Some problems of recursive descent parsers (https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers)

March 14, 2009 at 11:24 **Tags** Articles (https://eli.thegreenplace.net/tag/articles) , Compilation (https://eli.thegreenplace.net/tag/compilation) , Recursive descent parsing (https://eli.thegreenplace.net/tag/recursive-descent-parsing)

Reminder - recursive descent (RD) parsers

Here's an article (https://eli.thegreenplace.net/2008/09/26/recursive-descent-ll-and-predictive-parsers/) I wrote on the subject a few months ago. It provides a good introduction on how RD parsers are constructed and what grammars they can parse.

Here I want to focus on a couple of problems with the RD parser developed in that article, and propose solutions.

Problem #1: operator associativity

If you recall from the previous article (https://eli.thegreenplace.net/2008/09/26/recursive-descent-ll-and-predictive-parsers/), the `expr` rule of the parser looks like this (BNF notation):

```
<expr>    : <term> + <expr>
          | <term> - <expr>
          | <term>
```

It's built this way (`expr` on the right-hand side of the expression, `term` on the left-hand side), to avoid *left-recursion* in the grammar, which can crash a RD parser by sending it wheeling in an infinite loop.

But as I hinted in the footnotes (and some readers caught on in the comments), this injects an associativity problem into the grammar. Let's see why.

Wikipedia is much better than me at explaining what operator associativity (http://en.wikipedia.org/wiki/Operator_associativity) is, so I'll assume you've read and understood it.

In short, however, left associativity of the minus operator means that `5 - 1 - 2 = (5 - 1) - 2` and not `5 - (1 - 2)` (which returns a different result).

But if you run `5 - 1 - 2` in the parser with the above BNF for `expr`, you'll get 6 instead of 2. So what went wrong?

The problem is in the grammar definition (BNF) itself. The way the `expr` rule is defined makes it inherently right-associative instead of left-associative. The hierarchy of the rules implicitly defines their associativity, because it defines what will be grouped together. To understand it better, perhaps the code implementing the `expr` rule will help:

```
def _expr(self):
    lval = self._term()

    if self.cur_token.type == '+':
        self._match('+')
        op = lambda a, b: a + b
    elif self.cur_token.type == '-':
        self._match('-')
        op = lambda a, b: a - b
    else:
        print 'returning lval = %s' % lval
        return lval

    rval = self._expr()
    print 'lval = %s, rval = %s, res = %s' % (
        lval, rval, op(lval, rval))
    return op(lval, rval)
```

Note that the first `term` is parsed, and then the rule recursively calls itself for the next one. So the expression is being built from right to left, and this causes its right-associativity.

As you can see, I've added a couple of printouts to better show what's going on. When run on the expression `5 - 1 - 2`, this prints:

```
returning lval = 2
lval = 1, rval = 2, res = -1
lval = 5, rval = -1, res = 6
```

We clearly see the problem here. The actual returns are done from right to left because of the recursion.

Note that this grammar evaluates addition, multiplication, subtraction and division in a right-associative way. This causes problems for both subtraction and division, but not for addition and multiplication, because these operations compute the same whether right-to-left or left-to-right [1].

A solution for the associativity problem

I suppose the problem can be solved by rewriting the BNF rules in some sophisticated way that makes them both left-associative and not left-recursive [2], but I'll pick another way.

BNF is somewhat limiting, since it doesn't really allow much options when defining rules. All the rules must have a very strict structure, and if you want to customize something you must resort to defining sub-rules and referencing them recursively.

Enter EBNF (http://en.wikipedia.org/wiki/Ebnf). It was developed to fix some of the deficiencies of plain BNF. One of those is the addition of repetition of sub-rules. For instance, we can write the `expr` rule in EBNF as follows:

```
<expr>    : <term> {+ <term>}
          | <term> {- <term>}
```

Note the braces `{ ... }`. In EBNF, these mean "repeated 0 or more times". This is still a `LL(1)` grammar, but now it's expressed a bit more comfortably. Such a representation is very suitable for coding, because the repetition can be expressed naturally with a loop.

Here's a re-implementation of the `expr` rule using this idiom:

```
def _expr(self):
    lval = self._term()

    while ( self.cur_token.type == '+' or
            self.cur_token.type == '-'):
        if self.cur_token.type == '+':
            self._match('+')
            lval += self._term()
        elif self.cur_token.type == '-':
            self._match('-')
            lval -= self._term()

    return lval
```

Note the `while` loop "eating up" all successive terms in the expression and accumulating the result in the expected left-to-right manner. Now the computation `5 - 1 - 2` will correctly produce `2`.

The code

This is a good place to refer to the code. In here (https://github.com/eliben/code-for-blog/tree/master/2009/py_rd_parser_example) you will find the source of both the old (BNF-based) parser and the new (EBNF-based) one, along with the `lexer` module that implements the tokenizer. Each of the parsers is self contained and can be used separately. Note that they were developed and tested with Python 2.5

Right-associative operators

Some operators are inherently right-associative. Exponentiation, for example. `2^3^2 = 2^(3^2) = 512`, and not `(2^3)^2` (which equals 64).

We can leave these operators defined as before, using a recursive rule that naturally results in right-associativity. Here's the code of the `power` rule that was added to the EBNF-based parser to support exponentiation:

```
# <power>   : <factor> ** <power>
#           | <factor>
#
def _power(self):
    lval = self._factor()

    if self.cur_token.type == '**':
        self._match('**')
        lval **= self._power()

    return lval
```

Intermission

We now have a correct recursive descent parser that uses EBNF-based rules to parse expressions with the desired associativity for each operator. This parser can be readily employed to parse simple languages - it is production-use ready. The next "problem" I present only has to do with the parser's efficiency, so it is probably of no concern unless performance is crucial.

Problem #2: efficiency

There's an inherent performance problem with recursive-descent parsers when dealing with expressions. This problem stems from the need to define operator precedence, and in RD parsers the only way to define this precedence is by using recursive sub-rules. For example (from the EBNF-based code):

```
<expr>    : <term> {+ <term>}
          | <term> {- <term>}
<term>    : <power> {* <power>}
          | <power> {/ <power>}
```

The nesting of these rules defines the relative precedence of addition and multiplication. It tells the parser: between plus signs, dive into the expression and collect all sub-terms connected by multiply signs. In other words, it tells it to group the expression: `5 + 2 * 2` as `5 + (2 * 2)` and not as `(5 + 2) * 2`.

To see the problem this nesting causes, I've inserted simple printouts into each of the `expr`, `term`, `power` and `factor` rules to show which functions get called while parsing. Let's see what happens when the trivial expression `42` is

parsed:

```
expr called with NUMBER(42) at 0
term called with NUMBER(42) at 0
power called with NUMBER(42) at 0
factor called with NUMBER(42) at 0
```

Yikes!!! 4 function calls just to parse the single-token input 42! Unfortunately, while this problem may look simple on the surface, it is not. There's simply no other way to express precedence in RD parsers - you have to use nested rules, and this nesting turns out to be inefficient for parsing expressions.

The solution to this problem is to use a hybrid parser instead of a pure RD one. Some algorithms were developed to efficiently parse infix expressions (http://en.wikipedia.org/wiki/Infix_notation). This article (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) provides a good survey. One such algorithm can be combined with RD to provide a general-purpose parser for both expressions and higher programming language constructs.

In a future article I will discuss an implementation of such a parser.

---

[1] To be more precise, addition and multiplication are associative binary operators (http://en.wikipedia.org/wiki/Associativity) in the mathematical sense.

[2] But I'm too lazy to look for such a way at the moment. Let me know if you find it.

For comments, please send me ✉ an email (mailto:eliben@gmail.com).

↑ Back to top