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)*
```

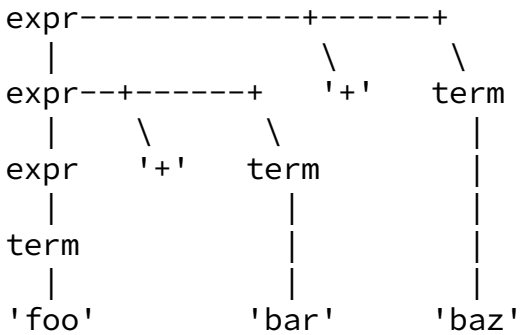
This is in fact exactly what Python's current grammar uses to accommodate the *pgen* parser generator (which has the same issue with left-recursive rules).

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:



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).

So here today's story 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 Python e Parsing an 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.)

Follow

Written by Guido van Rossum

3.3K Followers

Creator of Python

More from Guido van Rossum