References and Remarks

joekreu

September 2021

1. Introduction

This document contains some references to the parsers in this repository and some further remarks on related parsing algorithms.

Note that the terminology for the parsing algorithms is not always consistent.

2. Parsing expressions with precedence levels: Classic, Shunting-yard, Precedence climbing, Pratt

The 'classic' way for parsing expressions with different precedence levels of operators is modelled after a *context free grammar* (*CFG*) that uses different non-terminals for different precedence levels and separate procedures for the non-terminal. This leads to a large number of procedure definitions and calls. In

- *Compact recursive-descent Parsing of Expressions* by *David R. Hanson* (1985)

the author showed how the number of procedure *definitions* can be reduced.

The classic algorithm is not considered further here. CFGs will not play a major role. Instead, the algorithms in this repository are based on *binding powers* of the operators. Binding powers are related to *precedence*.

The iterative code (the *shunting-yard* algorithm by *Dijkstra*) and the iterative-recursive code for precedence parsing are described in several publications. *Pratt parsing* may also be considered in this context. See, for example,

- *Parsing Expressions by Recursive Descent* by *Theodore S. Norvell* (1999), https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
- *From Precedence Climbing to Pratt Parsing* by *Theodore S. Norvell* (2016), https://www.engr.mun.ca/~theo/Misc/pratt_parsing.htm

Two recent articles on this topic by *Aleksey Kladov* (*matklad*) (2020):

- *Simple but Powerful Pratt Parsing* (2020), https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html
- *From Pratt to Dijkstra* (2020), https://matklad.github.io/2020/04/15/from-pratt-to-dijkstra.html

I do not use the term *Pratt parser* for the parsers in the repository - I think *precedence climbing* is an appropriate generic name.

The parsers are classified as *iterative*, *iterative-recursive* or *recursive*, with corresponding names of the form `pcp_it...`, `pcp_ir...`, `pcp_rec...`, resp.

There is strong evidence for the following claim: *All implementations in this repository accept the same parsing rule definitions and the same kinds of expressions. Acceptance of an expression and its parse result may depend on the specific parsing rule definitions but not on the individual parser within this repo. All parsers produce the same result on the same input; subexpression are created in the same order.*

The iterative parser in this repo implement essentially the same as what is usually called the *shunting yard* algorithm. Note, however, that many implementations of the shunting-yard algorithm can directly (without recursive calls) parse parenthesized subexpressions.

The *iterative-recursive* code is often simply called *precedence climbing*.

The parsers `pcp_rec_...` show that equivalent *recursive*, even *purely functional*, parsers can be implemented in Python.

The original description of the shunting-yard algorithm is in

- *Algol 60 translation* (1961) by *E. W. Dijkstra*; see https://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF

An early reference to the *precedence climbing* algorithm (iterative-recursive code), with pseudo-code very similar to the Python code in `pcp_ir_0`, is

- *The top-down parsing of expressions* by *Keith Clarke* (1986), https://www.antlr.org/papers/Clarke-expr-parsing-1986.pdf.

A more recent article on precedence climbing is

- *Parsing expressions by precedence climbing* by *Eli Bendersky* (2012), https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing (with Python code).

The differences between Pratt parsing and precedence climbing are not great. See, e.g., the following blog post:

- *Pratt Parsing and Precedence Climbing Are the Same Algorithm* by *Andy Chu*, https://www.oilshell.org/blog/2016/11/01.html

Andy Chu's blog (mainly about a command shell called "Oil") contains a number of hints and references to expression parsing (precedence climbing, Pratt parsing, shunting-yard algorithm):

- http://www.oilshell.org/blog/tags.html?tag=parsing#parsing

Pratt parsers (also called *Top down operator precedence parsers*) typically define a *class* for every operator, with two essential methods often called *led* (*left denotation*) and *nud* (*null denotation*). This approach is more general than *precedence climbing*. It seems more difficult to me to make this method *dynamic*, in the sense that the parsing rules can be read from a table or a dictionary.

Pratt parsing was first described in

- *Top down operator precedence* by *Vaughan Pratt* (1973), https://dl.acm.org/doi/10.1145/512927.512931.

A more recent article on Pratt parsing is

- *Top-Down operator precedence parsing* by *Eli Bendersky* (2010), https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing.

The GitHub repository

- *Operator precedence parsers* by *Jean-Marc Bourguet*, https://github.com/bourguet/operator_precedence_parsing

contains Python implementations of Pratt, shunting-yard, precedence climbing and related parsers.

3. Precedence and associativity. Binding powers

Both *precedence climbing* and *Pratt parsing* can be based on *precedence* and *associativity*, or alternatively on two *binding powers*, for each operator.

In most cases the approach *precedence* and *associativity* can be easily emulated by binding powers: for precedence `n` and associativity `left` set `lbp = n`, `rbp = n`; for precedence `n` and associativity `right` set `lbp = n+1`, `rbp = n`. To make this work, a crucial comparison operator in the parser's source code (`<` or `<=`) must be set such that `lbp = rbp` means *left associative*. Difficulties may arise if two different operators have the same precedence but different associativity.

The approach based on arbitrary *lbp* and *rbp* is more general. See, e.g., posts by *Dmitry A. Kazakov* and *James Harris* and others in

- http://compgroups.net/comp.compilers/compiling-expressions/2145696

from 3 Jan 2013 12:01:33.

In the *Maxima* computer algebra system, operators have *lbp* and *rbp*, and for assignment operators the *lbp* is 180 and the *rbp* is 20. See the *Maxima* manual, for example at

- http://maxima.sourceforge.net/docs/manual/maxima_35.html

Maxima also supports *user defined operators*, for these operators *lbp* and *rbp* can be specified. See

- http://maxima.sourceforge.net/docs/manual/maxima_41.html

The *Maxima* expression parser is a Pratt parser, implemented in Lisp:

- https://github.com/andrejv/maxima/blob/master/src/nparse.lisp

Another, now historical, computer algebra system that uses Pratt parsing and binding powers is *muMATH*, based on *muSIMP*. Search for `Pratt` in

- https://archive.org/stream/MuMath_and_MuSimp_1980_Soft_Warehouse/MuMath_and_MuSimp_1980_Soft_Warehouse_djvu.txt

In *muMATH*, the assignment operator also has an *lbp* of 180 and an *rbp* of 20.

In this repository, the approach based on binding powers is also used to parse unary operators (prefix and postfix) as infix operators, after inserting fake operand tokens and assigning additional fake binding powers to the unary operators. See README.

4. Further references

In

- *Making a Pratt Parser Generator*, by *Robert Jacobson* (2020), https://www.robertjacobson.dev/designing-a-pratt-parser-generator

the author supports the definition of parse properties of operators in a table that should be used in a kind of Pratt parsing.

Additional references for *Pratt parsing*:

- *Top Down Operator Precedence* by *Douglas Crockford* (2007), https://www.crockford.com/javascript/tdop/tdop.html. Code in *JavaScript*.
- *Simple Top-Down Parsing in Python* by *Fredrik Lundh* (*effbot*) (2008). Lundh's site https://effbot.org/ is currently (September 2021) not accessible.
- *Pratt Parsers: Expression Parsing Made Easy* by *Bob Nystrom* (2011), https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/. Code in *Java*.

In the two articles (the second one is more detailed)

- *Precedences in specifications and implementations of programming languages*, by *Annika Aasa* (1995), https://www.sciencedirect.com/science/article/pii/030439759590680J
- *User defined syntax*, by *Annika Aasa* (1992), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3542

among other questions, precedence in connection with prefix, infix and postfix operators is investigated; the operators can have any combination of precedence, and (in case of infix operators) associativity.

Aasa gives a definition of *precedence correct parse tree* in terms of precedence and associativity and shows that every *valid* expression (i.e., an expression generated by a *precedence grammar*) has exactly one precedence correct parse tree.

She investigates relations to parsers and CFGs. Parsing with an *Operator precedence parser*, as described by *Aho*, *Sethi*, *Ullman* in Chapter 4.6. of *Compilers: Principles, Techniques, and Tools* (1986) creates precedence correct parse trees. Further, she shows how a precedence grammar can be transformed to an equivalent unambiguous CFG.

The module `helpers.py` in this repo contains code to check the parse results for *precedence correctness*. The definition of correctness used here is a variant of Aasa's definition. It has been adapted to the use of *arbitrary left and right binding powers* instead of *precedence* and *associativity*. Furthermore, *Aasa* uses precedence values in reverse sense (smaller numbers mean tighter binding).

5. Complexity

The parser implementations in the repository are of complexity `O(n)`, i.e., *linear*, where `n` is the number of input tokens: A call of the `parse_expr` function in the recursive and iterative-recursive variants typically consumes two tokens; in the iterative parsers two tokens are consumed in every iteration, or two subexpressions from the stack are combined to one subexpression.

Increasing the number of precedence or binding power levels will not increase this complexity.

The statements about `O(n)` complexity do not automatically apply to the *correctness checks* of the parse results mentioned in Section 4.

For recursive and iterative-recursive parsers, there is no limit to recursion depth in terms of the number of binding powers levels. Accordingly, for the iterative parsers, there is no limit to the stack usage (number of items on the explicit stack) in terms of the number of binding power levels.

If `^` is a right associative infix operator, the expression

```
a ^ b ^ c ^ d ^ e
```

will be parsed to `(^ a (^ b (^ c (^ d e))))`. The recursion depth is three, and this number can obviously be increased by appending more of `^ f ^ g ...` to the expression.

A possibly more interesting example is the expression

```
& a + & b + & c + & d + & e + & f
```

where `+` is an infix operator and `&` is a prefix operator with lower binding power than `+`; i.e., $rbp(\&) < lbp(+)$ and $rbp(\&) < rbp(+)$.

The expression will be parsed to

```
(& (+ a (& (+ b (& (+ c (& (+ d (& (+ e (& f))))))))))))
```

If parsed with `pcp_ir_0` or `pcp_rec_0...`, the recursion depth will be 10. Obviously, the scheme `& a + & b + & c + & d ...` can be extended to any length. The recursion depth will be twice the number of `+` operators.