A recursive descent parser with an infix expression evaluator (https://eli.thegreenplace.net/2009/03/20/a-recursive-descent-parser-with-an-infix-expression-evaluator)

March 20, 2009 at 18:01 **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)

Last week (https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers/) I wrote about some of the inherent problems of recursive-descent parsers. An elegant solution to the operator associativity problem was shown, but another problem remained - and that is of the unwieldy handling of expressions, mainly performance-wise.

Here I want to present one alternative to the pure-RD approach, and that is intermixing RD with another parsing method.

The code

I'll begin by pointing to the code for this article (https://github.com/eliben/code-for-blog/tree/master/2009/py_rd_parser_example). It contains several Python files and a `readme.txt` explaining what is what. Throughout the article I'll present short snippets from the code, but it's encouraged to run it on your own. The code is self-contained and only requires Python (version 2.5) to run.

Extending the grammar

To illuminate some of the points I'm presenting better, I've greatly extended the EBNF grammar we'll be parsing. Here's the new grammar (taken from the top of the `rd_parser_ebnf.py` in the code .zip):

```
# EBNF:
#
# <stmt>        : <assign_stmt>
#               | <if_stmt>
#               | <cmp_expr>
#
# <assign_stmt> : set <id> = <cmp_expr>
#
## Note 'else' binds to the innermost 'if', like in C
#
# <if_stmt>     : if <cmp_expr> then <stmt> [else <stmt>]
#
# <cmp_expr>    : <bitor_expr> [== <bitor_expr>]
#               | <bitor_expr> [!= <bitor_expr>]
#               | <bitor_expr> [> <bitor_expr>]
#               | <bitor_expr> [< <bitor_expr>]
#               | <bitor_expr> [>= <bitor_expr>]
#               | <bitor_expr> [<= <bitor_expr>]
#
# <bitor_expr>  | <bitxor_expr> {| <bitxor_expr>}
#
# <bitxor_expr> | <bitand_expr> {^ <bitand_expr>}
#
# <bitand_expr> | <shift_expr> {& <shift_expr>}
#
# <shift_expr>  | <arith_expr> {<< <arith_expr>}
#               : <arith_expr> {>> <arith_expr>}
#
# <arith_expr>  : <term> {+ <term>}
#               | <term> {- <term>}
#
# <term>        : <power> {* <power>}
#               | <power> {/ <power>}
#
# <power>       : <power> ** <factor>
#               | <factor>
#
# <factor>      : <id>
#               | <number>
#               | - <factor>
#               | ( <cmp_expr> )
```

```
#
# <id>          : [a-zA-Z_]\w+
# <number>      : \d+
```

As you can see, this simple calculator is starting to approach a real programming language, as it supports a plethora of mathematical and logical expressions, as well as conditional statements (`if ... then ... else`) and assignments. I've added a simplistic "prompt" so you can experiment with the calculator from the command line:

```
D:\zzz\rd_parser_calc>rd_parser_ebnf.py -p
Welcome to the calculator. Press Ctrl+C to exit.
--> set x = 2 + 2 * 3
8
--> set y = (x - 1) * (x - 2)
42
--> if y > x then set y = x else set x = y
8
--> x
8
--> y
8
--> x ** ((y - 10) * -3)
262144
--> ... Thanks for using the calculator.
```

Note that since a separate expression "level" is required for each precedence, the resulting code is somewhat repetitive. I'll get back to this point later on.

Evaluating infix expressions

An alternative method of evaluating expressions is required, then. Luckily, such a need arose early enough (in the 1950s and 60s, when first compilers and interpreters were constructed) and some luminaries examined this problem in detail. In particular, Edsger W. Dijkstra (http://en.wikipedia.org/wiki/Edsger_Dijkstra) proposed an efficient and intuitive algorithm for converting from infix notation (http://en.wikipedia.org/wiki/Infix_notation) to RPN (http://en.wikipedia.org/wiki/Reverse_Polish_notation), called the Shunting Yard algorithm (http://en.wikipedia.org/wiki/Shunting_yard_algorithm).

I will not describe the algorithm here, as it's been done several times already. If the Wikipedia article is not enough, here's another good source (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) (which I've actually used as the basis for my implementation).

The algorithm employs two stacks to resolve the precedence dilemmas of infix notation. One stack is for storing operators of relatively low precedence that await results from computations with higher precedence. The other stack keeps the result accumulated so far. The result can either be a RPN expression, an AST or just the computed result (a number) of the computation.

In my code, the file `rd_parser_infix_exper.py` implements a hybrid parser, using Shunting Yard to evaluate expressions and a top-level RD parser for statements and combining everything together. It's instructive to examine the implementation and see how things fit together.

The grammar this parser accepts is exactly the same as the pure RD EBNF parser presented eariler. The statements (`assign_stmt`, `if_stmt`, and `stmt`) are evaluated by traditional RD, but getting deeper into expressions is done with an "infix evaluator", the gateway to which is the `_infix_eval` method [1]:

```python
def _infix_eval(self):
    """ Run the infix evaluator and return the result.
    """
    self.op_stack = []
    self.res_stack = []

    self.op_stack.append(self._sentinel)
    self._infix_eval_expr()
    return self.res_stack[-1]
```

This method prepares the Shunting Yard stacks and begins evaluating the expression, terminating with returning its results.

Note that the connection to the RD parser is seamless. When `_infix_eval` is called, it assumes that the current token is the beginning of an expression (just like any RD rule), and consumes as much tokens as required to parse the full expression before returning the result.

The rest of the implementation (the `_infix_eval_expr`, `_infix_eval_atom`, `_push_op` and `_pop_op` methods) is pretty much a word by word translation of the algorithm described in this article (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) into Python.

Adding expressions

Here's a big advantage of this hybrid parser: adding new expressions and/or changing precedence levels is much simpler and requires far less code. In the pure RD parser, the operators and their precedences are determined by the structure of recursive calls between methods. Adding a new operator requires a new method, as well as modifying some of the other methods [2]. Changing the precedence of some operator is also troublesome and requires moving around lots of code.

Not so in the infix expression parser. Once the Shunting Yard machinery is in place, all we have to do to add new operators or modify existing ones is update the `_ops` table:

```
_ops = {
    'u-':   Op('unary -', operator.neg, 90, unary=True),
    '**':   Op('**', operator.pow, 70, right_assoc=True),
    '*':    Op('*', operator.mul, 50),
    '/':    Op('/', operator.div, 50),
    '+':    Op('+', operator.add, 40),
    '-':    Op('-', operator.sub, 40),
    '<<':   Op('<<', operator.lshift, 35),
    '>>':   Op('>>', operator.rshift, 35),
    '&':    Op('&', operator.and_, 30),
    '^':    Op('^', operator.xor, 29),
    '|':    Op('|', operator.or_, 28),
    '>':    Op('>', operator.gt, 20),
    '>=':   Op('>=', operator.ge, 20),
    '<':    Op('<', operator.lt, 20),
    '<=':   Op('<=', operator.le, 20),
    '==':   Op('==', operator.eq, 15),
    '!=':   Op('!=', operator.ne, 15),
}
```

I also find this table much more descriptive in the sense of understanding how the operators relate to one another than the parallel 9 methods required to implement them in the pure RD version (`rd_parser_ebnf.py`).

Performance

Now here is the funny thing. My initial motivation for examining the infix expression hybrid was the allegedly poor performance of the RD parser for parsing expressions (as described in the previous article (https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers/)). But the performance hasn't improved! In fact, the new hybrid parser is a bit slower than the pure RD parser!

And the annoying thing is that it's entirely unclear to me how to optimize it, since profiling shows that the runtime divides rather evenly between the various methods of the algorithm. Yes, the pure RD parser requires the full precedence-chain of methods called for each single terminal, but the infix version has more method calls in total.

If anything, this has been a lesson in optimization, as profiling initially showed that the vast majority of the time is spent in the lexer [3]. So I've managed to optimize my lexer (by precompiling all its regexes into a single large one using alternation), which greatly reduced the runtime.

Conclusion

This article has presented an alternative to the pure recursive-descent parser. The hybrid parser developed here combines RD with infix expression evaluation using the Shunting Yard algorithm.

We've seen that the new code is more manageable for operator-rich grammars. If even more operators are to be added to the parser (such as the full set of operators supported by C), it's much simpler to implement into the parser, and the operator table is a single place summarizing the operators, their associativities and precedences, making the parser more readable.

However, this has not made the parser any faster. The pure-RD implementation is lean enough to be efficient even when the grammar consists of many precedence levels. This is an important lesson in optimization - it's difficult to assess the relative runtimes of complex chunks of code in advance, without actually trying them out and profiling them.

[1] It would be a swell idea to read the description of the algorithm and have an intuitive understanding of it from this point and on in the article.

[2] Suppose we had no multiplication and division and had to add the `term` rule. In addition to writing the code for the new rule, we must modify the `arith_expr` rule to now call `term` instead of `power`.

[3] Which makes lots of sense, as it's well known that lexing/tokenization is usually the most time consuming stage of parsing. This is because the lexer has to examine every single character of the input, while the parser above it works on the level of whole tokens.

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

Back to top