has three basic methods:

- `get_token()` returns the next token, advancing the position in the array (reading another token from the source if we're at the end of the array);
- `mark()` returns the current position in the array;
- `reset(pos)` sets the position in the array (the argument *must* be something you got from `mark()`).

We add one convenience function, `peek_token()` which returns the next token without advancing the position.

Here, then, is the core of the `Tokenizer` class:

```
class Tokenizer:

    def __init__(self, tokengen):
        """Call with tokenize.generate_tokens(...)."""
        self.tokengen = tokengen
        self.tokens = []
        self.pos = 0

    def mark(self):
        return self.pos

    def reset(self, pos):
        self.pos = pos

    def get_token(self):
        token = self.peek_token()
        self.pos += 1
        return token

    def peek_token(self):
        if self.pos == len(self.tokens):
            self.tokens.append(next(self.tokengen))
        return self.tokens[self.pos]
```

Now, there are various things still missing (and the names of the methods and instance variables should really start with an underscore), but this will do as a sketch of the `Tokenizer` API.

The parser also needs to become a class, so that `statement()`, `expr()` and so on can become methods. The tokenizer becomes an instance variable, but we don't want the parsing methods to call `get_token()` directly — instead, we give the `Parser` class an `expect()` method which can succeed or fail just like a parsing method. The argument to `expect()` is the expected token — either a string (like `"+"`) or a token type (like `NAME`). I'll get to the return type after discussing the parser's output.

In my first sketch of the parser, the parsing functions just returned `True` or `False`. That's fine for theoretical computer science (where the question a parser answers is "is this a valid string in the language?") but not when you're building a parser — instead, we want the parser to create an AST. So let's just arrange it so that each parsing method returns a `Node` object on success, or `None` on failure.

The `Node` class can be super simple:

```
class Node:

    def __init__(self, type, children):
        self.type = type
        self.children = children
```

Here, `type` indicates what kind of AST node this is (e.g. an `"add"` node or an `"if"` node), and `children` is a list of nodes and tokens (instances of `TokenInfo`). This is enough for a compiler to generate code or do other analysis such as linting or static type checking, although in the future I'd like to change the way we represent the AST.

To fit into this scheme, the `expect()` method returns a `TokenInfo` object on success, and `None` on failure. To support backtracking, I wrap the tokenizer's `mark()` and `reset()` methods (no API change here). Here then is the infrastructure for the `Parser` class:

```
class Parser:

    def __init__(self, tokenizer):
        self.tokenizer = tokenizer

    def mark(self):
        return self.tokenizer.mark()

    def reset(self, pos):
        self.tokenizer.reset(pos)

    def expect(self, arg):
        token = self.tokenizer.peek_token()
        if token.type == arg or token.string == arg:
            return self.tokenizer.get_token()
        return None
```

Again, I've left out some details, but this works.

At this point I need to introduce an important requirement for parsing methods: a parsing method either returns a `Node`, positioning the tokenizer after the last

token of the grammar rule it recognized; or it returns `None`, *and then it leaves the tokenizer position unchanged*. If a parsing method reads several tokens and then decides to fail, it *must* restore the tokenizer's position. That's what `mark()` and `reset()` are for. Note that `expect()` also follows this rule.

So here's a sketch of the actual parser. Note that I am using Python 3.8's walrus operator (`:=`):

```
class ToyParser(Parser):

    def statement(self):
        if a := self.assignment():
            return a
        if e := self.expr():
            return e
        if i := self.if_statement():
            return i
        return None

    def expr(self):
        if t := self.term():
            pos = self.mark()
            if op := self.expect("+"):
                if e := self.expr():
                    return Node("add", [t, e])
            self.reset(pos)
            if op := self.expect("-"):
                if e := self.expr():
                    return Node("sub", [t, e])
            self.reset(pos)
            return t
        return None

    def term(self):
        # Very similar...

    def atom(self):
        if token := self.expect(NAME):
            return token
        if token := self.expect(NUMBER):
            return token
        pos = self.mark()
        if self.expect("("):
            if e := self.expr():
                if self.expect(")"):
                    return e
        self.reset(pos)
        return None
```

I've left some parsing methods as exercises for the reader — this is really more to give a flavor of what such a parser looks like, and eventually we'll generate code like this automatically from the grammar. Constants like `NAME` and `NUMBER` are imported from the `token` module in the standard library. (This ties us further to Python tokenization; there are ways around this that we should explore if we want to make a more general PEG parser generator.)

Also note that I cheated a bit: `expr` is left-recursive, but I made the parser *right*-recursive, because recursive-descent parsers don't work with left-recursive grammar rules. There's a fix for this, but it's still the topic of some academic research and I'd like to present it separately. Just realize that this version doesn't correspond 100% with the toy grammar.

The key things I want you to get at this point are:

- Grammar rules correspond to parser methods, and when a grammar rule references another grammar rule, its parsing method calls the other rule's parsing method.
- When multiple items make up an alternative, the parsing method calls the corresponding methods one after the other.
- When a grammar rule references a token, its parsing method calls `expect()`.
- When a parsing method successfully recognizes its grammar rule at the given input position, it returns a corresponding AST node; when it fails to recognize its grammar rule, it returns `None`.
- Parsing methods must explicitly reset the tokenizer position when they abandon a parse after having consumed one or more tokens (directly, or indirectly by calling another parsing method that succeeded). This applies when abandoning one alternative to try the next, and also when abandoning the parse altogether.

If all parsing methods abide by these rules, it's not necessary to use `mark()` and `reset()` around a single parsing method. You can prove this using induction.

As an aside, it's tempting to try to get rid of the explicit `mark()` and `reset()` calls

by using a context manager and a `with` statement, but this doesn't work: the `reset()` call shouldn't be called upon success! As a further fix you could try to use exceptions for control flow, so the context manager knows whether to reset the tokenizer (I think TatSu does something like this). For example, you could arrange for this to work:

```
    def statement(self):
        with self.alt():
            return self.assignment()
        with self.alt():
            return self.expr()
        with self.alt():
            return self.if_statement()
        raise ParsingFailure
```

In particular, the little ladder of `if` statements in `atom()` for recognizing a parenthesized expression could become:

```
        with self.alt():
            self.expect("(")
            e = self.expr()
            self.expect(")")
            return e
```

But I find this too "magical" — when reading such code you must stay aware that each parsing method (and `expect()`) may raise an exception, and that this exception is caught and ignored by the context manager in the `with` statement. That's pretty unusual, although definitely supported (by returning true from `__exit__`). Also, my ultimate goal is to generate C, not Python, and in C there's no `with` statement to alter the control flow.

Anyway, here are some topics for future installments:

- generating parsing code from the grammar;
- packrat parsing (memoization);
- EBNF features like `(x | y)`, `[x y ...]`, `x*`, `x+`;

- tracing (for debugging the parser or grammar);
- PEG features like lookahead and “cut”;
- left recursive rules;
- generating C code.

Python Parser

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

Generating a PEG Parser

Guido van Rossum · Follow

6 min read · Aug 5, 2019

Now that I've sketched the infrastructure for a parser and a simple hand-written parser in part 2, let's turn to generating a parser from a grammar, as promised. I'll also show how to implement packrat parsing using a `@memoize` decorator.

[This is part 3 of my PEG series. See the Series Overview for the rest.]

Last time we ended with a hand-written parser. With some limitations to the grammar, it's easy to generate such parsers automatically from the grammar. (We'll lift those limitations later.)

We need two things: something that reads the grammar, constructing a data structure representing the grammar rules; and something that takes that data structure and generates the parser. We also need boring glue that I'll omit.

So what we're creating here is a simple compiler-compiler. I'm simplifying the grammar notation a bit to the point where we just have rules and alternatives; this is actually sufficient for the toy grammar I've been using in the previous parts of the series:

```
statement: assignment | expr | if_statement
expr: expr '+' term | expr '-' term | term
term: term '*' atom | term '/' atom | atom
atom: NAME | NUMBER | '(' expr ')'
assignment: target '=' expr
target: NAME
if_statement: 'if' expr ':' statement
```

Using the full notation we can write up the grammar for grammar files:

```
grammar: rule+ ENDMARKER
rule: NAME ':' alternative ('|' alternative)* NEWLINE
alternative: item+
item: NAME | STRING
```

Using a fancy word, this is our first meta-grammar (a grammar for grammars), and our parser generator will be a meta-compiler (a compiler is a program that translates programs from one language into another; a meta-compiler is a compiler whose input is a grammar and whose output is a parser).

A simple way to represent the meta-grammar uses mostly built-in data types: the right-hand-side of a rule is just a list of lists of items, and the items can just be strings. (Hack: we can tell `NAME` and `STRING` apart by checking whether the first character is a quote.)

For rules I am using a simple class, `Rule`, and the whole grammar is then a list of `Rule` objects. Here's the `Rule` class, leaving out `__repr__` and `__eq__`:

```
class Rule:

    def __init__(self, name, alts):
        self.name = name
        self.alts = alts
```

And here's the `GrammarParser` class that uses it (for the `Parser` base class see my previous post):

```
class GrammarParser(Parser):

    def grammar(self):
        pos = self.mark()
        if rule := self.rule():
            rules = [rule]
            while rule := self.rule():
                rules.append(rule)
            if self.expect(ENDMARKER):
                return rules    # <------------- final result
```

```
            self.reset(pos)
            return None

        def rule(self):
            pos = self.mark()
            if name := self.expect(NAME):
                if self.expect(":"):
                    if alt := self.alternative():
                        alts = [alt]
                        apos = self.mark()
                        while (self.expect("|")
                               and (alt := self.alternative())):
                            alts.append(alt)
                            apos = self.mark()
                        self.reset(apos)
                        if self.expect(NEWLINE):
                            return Rule(name.string, alts)
            self.reset(pos)
            return None

        def alternative(self):
            items = []
            while item := self.item():
                items.append(item)
            return items

        def item(self):
            if name := self.expect(NAME):
                return name.string
            if string := self.expect(STRING):
                return string.string
            return None
```

Note the use of `ENDMARKER` to make sure there isn't anything left over after the last rule (which there might be if there's a typo in the grammar). I've placed a primitive arrow pointing to the place where the `grammar()` method returns a list of `Rule`s. The rest is very similar to the `ToyParser` class from the last episode, so I won't try to explain it. Just observe that `item()` returns a string, `alternative()` returns a list of strings, and the `alts` variable inside `rule()` collects a list of lists of strings. The `rule()` method then combines the rule name (a string) and `alts` into a `Rule` object.

If we let this code loose on a file containing our toy grammar, the `grammar()` method will return the following list of `Rule`s:

```
[
```

```
  Rule('statement', [['assignment'], ['expr'], ['if_statement']]),
  Rule('expr', [['term', "'+'", 'expr'],
                ['term', "'-'", 'term'],
                ['term']]),
  Rule('term', [['atom', "'*'", 'term'],
                ['atom', "'/'", 'atom'],
                ['atom']]),
  Rule('atom', [['NAME'], ['NUMBER'], ["'('", 'expr', "')'"]]),
  Rule('assignment', [['target', "'='", 'expr']]),
  Rule('target', [['NAME']]),
  Rule('if_statement', [["'if'", 'expr', "':'", 'statement']]),
]
```

Now that we have the parsing part of our meta-compiler, let's make the code generator. Together these form a rudimentary meta-compiler:

```
def generate_parser_class(rules):
    print(f"class ToyParser(Parser):")
    for rule in rules:
        print()
        print(f"    @memoize")
        print(f"    def {rule.name}(self):")
        print(f"        pos = self.mark()")
        for alt in rule.alts:
            items = []
            print(f"        if (True")
            for item in alt:
                if item[0] in ('"', "'"):
                    print(f"            and self.expect({item})")
                else:
                    var = item.lower()
                    if var in items:
                        var += str(len(items))
                    items.append(var)
                    if item.isupper():
                        print("            " +
                              f"and ({var} :=
self.expect({item}))")
                    else:
                        print(f"            " +
                              f"and ({var} := self.{item}())")
            print(f"        ):")
            print(f"            " +
              f"return Node({rule.name!r}, [{', '.join(items)}])")
            print(f"        self.reset(pos)")
        print(f"        return None")
```

This code is pretty ugly, but it works (kind of), and in a future episode I plan to

rewrite it anyway.

A few details of the code inside the `for alt in rule.alts` loop may require explanation: for each item in an alternative, we choose between three possibilities:

- if the item is a string literal, e.g. `'+'`, we generate `self.expect('+')`
- if the item is all upper case, e.g. `NAME`, we generate `(name := self.expect(NAME))`
- otherwise, e.g. if the item is `expr`, we generate `(expr := self.expr())`

If there are multiple items with the same item name in a single alternative (e.g. `term '-' term`), we append a digit to the second one. There's also a small bug here which I'll fix in a future episode.

Here's a bit of its output (the whole class would be very boring). Don't worry about the odd, redundant `if (True and ... )` idiom, which I am using so every generated condition can start with `and`; Python's bytecode compiler optimizes this out.

```
class ToyParser(Parser):

    @memoize
    def statement(self):
        pos = self.mark()
        if (True
            and (assignment := self.assignment())
        ):
            return Node('statement', [assignment])
        self.reset(pos)
        if (True
            and (expr := self.expr())
        ):
            return Node('statement', [expr])
        self.reset(pos)
        if (True
            and (if_statement := self.if_statement())
        ):
            return Node('statement', [if_statement])
        self.reset(pos)
        return None

    ...
```

Note the `@memoize` decorator: I smuggled that in so I can segue to a different topic: using memoization to make the generated parser fast enough.

Here's the `memoize()` function implementing this decorator:

```
def memoize(func):

    def memoize_wrapper(self, *args):
        pos = self.mark()
        memo = self.memos.get(pos)
        if memo is None:
            memo = self.memos[pos] = {}
        key = (func, args)
        if key in memo:
            res, endpos = memo[key]
            self.reset(endpos)
        else:
            res = func(self, *args)
            endpos = self.mark()
            memo[key] = res, endpos
        return res

    return memoize_wrapper
```

As is typical for a decorator, it contains a nested function that will replace (or *wrap*) the decorated function, for example the `statement()` method of the `ToyParser` class. Because the wrapped function is a method, the wrapper is also effectively a method: its first argument is named `self` and refers to the `ToyParser` instance on which the decorated method is called.

The wrapper caches the result of calling the parsing method *at every input position* — that's why it's called packrat parsing! The cache is a dict of dicts stored on the `Parser` instance. The outer cache key is the input position; I added `self.memos = {}` to `Parser.__init__()` to initialize it. The inner dicts are added as needed; their keys consist of the method and its arguments. (In the current design there are no arguments, but we could memoize `expect()`, which does have an argument, and there's little cost to this added generality.)

The result of a parsing method is represented as a tuple, since parsing methods really have two results: an explicit return value (for our generated parsers this is a `Node` representing the matched rule), and a new input position, which we get

from `self.mark()`. After calling the parsing method, we store both its return value (`res`) and the new input position (`endpos`) in the inner memoization dict. Upon further calls of the same parsing method with the same arguments at the same input position, we get both results from the cache, move the input position forward using `self.reset()`, and return the cached return value.

It is important to cache negative results too — in fact most calls to parsing methods will be negative results. In this case the return value is `None` and the input position is unchanged. You could add an `assert` to check this.

Note: A common memoization idiom in Python is to make the cache a local variable in the `memoize()` function. That won't do here: as I found out during a last-minute debug session, each `Parser` instance *must* have its own cache. However, you could get rid of the nested-dict design by using `(pos, func, args)` as the key.

Next week I will trace through the code to show how all this actually fits together when parsing an example program. I am still pulling my hair out over the best way to visualize how the tokenization buffer, the parser and the memoization cache work together. Maybe I'll manage to produce animated ASCII art instead of just trace logging output.

Python Parser

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

Visualizing PEG Parsing

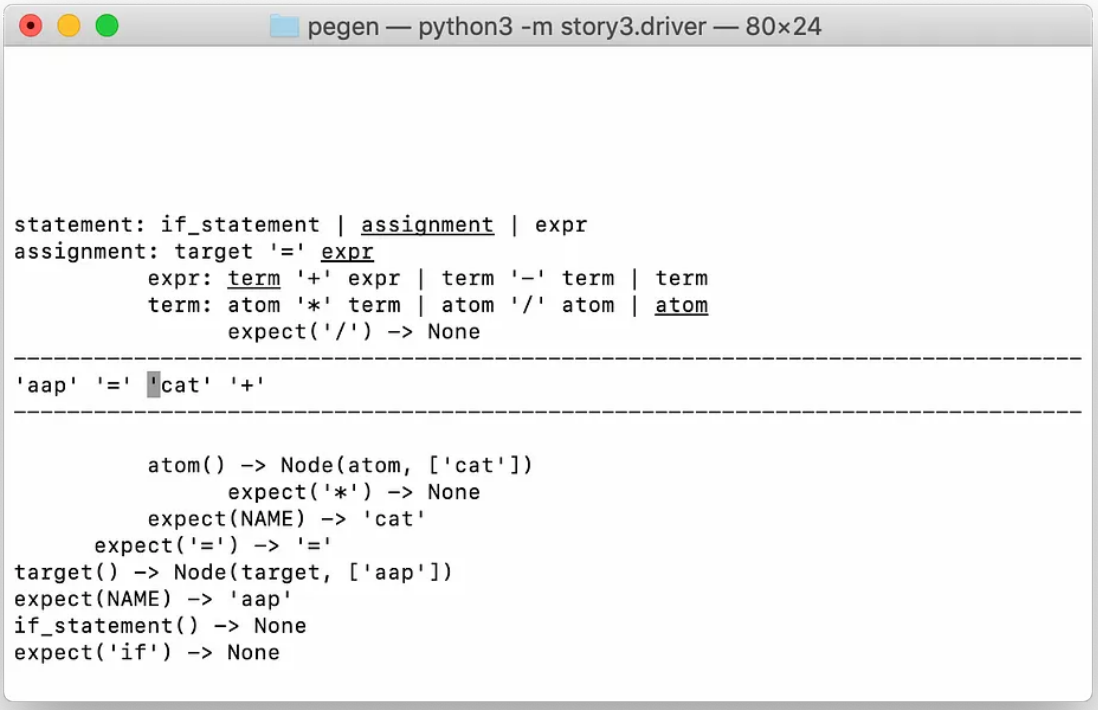
Guido van Rossum · Follow
4 min read · Aug 11, 2019

Last week I showed a simple PEG parser generator. This week I'll show what the generated parser actually does when it's parsing a program. I took a dive into the retro world of ASCII art, in particular a library named "curses" which is available in the Python standard library for Linux and Mac, and as an add-on for Windows.

[This is part 4 of my PEG series. See the Series Overview for the rest.]

Let's have a look at the visualization in progress. The screen is divided in three sections divided by that old standby of ASCII art, the line of hyphens:

- The top part shows the call stack of the parser, which as you may recall is a recursive descent parser with unlimited backtracking. I'll explain how to read this below.
- The single-line section in the middle shows the contents of the token buffer, with a cursor pointing at the token that's next under consideration.
- At the bottom we render the memoization cache used by the packrat parsing algorithm. Its entries are similar to some of the parser stack entries (the ones with results).



Parsing a toy program with a toy grammar

The main thing to know to read this chart is that the indentation of lines in the top and bottom parts corresponds to the token buffer.

- The first two lines (starting with `statement` and `assignment`) represent parsing method calls that haven't returned yet, and that were called when the token position was before the first token (`'aap'`).
- The next two lines (starting with `expr` and `term`) are aligned with the start of the token `'cat'`, which is where the corresponding parsing methods were called.
- The fifth and final line of the stack display shows an `expect('/')` call that is returning `None`. It was invoked at the position of the `'+'` token.

Indentation of items in the memoization cache also correspond to the token buffer position. For example, towards the bottom we find negative cache entries looking for the token `'if'` and the rule `if_statement` at the start of the token buffer. We also find successful cache entries for the token `'='` and for `NAME`

(specifically `'cat'`) corresponding to further input positions.

In both the parser stack display and the cache display, calls that have returned are shown as `function(args) -> result`. Sometimes the parser stack shows several returned methods — I did this to reduce the "jumpiness" of the display.

(Speaking of "jumpiness", the top of the parser stack display moves up when a call is added to the stack and it moves down when a call is popped from the stack. It seems our eyes don't have too much of a problem following such moves— at least mine don't. There's probably a significant part of our brain devoted to tracking objects that move. :-)

The cache is visualized as an LRU cache, with the most recently used cache item at the top, and less used items dropping off towards the bottom of the screen. (The prototype packrat parser I showed in previous posts does not use LRU, but it would likely be a good strategy to improve its memory use.)

Let's look at some more detail in the parsing stack display. The top four entries correspond to parsing methods that haven't returned yet, each line showing the line from the grammar. The underlined item is the one that spawned the next call.

In this case we see that we're on the second alternative for `statement`, namely `assignment`, and in that rule we're on the third item, namely `expr`. In the `expr` rule we're only at the first item of the first alternative (`term '+' expr`); and in the `term` rule we're at the final alternative (`atom`).

Below that we see the result that caused the second alternative (`atom '/' term`) to fail: `expect('/') -> None` indented with the `'+'` token. When we move the visualization forward we'll see it sink into the cache.

But surely you would rather see the animation for yourself! I've recorded the full parse of the canonical example program. You can also play with the code yourself, but note that this is a quick hack.

When you're viewing the recorded GIF, it may feel a bit disorienting that sometimes the next token is not shown (e.g. at the very start, the stack grows several entries before the token `'aap'` is revealed). This is exactly what the parser

sees though: the token buffer is filled lazily, and tokens are not scanned until the parser asks for them by calling `expect()`. Once the token is in the buffer, it stays there, even if the parser backtracks. Backtracking is shown by the cursor in the token buffer jumping left; the recording has numerous occurrences of this phenomenon. You can also observe cache fills in the recording, where the parser doesn't make additional recursive calls. (I ought to highlight when this happens, but I ran out of time.)

Python Parsing

Next week I'll develop the parser further, probably adding my take on left recursive grammar rules. (They're great!)

Acknowledgements: for the recording I used `ttygif` by Ilia Choly and `ttyrec` by Matthew Jording.

Member-only story

Left-recursive PEG Grammars

Guido van Rossum · Follow
10 min read · Aug 25, 2019

I've alluded to left-recursion as a stumbling block a few times, and it's time to tackle it. The basic problem is that with a recursive descent parser, left-recursion causes instant death by stack overflow.

[This is part 5 of my PEG series. See the Series Overview for the rest.]

Consider this hypothetical grammar rule:

```
expr: expr '+' term | term
```

If we were to naively translate this grammar fragment into a fragment of a recursive descent parser we'd get something like the following:

```
def expr():
    if expr() and expect('+') and term():
        return True
    if term():
        return True
    return False
```

So `expr()` starts by calling `expr()`, which starts by calling `expr()`, which starts by calling... This can only ends with a stack overflow, expressed as a `RecursionError` exception.

The traditional remedy is to rewrite the grammar. In the previous parts I've done just that. In fact, the above grammar would recognizes the same language if we rewrote that rule like this:

```
expr: term '+' expr | term
```

However, if we were to produce a parse tree, the shape of the parse tree would be different, and that would ruin things if we added a `'-'` operator to the grammar (since `a - (b - c)` is not the same as `(a - b) - c`). This is usually addressed by using more powerful PEG features, such as grouping and iteration, and we could rewrite the above rule as

```
expr: term ('+' term)*
```

This is in fact exactly what Python's current grammar uses to accommodate the *pgen* parser generator (which has the same issue with left-recursive rules).

There's still a slight problem with this though: since operators like `'+'` and `'-'` are (in Python) fundamentally binary, when we parse something like `a + b + c` we must loop over the parsing result (which is essentially the list `['a', '+', 'b', '+', 'c']`) to construct a left-recursive parse tree (which would be something like `[['a', '+', 'b'], '+', 'c']`).

The original left-recursive grammar expresses the desired associativity already, so it would be nice if we could generate a parser directly from that form. And we can! A fan pointed me to a nice trick, with a a mathematical proof attached to it, that was easy to implement. I'll try to explain it here.

Let's consider the example input `foo + bar + baz`. The parse tree we'd like to get out of this corresponds to `(foo + bar) + baz`. This requires three left-recursive

calls to `expr()`: one corresponding to the top-level `'+'` operator (i.e., the second one); one corresponding to the inner `'+'` operator (i.e., the first one); and one that chooses the second alternative (i.e., `term`).

Since I am bad at drawing actual diagrams using a computer, I'll show it here using ASCII art:

```
expr------------+------+
  |              \      \
expr--+------+   '+'   term
  |    \      \          |
expr   '+'   term        |
  |            |         |
term           |         |
  |            |         |
'foo'        'bar'     'baz'
```

The idea is that in the `expr()` function we wish for an "oracle" that will tell us whether to take the first alternative (i.e., calling `expr()` recursively) or the second (i.e., calling `term()`). In the first call to `expr()` the oracle should return true; in the second (recursive) call it should also return true, but in the third call it should return false, so there we can call `term()`. In code, this would look like:

```
def expr():
    if oracle() and expr() and expect('+') and term():
        return True
    if term():
        return True
    return False
```

How would we write such an oracle? Let's see... We could try to keep track of the number of (left-recursive) calls to `expr()` were on the call stack, and compare that to the number of `'+'` operators in the following expression. If the call stack is deeper than the number of operators, the oracle should return false. I'm almost tempted to implement this using `sys._getframe()`, but there is a better way: let's reverse the call stack!

The idea here is that we start with the call where the oracle returns false, and

save the result. This gives us `expr() -> term() -> 'foo'`. (It should return the parse tree for the initial `term`, i.e. `'foo'`. The code above just returns `True`, but in Part 2 I've already shown how to return a parse tree instead.) It's easy to write an oracle that does this, since it should just return false the first time it is called — no stack inspection or looking ahead required.

Then we call `expr()` again, and this time the oracle returns true, but instead of making a left-recursive call to `expr()` we substitute the saved result from the previous call. Lo and behold, the expected `'+'` operator and following `term` are also present, so this will give us the parse tree for `foo + bar`.

We repeat the procedure, and again things look bright: this time we get the parse tree for the full expression, and it's properly left-recursive (`(foo + bar) + baz`).

Then we repeat the procedure once more, and this time, while the oracle returns true and the saved result from the previous call is available, there is no further `'+'` operator, and the first alternative fails. So we try the second alternative, which succeeds, finding just the initial term (`'foo'`). This is a poor result compared to the previous call, so at this point we stop and *keep the longest parse* (i.e., `(foo + bar) + baz`).

To turn this into actual working code, I'm first going to rewrite the code slightly to combine the `oracle()` call with the left-recursive `expr()` call. Let's call it `oracle_expr()`. Code:

```
def expr():
    if oracle_expr() and expect('+') and term():
        return True
    if term():
        return True
    return False
```

Next we'll write a wrapper that implements the logic described above. It uses a global variable (don't worry, I'll get rid of that in a bit). The `oracle_expr()` function will read the global variable, and the wrapper manipulates it:

```
saved_result = None
```

```
def oracle_expr():
    if saved_result is None:
        return False
    return saved_result

def expr_wrapper():
    global saved_result
    saved_result = None
    parsed_length = 0
    while True:
        new_result = expr()
        if not new_result:
            break
        new_parsed_length = <calculate size of new_result>
        if new_parsed_length <= parsed_length:
            break
        saved_result = new_result
        parsed_length = new_parsed_length
    return saved_result
```

This is of course pathetic, but it represents the gist of the code, so let's try to develop it into something we can be proud of.

The crucial insight (which is my own, though I'm probably not the first to notice this) is that we can use the memoization cache instead of a global variable to save the result from one call to the next, and then we won't need the separate `oracle_expr()` function — we can generate a standard call to `expr()` regardless of whether it's in a left-recursive position or not.

To make this work, we need a separate `@memoize_left_rec` decorator that's only used for left-recursive rules. It serves as the `oracle_expr()` function, by pulling the saved value out of the memoization cache, and it contains the loop that calls `expr()` repeatedly as long as each new result covers a longer portion of the input than the previous. And of course, because the memoization cache handles caching separately per input position and per parsing method, it's unfazed by backtracking or multiple recursive rules (for example, in the toy grammar I've been using both `expr` and `term` are left-recursive).

Another nice property of the infrastructure I created in Part 3 is that it makes the check whether the new result is longer than the old result easy: the `mark()` method returns the index into the array of input tokens, so we can just use that instead of `parsed_length` above.

I'm leaving out the proof of why this algorithm always works, regardless of how crazy the grammar is. That's because I've not actually read that proof. I can see that it works for simple cases like `expr` in my toy grammar, and also for somewhat more complex cases (e.g. involving left-recursion hidden behind optional items in an alternative, or involving mutual recursion between multiple rules), but the most complicated situation I can think of in Python's grammar is still pretty tame, so I'm okay just trusting the theorem and the people who proved it.

So let's soldier on and show some real code.

First, the parser generator must detect which rules are left-recursive. This is a solved problem in graph theory. I'm not going to show the algorithm here, and in fact I'm going to simplify things even further and assume that the only left-recursive rules in the grammar are *directly* left-recursive, like `expr` in our toy grammar. The check for left-recursiveness then just needs to look for an alternative starting with the current rule's name. We can write it like this:

```
def is_left_recursive(rule):
    for alt in rule.alts:
        if alt[0] == rule.name:
            return True
    return False
```

Now we modify the parser generator so that for left-recursive rules it generates a different decorator. Recall that in Part 3 we decorated all parsing methods with `@memoize`. We now make one small change to the generator, so that for left-recursive rules we emit `@memoize_left_rec` instead, and then we implement the magic in the `memoize_left_rec` decorator. The rest of the generator and the support code needs no changes! (Though I did have to fool around some with the visualization code.)

For reference, here's the original `@memoize` decorator again, copied from Part 3). Remember that `self` is a `Parser` instance that has a `memo` attribute (initialized with an empty dictionary) and `mark()` and `reset()` methods that get and set the tokenizer's current position:

```
def memoize(func):

    def memoize_wrapper(self, *args):
        pos = self.mark()
        memo = self.memos.get(pos)
        if memo is None:
            memo = self.memos[pos] = {}

        key = (func, args)
        if key in memo:
            res, endpos = memo[key]
            self.reset(endpos)
        else:
            res = func(self, *args)
            endpos = self.mark()
            memo[key] = res, endpos
        return res

    return memoize_wrapper
```

The `@memoize` decorator remembers previous calls *per input position* — there is a separate `memo` dictionary for each position in the (lazy) array of input tokens. The first four lines of the `memoize_wrapper` function concern themselves with getting the right `memo` dictionary.

And here's `@memoize_left_rec`. Only the `else` branch is different from `@memoize` above:

```
    def memoize_left_rec(func):

    def memoize_left_rec_wrapper(self, *args):
        pos = self.mark()
        memo = self.memos.get(pos)
        if memo is None:
            memo = self.memos[pos] = {}

        key = (func, args)
        if key in memo:
            res, endpos = memo[key]
            self.reset(endpos)
        else:
            # Prime the cache with a failure.
            memo[key] = lastres, lastpos = None, pos

            # Loop until no longer parse is obtained.
            while True:
```

```
                self.reset(pos)
                res = func(self, *args)
                endpos = self.mark()
                if endpos <= lastpos:
                    break
                memo[key] = lastres, lastpos = res, endpos

            res = lastres
            self.reset(lastpos)

        return res

    return memoize_left_rec_wrapper
```

It will probably help to show the generated `expr()` method, so we can trace the flow between the decorator and the decorated method:

```
    @memoize_left_rec
    def expr(self):
        pos = self.mark()
        if ((expr := self.expr()) and
            self.expect('+') and
            (term := self.term())):
            return Node('expr', [expr, term])
        self.reset(pos)
        if term := self.term():
            return Node('term', [term])
        self.reset(pos)
        return None
```

Let's walk through parsing `foo + bar + baz`.

Whenever you call the decorated `expr()` function, the call is "intercepted" by the decorator, which looks for a previous call at the current position. On the first call it finds its way into the `else` branch, where it repeatedly calls the *undecorated* function. When the undecorated function calls `expr()`, this of course references the decorated version, so this recursive call is again intercepted. And here the recursion stops, because now the memo cache has a hit.

What happens next? The initial cache value comes from this line:

```
            # Prime the cache with a failure.
            memo[key] = lastres, lastpos = None, pos
```

This causes the decorated `expr()` to return `None`, at which point the first `if` in `expr()` fails (at `expr := self.expr()`). So we move on to the second `if`, which succeeds in recognizing a `term` (in our example `'foo'`) and `expr` returns a `Node` instance. Where does it return to? To the `while` loop in the decorator. This updates the memo cache with the new result (that `Node` instance) and then the next iteration starts.

The undecorated `expr()` gets called again, and this time the intercepted recursive call returns the newly cached `Node` instance (a term). This being a success, the call continues with `expect('+')`. That's a success again, since we're now at the first `'+'` operator. After this we look for a term, which is also successful (finding `'bar'`).

So now the bare `expr()`, having recognized `foo + bar` so far, returns to the `while` loop, which goes through the same motions: it updates the memo cache with the new (longer) result and starts the next iteration.

This game repeats itself once more. Again, the intercepted recursive `expr()` call retrieves the new result (this time `foo + bar`) from the cache, and we expect and find another `'+'` (the second one) and another `term` (`'baz'`). We construct a `Node` representing `(foo + bar) + baz` and return that to the `while` loop, which stuffs this into the memo cache and iterates once again.

But the next time around things go a bit differently. With the new result in hand we look for another `'+'`, but don't find one! So this `expr()` call falls back to its second alternative, and returns a measly `term`. When this bubbles up to the `while` loop, it finds to its disappointment that this result is shorter than the last, and it breaks, returning the longer result (`(foo + bar) + baz`) to the original call, which is whatever initiated the outer `expr()` call (for example, a `statement()` call — not shown here).

Python Parsing ends: we've successfully tamed left-recursion in a PEG(-ish)

parser. For next week I'm planning to discuss adding "actions" to the grammar, which let us customize the result returned by a parsing method for a given alternative (rather than always returning a `Node` instance).

If you want to play with the code, see the GitHub repo. (I also added visualization code for left-recursion, but I'm not super happy about it, so I'm not bothering to link to it here.)

Written by Guido van Rossum

3.3K Followers

Creator of Python

Adding Actions to a PEG Grammar

Guido van Rossum · Follow
3 min read · Aug 31, 2019

Grammars are better if you also can add (some) semantics inline with the grammar rules. In particular, for the Python parser I'm building I need to control the AST node returned by each alternative, since the AST format is given.

[This is part 6 of my PEG series. See the Series Overview for the rest.]

Many grammars use the convention of allowing *actions* to be added to rules, usually a block of code inside {curly braces}. More precisely, actions are attached to alternatives. The code in an action block is usually written in the language that the rest of the compiler is written in, like C, augmented with some facility for referring to items in the alternative. In Python's original *pgen* I did not add this feature, but for the new project I'd like to have it.

Here's how we do it for the simplified parser generator I'm developing for this series of blog posts.

The syntax for actions is usually like this:

```
rule: item item item { action 1 } | item item { action 2 }
```

Because this makes the grammar more verbose, parser generators typically allow splitting rules across lines, e.g.

```
rule: item item item { action 1 }
```

```
    | item item { action 2}
```

It makes the grammar parser a tad more complicated, but it's important for readability, so I'll use it.

An eternal question is when to execute the action block. In Yacc/Bison this is done once the rule is recognized by the parser, since there's no backtracking. Executing each action exactly once means that it's safe for actions to have global side effects (such as updating a symbol table or other compiler data structure).

In PEG parsers, with their unlimited backtracking, we have some choices:

- Delay all actions until all is parsed. This would not be useful for my purpose, since I want to build up an AST during the parse.
- Execute an action whenever its alternative is recognized, and require the action code to be idempotent (i.e. having the same effect no matter how many times it's executed). This means an action may be executed but its result ultimately discarded.
- Cache the action result, so each action is only executed the first time its alternative is recognized at a given position.

I'm going with the third option — we're caching anyway using the packrat algorithm, so we might as well cache action results too.

Regarding what goes inside the {curlies}, tradition is to use C code with a $-based convention for referencing items in the recognized alternative (e.g. `$1` to refer to the first item), and assignment to `$$` to indicate the result of the action. This sounds awfully old-fashioned to me (I have memories of using assignment to the function name in Algol-60 to specify a return value) so I'll go with something more Pythonic: inside the brackets you'll need to place a single expression, whose value is the action value, and references to items are simple names giving the text of the item. As an example, here's a simple calculator that can add and subtract numbers:

```
start: expr NEWLINE { expr }
expr: expr '+' term { expr + term }
```

```
    | expr '-' term { expr - term }
    | term { term }
term: NUMBER { float(number.string) }
```

When we run this with an input like `100 + 50 - 38 - 70`, it will compute the answer as it is recognizing the pieces, calculating `((100 + 50) - 38) - 70`, which of course comes out to `42`.

One little detail: in the action for `term`, the variable `number` holds a `TokenInfo` object, so the action must use its `.string` attribute to get the token in string form.

What do we do when an alternative has multiple occurrences of the same rule name? The parser generator gives each occurrence a unique name adding `1`, `2`, etc. to subsequent occurrences within the same alternative. For example:

```
factor: atom '**' atom { atom ** atom1 }
      | atom { atom }
```

The implementation is boring, so I invite you to check out the repo and play with the code. Try this:

```
python3.8 -m story5.driver story5/calc.txt -g
story5.calc.CalcParser
```

The visualization now allows you to go back and forth using the left and right arrow keys!

Python Parsing

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

Member-only story

A Meta-Grammar for PEG Parsers

Guido van Rossum · Follow
9 min read · Sep 9, 2019

This week we make the parser generator “self-hosted”, meaning the parser generator generates its own parser.

[This is part 7 of my PEG series. See the Series Overview for the rest.]

So we have a parser generator, a piece of which is a parser for grammars. We could call this a meta-parser. The meta-parser works similar to the generated parsers: `GrammarParser` inherits from `Parser`, and it uses the same `mark()` / `reset()` / `expect()` machinery. However, it is hand-written. But does it have to be?

It’s a tradition in compiler design to have the compiler written in the language that it compiles. I fondly remember that the Pascal compiler I used when I first learned to program was written in Pascal itself, GCC is written in C, and the Rust compiler is of course written in Rust.

How is this done? There’s a bootstrap: a compiler for a subset or earlier version of the language is written in a different language. (I recall that the initial Pascal compiler was written in FORTRAN!) Then a new compiler is written in the target language and compiled with the bootstrap compiler. Once the new compiler works well enough, the bootstrap compiler is scrapped, and each next version of the language or compiler is constrained by what the previous version of the

compiler can compile.

Let's do this for our meta-parser. We'll write a grammar for grammars (a meta-grammar) and then we'll generate a new meta-parser from that. Luckily I had planned this from the start, so it's a pretty simple exercise. The actions we added in the previous episode are an essential ingredient, because we don't want to have to change the generator — so we need to be able to generate a compatible data structure.

Here's a simplified version of the meta-grammar without actions:

```
start: rules ENDMARKER
rules: rule rules | rule
rule: NAME ":" alts NEWLINE
alts: alt "|" alts | alt
alt: items
items: item items | item
item: NAME | STRING
```

I'll show how to add actions from the bottom to the top. Recall from part 3 that there are `Rule` objects which have attributes `name` and `alts`. Originally, `alts` was just a list of lists of strings (the outer list representing the alternatives, the inner lists representing the items of each alternative), but in order to add actions I changed things so that alternatives are represented by `Alt` objects with attributes `items` and `action`. Items are still represented by plain strings. For `item` we get:

```
item: NAME { name.string } | STRING { string.string }
```

This requires a little bit of explanation: when the parser processes a token, it returns a `TokenInfo` object, which has attributes `type`, `string` and others. We don't want the generator to have to deal with `TokenInfo` objects, so the actions here extract the string from the token. Note that for all-uppercase tokens like `NAME` the generated parser uses the lowercased version (here `name`) as the variable name.

Next up is `items`, which must return a list of strings:

```
items: item items { [item] + items } | item { [item] }
```

I am using right-recursive rules here, so we're not depending on the left-recursion handling added in part 5. (Why not? It's always a good idea to keep things as simple as possible, and this grammar wouldn't gain much clarity from left-recursion.) Note that a single `item` is listified, but the recursive `items` isn't, since it already is a list.

The `alt` rule exists to construct the `Alt` object:

```
alt: items { Alt(items) }
```

I'll skip the actions for `rules` and `start` since they follow the same pattern.

There are two open issues, however. First, how does the generated code where to find the `Rule` and `Alt` classes? We need to add some `import` statements to the generated code for this purpose. The simplest approach would be to pass a flag to the generator that says "this is the meta-grammar" and let the generator spit out the extra `import`s at the top of the generated program. But now that we have actions, many other parsers will also want to customize their imports, so why don't we see if we can add a more general feature.

There are many ways to skin this cat. A simple and general mechanism is to add a section of "variable definitions" at the top of the grammar, and let the generator use those variables to control various aspects of the generated code. I chose to use the `@` character to start a variable definition, followed by the variable name (a `NAME`) and the value (a `STRING`). For example, we can put the following at the top of the meta-grammar:

```
@subheader "from grammar import Rule, Alt"
```

The parser generator will print the value of the `subheader` variable after the

standard imports which are always printed (e.g. to import `memoize`). If you want multiple `import`s, you can use a triple-quoted string in the variable declaration, e.g.

```
@subheader """
from token import OP
from grammar import Rule, Alt
"""
```

This is easy to add to the meta-grammar: we replace the `start` rule with this:

```
start: metas rules ENDMARKER | rules ENDMARKER
metas: meta metas | meta
meta: "@" NAME STRING NEWLINE
```

(I can't recall why I called these "metas" but that's the name I picked when I wrote the code, and I'm sticking to it. :-)

We have to add this to the bootstrap meta-parser as well. Now that a grammar is not just a list of Rules, let's add a Grammar object, with attributes `metas` and `rules`. We can put in the following actions:

```
start: metas rules ENDMARKER { Grammar(rules, metas) }
     | rules ENDMARKER { Grammar(rules, []) }
metas: meta metas { [meta] + metas }
     | meta { [meta] }
meta: "@" NAME STRING { (name.string, eval(string.string)) }
```

(Note that `meta` returns a tuple; and note that it uses `eval()` to process the string quotes.)

Speaking of actions, I left out actions from the rule for `alt`! The reason is that these are a bit messy. But I can't put it off any longer, so here goes:

```
alt: items action { Alt(items, action) }
   | items { Alt(items, None) }
```

```
action: "{" stuffs "}" { stuffs }
stuffs: stuff stuffs { stuff + " " + stuffs }
      | stuff { stuff }
stuff: "{" stuffs "}" { "{" + stuffs + "}" }
     | NAME { name.string }
     | NUMBER { number.string }
     | STRING { string.string }
     | OP { None if op.string in ("{", "}") else op.string }
```

The mess is caused by my desire to allow arbitrary Python code between the curly braces delineating the action, with an extra wrinkle to allow matching pairs of braces to be nested inside. For this purpose we use the special token `OP`, which our tokenizer generate for all punctuation that is recognized by Python (returning a single token with type `OP` for multi-character operators such as `<=` or `**`). The only other tokens that can legitimately occur in Python expressions are names, numbers, and strings. So the "stuff" between the outermost curly braces of an action would seem to be covered by a repetition of `NAME | NUMBER | STRING | OP`.

Alas, that doesn't work, because `OP` also matches curly braces, and since a PEG parser is always greedy, this would gobble up the closing brace, and we'd never see the end of the action. Therefore we add a little tweak to the generated parsers, allowing an action cause an alternative to fail by returning `None`. I don't know whether this is a standard thing in other PEG parsers — I came up with this on the spot when I had to solve the problem of recognizing the closing brace (even apart from nesting pairs). It seems to work well, and I think it fits in the general philosophy of PEG parsing. It could be seen as a special form of lookahead (which I will cover below).

Using this little tweak we can make the match on `OP` fail when it's a curly brace, so that it is available for matching by `stuff` and `action`.

With these things in place, the meta-grammar can be parsed by the bootstrap meta-parser, and the generator can turn it into a new meta-parser, which can parse itself. And importantly, the new meta-parser can still parse the same meta-grammar. If we compile the meta-grammar with the new meta-compiler, the output is the same: this proves that the generated meta-parser works correctly.

Here's the complete meta-grammar with actions. It can parse itself as long as you join the long lines together:

```
@subheader """
from grammar import Grammar, Rule, Alt
from token import OP
"""

start: metas rules ENDMARKER { Grammar(rules, metas) }
     | rules ENDMARKER { Grammar(rules, []) }

metas: meta metas { [meta] + metas }
     | meta { [meta] }
meta: "@" NAME STRING NEWLINE { (name.string, eval(string.string))
}

rules: rule rules { [rule] + rules }
     | rule { [rule] }
rule: NAME ":" alts NEWLINE { Rule(name.string, alts) }

alts: alt "|" alts { [alt] + alts }
    | alt { [alt] }
alt: items action { Alt(items, action) }
   | items { Alt(items, None) }

items: item items { [item] + items }
     | item { [item] }
item: NAME { name.string }
    | STRING { string.string }

action: "{" stuffs "}" { stuffs }

stuffs: stuff stuffs { stuff + " " + stuffs }
      | stuff { stuff }
stuff: "{" stuffs "}" { "{" + stuffs + "}" }
     | NAME { name.string }
     | NUMBER { number.string }
     | STRING { string.string }
     | OP { None if op.string in ("{", "}") else op.string }
```

Now that we've got a working meta-grammar, we're almost ready to make some improvements.

But first, there's a little annoyance to deal with: blank lines! It turns out that the stdlib `tokenize` module produces extra tokens to track non-significant newlines and comments. For the former it generates a `NL` token, for the latter a `COMMENT` token. Rather than incorporate these in the grammar (I've tried, it's no fun!), there's a very simple bit of code we can add to our tokenizer class that filters these out. Here's the improved `peek_token` method:

```
def peek_token(self):
        if self.pos == len(self.tokens):
            while True:
                token = next(self.tokengen)
                if token.type in (NL, COMMENT):
                    continue
                break
            self.tokens.append(token)
            self.report()
        return self.tokens[self.pos]
```

This filters out `NL` and `COMMENT` tokens completely, so we don't have to worry about them in the grammar any more.

Finally let's make an improvement to the meta-grammar! The thing I want to do is purely cosmetic: I don't like being forced to put all alternatives on the same line. The meta-grammar I showed above doesn't actually parse itself, because of things like this:

```
start: metas rules ENDMARKER { Grammar(rules, metas) }
     | rules ENDMARKER { Grammar(rules, []) }
```

This is because the tokenizer produces a `NEWLINE` token at the end of the first line, and at that point the meta-parser will think that's the end of the rule. Moreover, that `NEWLINE` will be followed by an `INDENT` token, because the next line is indented. There will also be a `DEDENT` token before the start of the next rule.

Here's how to deal with all that. To understand the behavior of the `tokenize` module, we can look at the sequence of tokens generated for indented blocks by running the `tokenize` module as a script and feeding it some text:

```
$ python -m tokenize
foo bar
    baz
    dah
dum
^D
```

We find that this produces the following sequence of tokens (I've simplified the output from the above run a bit):

```
NAME     'foo'
NAME     'bar'
NEWLINE
INDENT
NAME     'baz'
NEWLINE
NAME     'dah'
NEWLINE
DEDENT
NAME     'dum'
NEWLINE
```

So that means an indented group of lines is delineated by `INDENT` and `DEDENT` tokens. We can now rewrite the meta-grammar rule for `rule` as follows:

```
rule: NAME ":" alts NEWLINE INDENT more_alts DEDENT {
        Rule(name.string, alts + more_alts) }
    | NAME ":" alts NEWLINE { Rule(name.string, alts) }
    | NAME ":" NEWLINE INDENT more_alts DEDENT {
        Rule(name.string, more_alts) }

more_alts: "|" alts NEWLINE more_alts { alts + more_alts }
         | "|" alts NEWLINE { alts }
```

(I'm breaking the actions up across lines so they fit in Medium's narrow page — this is possible because the tokenizer ignores line breaks inside matching braces.)

The beauty of this is that we don't even need to change the generator: the data structure produced by this improved meta-grammar is the same as before. Also note the third alternative for `rule`: this lets us write

```
start:
    | metas rules ENDMARKER { Grammar(rules, metas) }
    | rules ENDMARKER { Grammar(rules, []) }
```

which some people find cleaner than the version I showed earlier. It's easy to [illegible] we don't have to argue about style.

Python Parser

In the next post I will show how I implemented various PEG features like optional items, repetitions and lookaheads. (To be fair, I was planning to put that in this part, but this one has gone on too long already, so I'm splitting it into two pieces.)

Member-only story

Implementing PEG Features

Guido van Rossum · Follow
7 min read · Sep 13, 2019

After making my PEG parser generator self-hosted in the last post, I'm now ready to show how to implement various other PEG features.

[This is part 8 of my PEG series. See the Series Overview for the rest.]

We'll cover the following PEG features:

- Named items: `NAME=item` (the name can be used in an action)
- Lookaheads: `&item` (positive) and `!item` (negative)
- Grouping items in parentheses: `(item item ...)`
- Optional items: `[item item ...]` and `item?`
- Repeated items: `item*` (zero or more) and `item+` (one or more)

. . .

Let's start with named items. These are handy when we have multiple items in one alternative that refer to the same rule, like this:

```
expr: term '+' term
```

Our generator allows us to refer to the second `term` by appending a digit `1` to the variable name, so we could just write the action like this:

```
expr: term '+' term { term + term1 }
```

But this is not to everyone's liking, and I would personally prefer to be able to write something like this:

```
expr: left=term '+' right=term { left + right }
```

It's easy to support this in the meta-grammar by changing the rule for `item` as follows:

```
item:
    | NAME = atom { NamedItem(name.string, atom) }
    | atom { atom }
atom:
    | NAME { name.string }
    | STRING { string.string }
```

(Where `atom` is just the old `item`.)

This requires us to add a definition of the `NamedItem` class to `grammar.py`, which is another one of those data classes you've heard so much about — this one having attributes `name` and `item`.

We also need to make some small changes to the code generator, which I'll leave as an exercise for the reader (or you can check my repo :-). The generated code will now assign the result of matching each item to a variable with the indicated name, rather than a name derived from the item's rule name. This also works for

items that are tokens (either of the form `NAME` or string literals like `':='`).

Next up are lookaheads. You may be familiar with these from regular expressions. A lookahead can reject or accept the current alternative based on what follows it, without the input pointer further. A positive lookahead accepts if its item is recognized; a negative lookahead accepts if its item is *not* recognized.

Lookaheads can actually be used as a cleaner way to address the mess with parsing actions I wrote about in the previous episode: Rather than allowing actions to reject an already accepted alternative by returning `None`, we can prefix the `OP` item with a lookahead that ensures it's not `"}"`. The old rule for stuff ended like this:

```
    | OP { None if op.string in ("{", "}") else op.string }
```

The new version looks like this:

```
    | !"}" OP { op.string }
```

This moves the special-casing of curly braces from the action to the grammar, where it properly belongs. We don't need to check for `"{"`, since it matches an earlier alternative (this was true for the old version too, actually, but I forgot :-).

We add the grammar for lookaheads to the rule for `item`, like so:

```
item:
    | NAME = atom { NamedItem(name.string, atom) }
    | atom { atom }
    | "&" atom { Lookahead(atom, True) }
    | "!" atom { Lookahead(atom, False) }
```

Again, we have to add a `Lookahead` data class to `grammar.py` (and import it in `@subheader`!), and twiddle the generator a bit. The generated code uses the following helper method:

```
    def lookahead(self, positive, func, *args):
        mark = self.mark()
        ok = func(*args) is not None
        self.reset(mark)
        return ok == positive
```

In our case, the generated code for this alternative looks like this:

```
        if (True
            and self.lookahead(False, self.expect, "}")
            and (op := self.expect(OP))
        ):
            return op . string
```

As the grammar fragment above suggests, lookaheads cannot be named. This could easily be changed, but I can't think of anything useful to do with the value; also, for negative lookaheads, the value would always be `None`.

. . .

Next let's add parenthesized groups. The best place to add these to the meta-grammar is in the rule for `atom`:

```
atom:
    | NAME { name.string }
    | STRING { string.string }
    | "(" alts ")" { self.synthetic_rule(alts) }
```

The first two alternatives are unchanged. The action for the new alternative uses a hack (whose implementation will remain unexplained) that allows the meta-parser to add new rules to the grammar on the fly. This helper function (defined on the meta-parser) returns the name of the new `Rule` object. This name is a fixed prefix followed by a number to make it unique, e.g. `_synthetic_rule_1`.

You might wonder what happens if the synthetic rule ends up being abandoned

due to the meta-parser backtracking over it. I don't see where the current meta-grammar would allow this without failing, but it's pretty safe — at worst there's an unused rule in the grammar. And due to the memoization cache, the same action will never be executed twice for the same input position, so that's not a problem either (but even if it were, at worst we'd have a dead rule).

Using `alts` inside the parentheses means that we can use the vertical bar in the group, which is one of the purposes of grouping. For example, if we wanted to make sure our lookahead-based solution for the "action mess" would not accidentally match `{`, we could update the negative lookahead like this:

```
    | !("{" | "}") OP { op.string }
```

Even better, groups can also contain actions. This wouldn't be useful in the "action mess" solution, but there are other cases where it's useful. And because we generate a synthetic rule anyway, implementing it doesn't require any extra work (beyond implementing `synthetic_rule` :-).

. . .

On to optional items. Like I did in the old *pgen*, I am using square brackets to indicate an optional group. This is often useful, for example a grammar rule describing a Python `for` loop might use this to indicate that there is an optional `else` clause. The grammar again can be added to the rule for `atom`, like so:

```
atom:
    | NAME { name.string }
    | STRING { string.string }
    | "(" alts ")" { self.synthetic_rule(alts) }
    | "[" alts "]" { Maybe(self.synthetic_rule(alts)) }
```

Here `Maybe` is another data class, with a single `item` attribute. We modify the code generator to generate code that preserves the value returned by the synthetic parsing function for the contained alternatives, but doesn't fail if that value is `None`. We do this by essentially adding `or True` to to the code, like this

code for `[term]`:

```
if (True
    and ((term := self.term()) or True)
):
    return term
```

. . .

Moving on to repetitions, another useful PEG feature (the notation is borrowed from regular expressions and is also used in *pgen*). There are two forms: appending a star to an atom means "zero or more repetitions" while appending a plus sign means "one or more repetitions". For various reasons I ended up rewriting the grammar rules for `item` and `atom`, inserting an intermediate rule that I named `molecule`:

```
item:
    | NAME '=' molecule { NamedItem(name.string, molecule) }
    | "&" atom { Lookahead(atom) }
    | "!" atom { Lookahead(atom, False) }

    | molecule { molecule }

molecule:
    | atom "?" { Maybe(atom) }
    | atom "*" { Loop(atom, False) }
    | atom "+" { Loop(atom, True) }
    | atom { atom }
    | "[" alts "]" { Maybe(self.synthetic_rule(alts)) }

atom:
    | NAME { name.string }
    | STRING {string.string }
    | "(" alts ")" { self.synthetic_rule(alts) }
```

Observe that this introduces an alternative syntax for optionals (`atom?`) which requires no additional implementation effort since it's just another way to create a `Maybe` node.

The rule refactoring here was needed because I don't want to easily allow anomalies like optional repetitions (since that's just a zero-or-more repetition),

repeated repetitions (the inner one would gobble up all matches since PEG always uses eager matching), or repeated optionals (which would stop the parser dead if the optional item doesn't match). Note that this isn't a 100% solution, since you can still write something like `(foo?)*`. The parser generator will have to add a check for this situation, but that's beyond the scope of this series.

The `Loop` data class has two attributes, `item` and `nonempty`. The generated code uses a helper method on the generated parser, `loop()`, which has a similar signature as `lookahead()` shown before:

```
    def loop(self, nonempty, func, *args):
        mark = self.mark()
        nodes = []
        while node := func(*args) is not None:
            nodes.append(node)
        if len(nodes) >= nonempty:
            return nodes
        self.reset(mark)
        return None
```

If nonempty is `False` (meaning the grammar used `*`) this will never fail — instead it will return an empty list when it sees no occurrences of the item. In order to make this work we make the parser generator emit `is not None` checks rather than the more lenient "truthy" checks I showed in a previous post — a "truthy" check would return `False` if an empty list was recognized.

. . .

And that's all for today! I was going to discuss the "cut" operator (`~`) present in TatSu, but I haven't encountered a real use case for it yet, so I wouldn't be the best person to explain it — the TatSu docs only give a toy example that doesn't motivate me much. I haven't found it in other PEG parser generators either, so maybe it's a TatSu invention. Maybe I'll explain it in the future. (In the meantime I did implement it, in case it's ever useful. :-)

I think the next episode will be about my experience writing a PEG grammar that can parse all of Python. This is mostly how I spent the Python core developer sprint that was held this week in London, with logistical support from Bloomberg

and financial support from the PSF and some attendees' employers (e.g. Dropbox paid for my hotel and airfare). Special thanks go to Emily Morehouse and Pablo Galindo Salgado, who were very helpful writing tools and tests. Next up for *that* project is writing a performance benchmark, and then we're going to add actions to this grammar so it can create AST trees that can be compiled by the CPython bytecode compiler. Exciting times!

Python Parsing

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

Member-only story

PEG at the Core Developer Sprint

Guido van Rossum · Follow

9 min read · Sep 23, 2019

This week I'm not showing any new code for the parser generator I've described it the previous parts. Instead, I'll try to describe what I did at the Core Developer Sprint last week before it all evaporates from my memory. Most of this relates to in PEG one way or another. Then I'll show some code anyway, because I like to talk about code, and it roughly shows the path I see to a PEG-based parser for CPython 3.9.

[This is part 9 of my PEG series. See the Series Overview for the rest.]

Every year for the past four years a bunch of Python core developers get together for a week-long sprint at an exotic location. These sprints are sponsored by the PSF as well as by the company hosting the sprint. The first two years were hosted by Facebook in Mountain View, last year was Microsoft's turn in Bellevue, and this year's sprint was hosted by Bloomberg in London. (I have to say that Bloomberg's office looks pretty cool.) Kudos to core dev Pablo Galindo Salgado for organizing!

Over 30 core devs attended this time, as well as two hopefuls. People worked on many different projects, from 3.8 release blockers to new PEPs. I hope that the PSF will publish a blog post with our accomplishments. One of the highlights was that the number of open PRs was pushed below 1000, merging over 100 PRs that had been waiting for some love from a core dev reviewer. There was even a

friendly contest with a leaderboard showing the top 10 attendees who merged the most pull requests *submitted by others*.

For me, the main reason to attend events like this is always meeting people with whom I collaborate online all year long but whom I only get to see once or twice a year. This sprint was in Europe, so we saw a slightly different mix of core devs, and that was particularly important. Regardless, I also did quite a bit of hacking, and that's what I'll focus on below.

Most of my coding time at the sprint I worked with Pablo and Emily Morehouse on the PEG-based parser generator that I hope will some day replace the current *pgen* based parser generator. This isn't the same code as the generator I've been blogging about, but it's pretty similar. Pablo had contributed to this version already.

The first day of the sprint, Monday, I spent mostly on part 7 and 8 of this series. Originally I planned to publish all the material in a single post, but at the end of the day I wasn't quite finished, so I cut things in two and posted the first half, on making a meta-grammar, as part 7. On Friday after dinner I finally found a little time to polish off part 8, but I still had to skip the "cut" operator because I realized I don't have a compelling example.

On Tuesday I started working on developing a PEG grammar for Python. PEG grammars, even before you add actions, are closer to code than to abstract specifications, and we realized that we needed to test the grammar under development against real Python code. So while I hacked on the grammar, Emily wrote a bulk testing script. Once that was completed, my workflow was roughly as follows:

1. Run the bulk testing script over some directory with Python code
2. Investigate the first issue reported by the script
3. Tweak the grammar to deal with that issue
4. Repeat until there are no issues left
5. Repeat with a different directory

I started with the pegen project's own code as the target. Eventually my grammar

could parse all of the Python constructs used in pegen, and I moved on to parts of the standard library, first focusing on `Lib/test`, then `Lib/asyncio`, and finally `Lib`, i.e. effectively the entire standard library (or at least the part written in Python). By the end of the week I could declare victory: the only files in the standard library that the new parser rejects are a few files with bad encodings, which exist purely to as test cases to verify that bad encodings are rejected, and some Python 2 code used as test cases for *lib2to3*.

Since then I've added some timing code to Emily's script, and it looks like the new PEG-based Python parser is a tad faster than the old *pgen* parser. That doesn't mean it will be all uphill from here though! The grammar defines more than 100 rules and is over 160 lines long. To make it produce ASTs, we will have to add an action to every rule (see part 6).

. . .

I did an earlier experiment, and adding actions will probably double or triple the size of the grammar file. Here's the grammar from that experiment:

```
start[mod_ty]: a=stmt* ENDMARKER{ Module(a, NULL, p->arena) }
stmt[stmt_ty]: compound_stmt | simple_stmt
compound_stmt[stmt_ty]: pass_stmt | if_stmt

pass_stmt[stmt_ty]: a='pass' NEWLINE { _Py_Pass(EXTRA(a, a)) }

if_stmt[stmt_ty]:
    | 'if' c=expr ':' t=suite e=[else_clause] {
          _Py_If(c, t, e, EXTRA(c, c)) }
else_clause[asdl_seq*]:
    | 'elif' c=expr ':' t=suite e=[else_clause] {
          singleton_seq(p, _Py_If(c, t, e, EXTRA(c, c))) }
    | 'else' ':' s=suite { s }

suite[asdl_seq*]:
    | a=simple_stmt { singleton_seq(p, a) }
    | NEWLINE INDENT b=stmt+ DEDENT { b }

simple_stmt[stmt_ty]: a=expr_stmt NEWLINE { a }

expr_stmt[stmt_ty]: a=expr { _Py_Expr(a, EXTRA(a, a)) }

expr[expr_ty]:
    | l=expr '+' r=term { _Py_BinOp(l, Add, r, EXTRA(l, r)) }
    | l=expr '-' r=term { _Py_BinOp(l, Sub, r, EXTRA(l, r)) }
    | term
term[expr_ty]:
```

```
    | l=term '*' r=factor { _Py_BinOp(l, Mult, r, EXTRA(l, r)) }
    | l=term '/' r=factor { _Py_BinOp(l, Div, r, EXTRA(l, r)) }
    | factor
factor[expr_ty]:
    | l=primary '**' r=factor { _Py_BinOp(l, Pow, r, EXTRA(l, r))
}
    | primary
primary[expr_ty]:
    | f=primary '(' e=expr ')' {
          _Py_Call(f, singleton_seq(p, e), NULL, EXTRA(f, e)) }
    | atom
atom[expr_ty]:
    | '(' e=expr ')' { e }
    | NAME
    | NUMBER
    | STRING
```

There are a bunch of things here that I should explain.

- The actions contain C code. Like for the Python generator from part 6, each action is a single C expression.
- The things in square brackets immediately after the rule names are return types for the corresponding rule methods. For example, `atom[expr_ty]` means that the rule method for `atom` returns an `expr_ty`. If you look through `Include/Python-ast.h` in the CPython repo, you'll see that this type is the struct used to represent expressions in the (internal) AST.
- If an alternative has exactly one item, the action can be omitted. The default action is then to return the AST node returned by that item. If there are multiple items in an alternative, the default action is different and useless.
- The C code needs help from a prelude spit out by the code generator. For example, it assumes that a bunch of CPython include files have been included (so that e.g. `expr_ty` is visible), and some additional definitions must exist.
- The variable `p` holds a pointer to the `Parser` structure used by the generated parser. (And yes, this means that you better not name any item in rule `p` — the generated C code won't compile!)
- `EXTRA(node1, node2)` is a macro that expands to a bunch of extra arguments that need to be passed to every AST construction function. This saves a lot of typing in the action — the alternative would be to write out the expressions

that give the starting and ending line number and column offset, and a pointer to the *arena* used for allocation. (AST nodes aren't Python objects and use a more efficient allocation scheme.)

- Due to some interesting behavior of the C preprocessor, this handy `EXTRA()` macro unfortunately means we cannot use the macros for AST node creation — we have to use the underlying functions. That's why you see for example `_Py_Binop(...)` rather than `Binop(...)`. In the future I may solve this differently.

- For repeated items (`foo*` or `foo+`) the code generator produces an auxiliary rule whose type is `asdl_seq*`. This is a data structure used in the AST to represent repetitions. In a few places we need to construct one of these containing a single element, and for that we've defined a helper function, `singleton_seq()`.

If some of this sounds a little dodgy, I'm the first to admit it. It's a prototype, and its main point is to show that it is, in principle, possible to generate a working AST using a parser generated from a PEG grammar. All this works without any changes to the existing tokenizer or bytecode compiler. The prototype can compile simple expressions and `if` statements, and the resulting AST can be compiled to bytecode and executed, and it seems to work.

. . .

Other stuff I did at the core development sprint:

- Convinced Łukasz Langa to change PEP 585 (his proposal for the future of type hints) to focus on generics, instead of the grab-bag of ideas it was before. The new PEP looks much better, and at a public typing meetup a few days ago where representatives of several type checkers for Python (mypy, pytype and Pyre) were present, it received general nods of approval. (Which isn't the same as approval by the Steering Council!)

- Helped Yury Selivanov with the API design for a *frozenmap* type he's interested in adding to the stdlib. Several other sprint attendees also contributed to the design — I believe we left the sprint with several whiteboards full of examples and API fragments. The result is PEP 603, and it is being vigorously debated

currently. (One note: the implementation of the proposed data type already exists in CPython, as part of the implementation of PEP 567, the `contextvars` module. It's a very interesting data structure, Hash Array Mapped Trie, that combines a hash table with a trie.)

- Yury, as always full of ideas, was also working on exception groups, an idea from Trio that he wants to introduce into asyncio in Python 3.9. Last I looked there was no PEP yet, but I do remember another whiteboard full of diagrams. (I wish I could find a link to the Trio concept, but many things have "cute" names in Trio, and I failed.)

- We vigorously debated Łukasz's proposal for a shorter release cycle for Python. This resulted in PEP 602, where he proposes to do releases annually, in October. (There's a reason for October: it has to do with the typical schedule of Python conferences and core development sprints.) This proposal continues to be debated *very* vigorously. There are at least two counter-proposals: PEP 598 by Nick Coghlan proposes biannual releases while allowing new features in point releases, and Steve Dower would like to see biannual releases without the latter complication (but hasn't written it up in PEP form yet).

- The three Steering Council members who attended the sprint (Brett Cannon, Carol Willing and myself) got together and discussed our vision for the future of Python core development. (I don't want to say much about the vision, which we expect to reveal at the next US PyCon, except that we will likely propose to go fundraising so the PSF can hire a few developers to assist and accelerate core development.)

- I had an interesting dinner conversation with Joannah Nanjekye, one of the hopefuls attending. She told the story of how she heard about the internet as an 8-year old, took her slightly younger brother to an internet cafe while their mom was out working, discovered Google and email, and came back every next day of that first week.

- The alcoholic highlight of the week was the pair of Zombie Apocalypse cocktails a few of us ordered at a bar named the Alchemist. Served in a 2000-ml Erlenmeyer flask and involving a lot of fake smoke generated by pouring dry ice over the usual alcoholic mixture, each Zombie Apocalypse serves four

people. As a drink it was quite forgettable.

- Friday night Lisa Roach led us to a nice Indian restaurant near her hotel. It was four Underground stops away, which was quite an adventure (it was rush hour and we nearly lost Christian Heimes a few times). The food was worth the trip!

- At some point we took a group photo. It looks quite futuristic, but that is the actual London skyline.

- **UPDATE: The PSF also blogged about the sprint.**

In the next episode I hope to share some progress with the actions to generate AST nodes.

Programming Python Parsing