Parsing expressions by precedence climbing (https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing)

August 02, 2012 at 05:48 **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)

I've written previously (https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers/) about the problem recursive descent parsers have with expressions, especially when the language has multiple levels of operator precedence.

There are several ways to attack this problem. The Wikipedia article on operator-precedence parsers (http://en.wikipedia.org/wiki/Operator-precedence_parser) mentions three algorithms: Shunting Yard, top-down operator precedence (TDOP) and precedence climbing. I have already covered Shunting Yard (https://eli.thegreenplace.net/2009/03/20/a-recursive-descent-parser-with-an-infix-expression-evaluator/) and TDOP (https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing/) in this blog. Here I aim to present the third method (and the one that actually ends up being used a lot in practice) - precedence climbing.

Precedence climbing - what it aims to achieve

It's not necessary to be familiar with the other algorithms for expression parsing in order to understand precedence climbing. In fact, I think that precedence climbing is the simplest of them all. To explain it, I want to first present what the algorithm is trying to achieve. After this, I will explain how it does this, and finally will present a fully functional implementation in Python.

So the basic goal of the algorithm is the following: treat an expression as a bunch of nested sub-expressions, where each sub-expression has in common the lowest precedence level of the the operators it contains.

Here's a simple example:

```
2 + 3 * 4 * 5 - 6
```

Assuming that the precedence of `+` (and `-`) is 1 and the precedence of `*` (and `/`) is 2, we have:

```
2 + 3 * 4 * 5 - 6

|---------------|   : prec 1
    |-------|       : prec 2
```

The sub-expression multiplying the three numbers has a minimal precedence of 2. The sub-expression spanning the whole original expression has a minimal precedence of 1.

Here's a more complex example, adding a power operator `^` with precedence 3:

```
2 + 3 ^ 2 * 3 + 4

|---------------|   : prec 1
    |-------|       : prec 2
    |---|           : prec 3
```

Associativity

Binary operators, in addition to precedence, also have the concept of *associativity*. Simply put, *left associative* operators stick to the left stronger than to the right; *right associative* operators vice versa.

Some examples. Since addition is left associative, this:

```
2 + 3 + 4
```

Is equivalent to this:

```
(2 + 3) + 4
```

On the other hand, power (exponentiation) is right associative. This:

```
2 ^ 3 ^ 4
```

Is equivalent to this:

```
2 ^ (3 ^ 4)
```

The precedence climbing algorithm also needs to handle associativity correctly.

Nested parenthesized sub-expressions

Finally, we all know that parentheses can be used to explicitly group sub-expressions, beating operator precedence. So the following expression computes the addition *before* the multiplication:

```
2 * (3 + 5) * 7
```

As we'll see, the algorithm has a special provision to cleverly handle nested sub-expressions.

Precedence climbing - how it actually works

First let's define some terms. *Atoms* are either numbers or parenthesized expressions. *Expressions* consist of atoms connected by binary operators [1]. Note how these two terms are mutually dependent. This is normal in the land of grammars and parsers.

The algorithm is *operator-guided*. Its fundamental step is to consume the next atom and look at the operator following it. If the operator has precedence lower than the lowest acceptable for the current step, the algorithm returns. Otherwise, it calls itself in a loop to handle the sub-expression. In pseudo-code, it looks like this [2]:

```
compute_expr(min_prec):
  result = compute_atom()

  while cur token is a binary operator with precedence >= min_prec:
    prec, assoc = precedence and associativity of current token
    if assoc is left:
      next_min_prec = prec + 1
    else:
      next_min_prec = prec
    rhs = compute_expr(next_min_prec)
    result = compute operator(result, rhs)

  return result
```

Each recursive call here handles a sequence of operator-connected atoms sharing the same minimal precedence.

An example

To get a feel for how the algorithm works, let's start with an example:

```
2 + 3 ^ 2 * 3 + 4
```

It's recommended to follow the execution of the algorithm through this expression with, on paper. The computation is kicked off by calling `compute_expr(1)`, because 1 is the minimal operator precedence among all operators we've defined. Here is the "call tree" the algorithm produces for this expression:

```
* compute_expr(1)               # Initial call on the whole expression
  * compute_atom() --> 2
  * compute_expr(2)             # Loop entered, operator '+'
    * compute_atom() --> 3
    * compute_expr(3)
      * compute_atom() --> 2
      * result --> 2            # Loop not entered for '*' (prec < '^')
    * result = 3 ^ 2 --> 9
    * compute_expr(3)
      * compute_atom() --> 3
      * result --> 3            # Loop not entered for '+' (prec < '*')
    * result = 9 * 3 --> 27
  * result = 2 + 27 --> 29
  * compute_expr(2)             # Loop entered, operator '+'
    * compute_atom() --> 4
    * result --> 4              # Loop not entered - end of expression
  * result = 29 + 4 --> 33
```

Handling precedence

Note that the algorithm makes one recursive call per binary operator. Some of these calls are short lived - they will only consume an atom and return it because the `while` loop is not entered (this happens on the second 2, as well as on the second 3 in the example expression above). Some are longer lived. The initial call to `compute_expr` will compute the whole expression.

The `while` loop is the essential ingredient here. It's the thing that makes sure that the current `compute_expr` call handles all consecutive operators with the given minimal precedence before exiting.

Handling associativity

In my opinion, one of the coolest aspects of this algorithm is the simple and elegant way it handles associativity. It's all in that condition that either sets the minimal precedence for the next call to the current one, or current one

plus one.

Here's how this works. Assume we have this sub-expression somewhere:

```
8 * 9 * 10

  ^
  |
```

The arrow marks where the `compute_expr` call is, having entered the `while` loop. `prec` is 2. Since the associativity of `*` is left, `next_min_prec` is set to 3. The recursive call to `compute_expr(3)`, after consuming an atom, sees the next `*` token:

```
8 * 9 * 10

      ^
      |
```

Since the precedence of `*` is 2, while `min_prec` is 3, the `while` loop never runs and the call returns. So the original `compute_expr` will get to handle the second multiplication, not the internal call. Essentially, this means that the expression is grouped as follows:

```
(8 * 9) * 10
```

Which is exactly what we want from left associativity.

In contrast, for this expression:

```
8 ^ 9 ^ 10
```

The precedence of `^` is 3, and since it's right associative, the `min_prec` for the recursive call stays 3. This will mean that the recursive call *will* consume the next `^` operator before returning to the original `compute_expr`, grouping the expression as follows:

```
8 ^ (9 ^ 10)
```

Handling sub-expressions

The algorithm pseudo-code presented above doesn't explain how parenthesized sub-expressions are handled. Consider this expression:

```
2000 * (4 - 3) / 100
```

It's not clear how the `while` loop can handle this. The answer is `compute_atom`. When it sees a left paren, it knows that a sub-expression will follow, so it calls `compute_expr` on the sub expression (which lasts until the matching right paren), and returns its result as the result of the atom. So `compute_expr` is oblivious to the existence of sub-expressions.

Finally, in order to stay short the pseudo-code leaves some interesting details out. What follows is a full implementation of the algorithm that fills all the gaps.

A Python implementation

Here is a Python implementation of expression parsing by precedence climbing. It's kept short for simplicity, but can be be easily expanded to cover a more real-world language of expressions. The following sections present the code in small chunks. The whole code is available here (https://github.com/eliben/code-for-blog/blob/master/2012/rd_infix_precedence.py).

I'll start with a small tokenizer class that breaks text into tokens and keeps a state. The grammar is very simple: numeric expressions, the basic arithmetic operators `+`, `-`, `*`, `/`, `^` and parens - `(`, `)`.

```
Tok = namedtuple('Tok', 'name value')


class Tokenizer(object):
    """ Simple tokenizer object. The cur_token attribute holds the current
        token (Tok). Call get_next_token() to advance to the
        next token. cur_token is None before the first token is
        taken and after the source ends.
    """
    TOKPATTERN = re.compile("\s*(?:(\d+)|(.))")

    def __init__(self, source):
        self._tokgen = self._gen_tokens(source)
        self.cur_token = None

    def get_next_token(self):
        """ Advance to the next token, and return it.
        """
        try:
            self.cur_token = self._tokgen.next()
        except StopIteration:
            self.cur_token = None
        return self.cur_token

    def _gen_tokens(self, source):
        for number, operator in self.TOKPATTERN.findall(source):
            if number:
                yield Tok('NUMBER', number)
            elif operator == '(':
                yield Tok('LEFTPAREN', '(')
            elif operator == ')':
                yield Tok('RIGHTPAREN', ')')
            else:
                yield Tok('BINOP', operator)
```

Next, `compute_atom`:

```
def compute_atom(tokenizer):
    tok = tokenizer.cur_token
    if tok.name == 'LEFTPAREN':
        tokenizer.get_next_token()
        val = compute_expr(tokenizer, 1)
        if tokenizer.cur_token.name != 'RIGHTPAREN':
            parse_error('unmatched "("')
        tokenizer.get_next_token()
        return val
    elif tok is None:
            parse_error('source ended unexpectedly')
    elif tok.name == 'BINOP':
        parse_error('expected an atom, not an operator "%s"' % tok.value)
    else:
        assert tok.name == 'NUMBER'
        tokenizer.get_next_token()
        return int(tok.value)
```

It handles true atoms (numbers in our case), as well as parenthesized sub-expressions.

Here is `compute_expr` itself, which is very close to the pseudo-code shown above:

```
# For each operator, a (precedence, associativity) pair.
OpInfo = namedtuple('OpInfo', 'prec assoc')

OPINFO_MAP = {
    '+':    OpInfo(1, 'LEFT'),
    '-':    OpInfo(1, 'LEFT'),
    '*':    OpInfo(2, 'LEFT'),
    '/':    OpInfo(2, 'LEFT'),
    '^':    OpInfo(3, 'RIGHT'),
}

def compute_expr(tokenizer, min_prec):
    atom_lhs = compute_atom(tokenizer)

    while True:
        cur = tokenizer.cur_token
        if (cur is None or cur.name != 'BINOP'
                        or OPINFO_MAP[cur.value].prec < min_prec):
            break

        # Inside this loop the current token is a binary operator
        assert cur.name == 'BINOP'

        # Get the operator's precedence and associativity, and compute a
        # minimal precedence for the recursive call
        op = cur.value
        prec, assoc = OPINFO_MAP[op]
        next_min_prec = prec + 1 if assoc == 'LEFT' else prec

        # Consume the current token and prepare the next one for the
        # recursive call
        tokenizer.get_next_token()
        atom_rhs = compute_expr(tokenizer, next_min_prec)

        # Update lhs with the new value
        atom_lhs = compute_op(op, atom_lhs, atom_rhs)

    return atom_lhs
```

The only difference is that this code makes token handling more explicit. It basically follows the usual "recursive-descent protocol". Each recursive call has the current token available in `tokenizer.cur_tok`, and makes sure to consume all the tokens it has handled (by calling `tokenizer.get_next_token()`).

One additional small piece is missing. `compute_op` simply performs the arithmetic computation for the supported binary operators:

```
def compute_op(op, lhs, rhs):
    lhs = int(lhs); rhs = int(rhs)
    if op == '+':   return lhs + rhs
    elif op == '-': return lhs - rhs
    elif op == '*': return lhs * rhs
    elif op == '/': return lhs / rhs
    elif op == '^': return lhs ** rhs
    else:
        parse_error('unknown operator "%s"' % op)
```

In the real world - Clang

Precedence climbing is being used in real world tools. One example is Clang (http://clang.llvm.org/), the C/C++/ObjC front-end. Clang's parser is hand-written recursive descent, and it uses precedence climbing for efficient parsing of expressions. If you're interested to see the code, it's `Parser::ParseExpression` in `lib/Parse/ParseExpr.cpp` [3]. This method plays the role of `compute_expr`. The role of `compute_atom` is played by `Parser::ParseCastExpression`.

Other resources

Here are some resources I found useful while writing this article:

- The Wikipedia page for Operator-precedence parsing (http://en.wikipedia.org/wiki/Operator-precedence_parser).
- The article by Keith Clarke (http://antlr.org/papers/Clarke-expr-parsing-1986.pdf) (PDF), one of the early inventors of the technique.
- This page (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) by Theodore Norvell, about parsing expressions by recursive descent.
- The Clang source code (exact locations given in the previous section).

Update (2016-11-02): *Andy Chu notes (http://www.oilshell.org/blog/2016/11/01.html) that precedence climbing and TDOP (https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing) are pretty much the same algorithm, formulated a bit differently. I tend to agree, and also note that Shunting Yard (https://eli.thegreenplace.net/2009/03/20/a-recursive-descent-parser-with-an-infix-expression-evaluator) is again the same algorithm, except that the explicit recursion is replaced by a stack.*

[1] There are a couple of simplifications made here on purpose. First, I assume only numeric expressions. Identifiers that represent variables can also be viewed as atoms. Second, I ignore unary operators. These are quite easy to incorporate into the algorithm by also treating them as atoms. I leave them out for succinctness.

[2] In this article I present a parser that computes the result of a numeric expression on-the-fly. Modifying it for accumulating the result into some kind of a parse tree is trivial.

[3] Clang's source code is constantly in flow. This information is correct at least for the date the article was written.

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

↑ Back to top