Precedence Climbing Parsing based on Binding Powers and Token Insertion

joekreu

June 2022

The repository contains several demo implementations of iterative, recursive and mixed (iterative and recursive) expression parsers based on *binding powers*, *precedence climbing* and *insertion of fake operands.* Very few lines of Python code are enough for the core of a parser that creates a parse tree from operands and operators (prefix, infix, postfix) with virtually arbitrary binding powers.

The term *precedence* is used here in a generic sense: *finding a particular parse tree for otherwise ambiguous expressions, based on some kind of specification of the binding strengths of the operators.* See the note at the end of section 3.1.

Python 3.8 or higher is required.

1. Introduction

The expressions to be parsed can consist of atomic operands, binary infix operators, unary prefix and unary postfix operators. In the following example, `&` is a prefix operator, `!` is a postfix operator, and `>`, `*`, `+` are infix operators:

```
(1)    & a > 7 * b ! + 2
```

Inserting 'fake operands' allows parsing unary operators as infix operators.

> *Note:* The parsing algorithms presented here are much more powerful. Using straightforward extensions, parenthesized subexpression, function invocations and *mixfix* operators, such as `if ... then ... else ...`, can be parsed. Instead of simply fetching the next atomic operand from the token sequence, a whole *primary expression* can be parsed recursively.
>
> *Fake operators* can be inserted to support parsing, in addition to fake operands.
>
> On the other hand, restrictions for valid expressions could be implemented, for example, by disallowing some combinations of adjacent operators.

> This should be investigated separately.
>
> *Note: Token insertion* is also used in another sense in connection with parsing, namely for *error recovery.* This is not considered here.

Generally, precedence climbing parsing of expressions can be controlled in one of the following two ways (other ways might exist):

1. An operator has a *precedence* (in the specific sense, i.e., a number) and an *associativity* (one of the two values: *left* or *right*). In some settings, *none* can be a third possible value of *associativity* (associative use of operator is not allowed).
2. *Binding powers*: An infix operator has a *left* and a *right binding power*, denoted by *lbp* and *rbp*. Initially, prefix operators have only an *rbp* and postfix operator have only an *lbp*. Binding powers are numbers, typically integers. Binding powers indicate the strength of binding in the corresponding direction.

In simple situations, greater binding powers mean the same as higher precedence. Parsing based on binding powers can be more powerful, though. Precedence and associativity can be expressed by equivalent definitions of binding powers, but not always vice versa.

In this repository, parsing of expressions with infix, prefix, and postfix operators is reduced to the following simple scheme with `n` operators and `n + 1` operands:

```
(**)          A0 Op1 A1 Op2 A2 ... Opn An
```

Here, `A0`, `A1`, … are *atomic operands* and `Op1`, `Op2`, … are *infix operators* with *lbp* and *rbp*. Under these conditions, exactly one parse result is found. The case `n = 0` (one atomic operand, no operator) is included.

The *parsing rules* consist of the set of valid operators and their binding powers. The rules can be dynamically loaded, for example, from a `csv` or a JSON file.

Atomic operands (e.g., numbers and identifiers) consist of one *token* only.

The parser's job is to transform the sequence `(**)` into a *parse tree*, taking into account the parsing rules.

For example, usually the operator `*` has higher precedence (or greater binding powers) than the operator `+`, therefore the expression `a + b * c` should be parsed as `a + (b * c)`, not as `(a + b) * c`.

> *Note:* The parentheses are used here only to indicate the precedence. The parsers in the repository can't process parenthesized subexpression (although they can easily be extended to do so).

An infix operator is *left associative* if consecutive occurrences of this operator are parsed left to right. The expression `a + b + c` is parsed as `(a + b) + c`, because `+` is (usually) left associative. The exponentiation operator `^` is usually right associative, therefore `a ^ b ^ c` is parsed as `a ^ (b ^ c)`.

An operator will be right associative if its *rbp* is less than its *lbp*, otherwise it will be left associative.

Unary operators do not fit directly into the scheme `(**)`. They get adjusted to the basic situation by inserting 'fake' operands and 'fake' binding powers. The left operand `$PRE` is inserted before a prefix operator, and the right operand `$POST` is inserted after a postfix operator. Furthermore, prefix operators are assigned a fake left binding power of `100`, and postfix operators are assigned a fake right binding power of `100`. This procedure virtually converts the unary operators to infix operators, with typically very different *lbp* and *rbp*. 'Normal' (user defined) binding powers are required to be less than 100.

By inserting fake operands, the expression `(1)` becomes

```
(2)   $PRE & a > 7 * b ! $POST + 2
```

The procedure of fake operand insertion also works with two or more consecutive prefix operators, and with two or more consecutive postfix operators. Multiple unary operators of the same kind (prefix or postfix) are always processed from "the inside to the outside", independent of the binding powers of the operators.

For example, if `&` and `%` are prefix operators and `A` is an operand, then `& % A` will be parsed as `(& $PRE (% $PRE A))`; after omitting the dummy operands this will become `(& (% A))`. This is independent of the `rbp` of `&` and `%`.

However, things get more complicated when there are prefix *and* postfix operators for the same operand. If, in addition to the previous example, `!` is a postfix operator, then the parse result of `& A !` will depend on the *rbp* of `&` and the *lbp* of `!`. Now consider

```
& % A ! ~
```

where `~` is another postfix operator. Depending on the binding powers, the parse result can be one of the following:

```
(~ (! (& (% A))))
(& (% (~ (! A))))
(& (~ (! (% A))))
(~ (& (% (! A))))
(~ (& (! (% A))))
(& (~ (% (! A))))
```

The fake tokens do not really need to be inserted. It is enough that the parser pretends that they are inserted. In fact, one of the ten parsers in this repository (`pcp_ir_0_no_ins`; see section 3.1) works like this. However, a real insertion, done by the tokenizer, greatly simplifies the parser's precedence climbing code, because it thereby only has to process infix expressions.

Binding powers smaller than 6 are also considered 'reserved'. For example, a negative *lbp* is assigned to the artificial `$END` token (see section 2). The benefits of other small 'internal' binding powers become visible in more elaborate parsers.

E.g., the *comma* can be parsed under certain conditions as a *left associative infix operator* with small binding powers (e.g., with $lbp = rbp = 5$).

In summary, user defined binding powers should be integers in range `6 to 99`. This does not seem to be a serious restriction. If required, the range could easily be extended.

The *lbp* and *rbp* values of a specific operator can be equal or differ by any number, as long as they are in this range. Binding powers of unary operators do not have to be greater than the binding powers of infix operators in the same expression.

The parsers return nested lists that represent parse trees. These lists can be formatted as Lisp-like *S-expressions*. For example, parsing `5 + 3 ! * 4` will create the list `[+, 5, [*, [!, 3, $POST], 4]]`; formatted as an S-expression this is `(+ 5 (* (! 3 $POST) 4))`. Fake operands can easily be removed from the parse tree, so finally we get `(+ 5 (* (! 3) 4))`.

1.1 Goals

The main goals of this repository are

1. Find and compare demo implementations of precedence climbing algorithms based on binding powers. Encourage experimentation.
2. Use insertion of fake operands to facilitate parsing of unary operators.
3. Separate definition of parsing rules from the implementation of the parsers.
4. Better understand the meaning of *precedence correct* parsing.

1.2 Limitations

Exploring the full potential of precedence climbing parsing based on binding powers and token insertion is not a goal of this repository.

Context-free grammars, Backus normal form and the like are not considered.

The related *Pratt parsing* is not explicitly considered. See, however, section 5 (Acknowledgements and References).

The software does not contain *evaluators* of the parsed expressions.

2. Tokenization. Lexical syntax

In a first step, a *tokenizer* (*lexical scanner*) creates a sequence of *tokens* from the input. A token may consist of one or more characters.

In this repo, tokens must be separated by whitespace, or by transition from an alphanumeric to a special character or vice versa. A minus sign immediately followed by a digit is considered alphanumeric. Also, the four characters `_`, `(`, `)`, `;` (underscore, left and right parenthesis, semicolon) are considered alphanumeric. This is because operators of type `(23;12)` or `(10;_)` will be generated if a parser is run with option `-r` or `-d` (see subsections 3.2.1, 3.2.2).

Examples: `3*4+5` is tokenized the same as `3 * 4 + 5` (5 tokens). The input `5!*7` is tokenized as `5 !* 7` (3 tokens), while `5! *7` is tokenized as `5 ! * 7` (4 tokens); `4*-2` is tokenized as `4 * -2` (3 tokens).

Operands should consist of alphanumeric characters, though this is not checked.

The tokenizers are also responsible for inserting the fake operands `$PRE` and `$POST` (except for `tokenizer_b`, see below). In addition, a special `$BEGIN` token is placed at the beginning, and an `$END` token is placed at the end of the token sequence. `$BEGIN` and `$END` can act as a kind of *operators* in the process of parsing. In this context, a negative *rbp* is assigned to `$BEGIN` and a negative *lbp* to `$END`.

The complete token sequence generated by the tokenizer for the example (1) is

```
$BEGIN $PRE & a > 7 * b ! $POST + 2 $END
```

> *Note 1:* With a 'real' tokenizer (usually based on *regular expressions*) the rules for separation of tokens (by whitespace or transition to another kind of characters) can be improved.
> *Note 2:* Only the iterative parsers (see 3.1) explicitly reference the `$BEGIN` token.

There are five tokenizers in this repository:
`tokenizer_a`, `tokenizer_b`, `tokenizer_c`, `tokenizer_d`, `tokenizer_e`. The standard is `tokenizer_a`, the others are included mainly because of special requirements of some parsers. The tokenizers provide interfaces for the actual parsing.

`tokenizer_a` returns a *function*. If `code` is a valid code string, and `toks = tokenizer_a(code)`, then `toks(0)` (or simply `toks()`) will return the current token, `toks(1)` will advance by one token and return the new current token. `tokenizer_b` does not insert the fake tokens `$PRE` and `$POST`, otherwise it works like `tokenizer_a`.

`tokenizer_e` is a *generator* (in the sense of Python) that returns an *iterator* on the tokens. A tokenizer implemented as generator can easily be used in *iterative* (*shunting yard*) parsers.

`tokenizer_c` and `tokenizer_d` return a singly linked list of tokens.

For `tokenizer_d`, a *token* is a named tuple that contains the binding powers of operator tokens. This allows the implementation of a more *functional* parser because, in this way, access to the global data `LBP` and `RBP` can be avoided.

For the other tokenizers, a token is simply the string that represents the token. In this case, a token is recognized as an *operator token* if it is a key in the `LBP`, `RBP` dictionaries.

3. Overview on the parsers. Notes on use

3.1 The individual parsers

There are ten parsers, in separate modules. Nine of them, which shall be called *basic parsers* here, share the same high-level interface:

1. `pcp_ir_0` is based on iteration (loops) *and* recursion. The actual parser is contained in the function `parse_expr` in five lines of Python code. This might be one of the simplest implementations of precedence climbing parsing for infix expressions where the operators can have independent left and right binding powers.

2. `pcp_ir_0_no_ins` is also based on iteration and recursion. Contrary to `pcp_ir_0`, and contrary to the general setting, it is not based on token insertion. Instead, special code takes care of prefix and postfix operators. In a way, the implementation pretends that the extra tokens are present.

3. `pcp_it_0_1w` implements an iterative (kind of *shunting yard*) algorithm with one explicit stack for operands and operators, and one `while` loop.

4. `pcp_it_0_1wg` uses a tokenizer that is implemented as a *generator* in the sense of Python programming. A generator uses the `yield` statement instead of `return`. Otherwise, `pcp_it_0_1wg` is similar to `pcp_it_0_1w`.

5. `pcp_it_0_2w` implements an iterative algorithm with two explicit stacks, one for operands and one for operators, and two nested `while` loops. After minor adjustments a *generator* as tokenizer could also be used here.

6. `pcp_rec_0_0` is recursive (without loops); otherwise, similar to `pcp_ir_0`.

7. `pcp_rec_0_1` is a recursive and more functional parser (in the sense of *functional programming*). It uses a Lisp-like *singly linked list* of tokens.

8. `pcp_rec_0_2` is a recursive and purely functional parser. The tokenizer for this parser and for `pcp_rec_03` uses a singly linked list of tokens, where tokens are implemented as triples (tuples of length 3); operator tokens contain the binding powers as second and third component.

9. `pcp_rec_03` is recursive and purely functional. Its parsing algorithm slightly differs from that of `pcp_re_0_2`.

All these parsers accept the same operator definitions. They use functions from the module `helpers.py`, and they are meant to be run by the same test driver.

There is probably no point in *explaining* the algorithms. The Python code itself seems to be the best explanation.

Analysis of the code and test results support the following claim (but do not provide a formal proof):

> *All basic parsers accept the same set of expressions and create identical results with identical input, provided they use identical operator*

> *and binding power definitions. In the parse process, they create subexpressions in the same order.*

This should also justify the use of the generic term *precedence climbing*.

> *Note:* The term *precedence* is used in both a generic sense and a specific sense.
> In the generic sense, it is about making a precedence decision between operators, in otherwise ambiguous situations.
> In the specific sense, *precedence* is a number assigned to an operator.
> The parsers in this repository are *precedence parsers* in the generic sense, but they are not based on *precedence* in the specific sense.

The remaining parser, `direct_pcp_ir_0`, uses the algorithm of `pcp_ir_0` to parse some 'hard coded' examples. It is 'self-contained' (without dependencies).

3.2 Usage of the parsers

Python 3.8 or later is required because the 'walrus'-operator `:=` is used. Furthermore, the `nonlocal` keyword is used.

Place all the necessary files (see section 4) in the same directory.

The parser modules are not meant to be imported by other Python code. The code is not optimized for speed. There is only minimal error handling.

Run the parsers on the command line. For `direct_pcp_ir_0.py` this is simply

```
python3 direct_pcp_ir_0.py
```

The rest of this section refers to the nine basic parsers (section 3.1).

The syntax definition is loaded from the file `binding_powers.json` unless specified otherwise (see options `-r` and `-d` below). Edit the definitions in this file if desired.

A basic parser can be run by

```
python3 PARSER_MODULE 'CODE'
```

where `PARSER_MODULE` is one of the basic parser modules and `CODE` is the code to be parsed. Example:

```
python3 pcp_rec_0_0.py 'xx + 5 ! * n ^ 2'
```

Use the correct interpreter name (e.g., `python` instead of `python3` if this is required). Enclose the code in single quotes (Linux) or double quotes (Windows?). Tokens are separated by whitespace, or by transition from an alphanumeric to a special character or vice versa. In this regard, the four characters `_`, `(`, `)`, `;` are considered alphanumeric. A minus sign immediately followed by a digit is also considered alphanumeric. Operands should be identifiers or integers (do not specify floating point numbers).

Use the option `-h` (with any basic parser) to find out all ways to run the parsers. There are several options that control the output – the output can be more verbose or more concise.

```
python3 pcp_ir_0.py -h
```

The output can, among others, contain a two-dimensional representation of the parse tree and indications of the *correctness* of the parsing. This is to facilitate experimentation.

> *Note:* The terms *precedence correctness* of parsing, *root operator weight*, *range* and *range correctness*, that may occur in the output, are not defined here. *Precedence correctness* is modelled after (but not identical to) the definition of this term by *Annika Aasa* in *User Defined Syntax* (1992) or *Precedences in Specifications and Implementations of Programming Languages* (1995).
> The result of parsing a valid expression should be *precedence correct* and *range correct*, and no other parse tree derived from this expression should be *precedence correct* or *range correct*.

The shorter call syntax `./PARSER_MODULE 'CODE'` may work, depending on the operating system and the shell. Set the *executable* flag of the parser module. Check the first line of the parser modules (the `#!`-line). An example for this call:

```
./pcp_it_0_1w.py '3 + 5! * 6^2'
```

3.2.1 Randomly generated expressions (option -r)

The command

```
python3 PARSER_MODULE -r [ nop [ nbp [ lexpr ] ] ]
```

will parse a generated expression containing *lexpr* infix operators which are taken randomly from a collection of up to *nop* operators. The *lbp* and *rbp* values of the operators are taken randomly and independently, from the range `6 ... 6+nbp-1`. Values that are not specified on the command line default to 6. The generated operators are of the form `(lbp;rbp)`, where `lbp` and `rbp` are the binding powers. E.g., `(6;8)` is an operator with `lbp=6`, `rbp=8`. The operands are denoted by `A0`, `A1`, ... For example,

```
python3 pcp_it_0_2w.py -r 4 3
```

could create and parse the expression

```
A0 (7;6) A1 (8;8) A2 (6;6) A3 (6;6) A4 (8;8) A5 (8;7 A6
```

with the operands `A0`, `A1`, ..., `A6` and the operators `(7;6)`, `(8;8)`, `(6;6)`, `(6;6)`, `(8;8)`, `(8;7)`. There are three binding powers (6 to 8) and four different operators: `(6;6)`, `(7;6)`, `(8;7)`, `(8;8)`. The total number of operators is six which is the default for the unspecified `lexpr` value.

Obviously, results obtained with option `-r` are not reproducible.

3.2.2 Expressions with explicitly specified binding powers (option `-d`)

The command

```
python3 PARSER_MODULE -d lbp1 rbp1, lbp2 rbp2, ..., lbpn rbpn
```

will parse an expression with operators `(lbp1;rbp1)` to `(lbpn;rbpn)` and operands `A0`, ..., `An`, where `lbpk`, `rbpk` are the binding powers of the `k`-th operator. All binding powers should be in range `6 ... 99`. For example,

```
python3 pcp_it_0_1w.py -d 7 8, 9 10
```

will create and parse the expression

```
A0 (7;8) A1 (9;10) A2
```

where `(7;8)` has `lbp=7` and `rbp=8`, and `(9;10)` has `lbp=9` and `rbp=10`.

Prefix and postfix operators are allowed. Use the help option (`-h`) for details.

3.2.3 The bash test script `run_tests.sh`

The `bash` shell script `run_tests.sh` reads and parses test codes from the file `basic_tests.txt` by the nine basic parsers. It should work on systems that support `bash` scripts. The script can be run without parameters:

```
./run_tests.sh
```

This will print detailed results. Run with option `-q` to get less verbose output:

```
./run_tests.sh -q
```

4. Structure of the source files. Dependencies

The software is in the following files: Ten parser modules (see 3.1), the modules `helpers.py` and `bintree.py`, the JSON file `binding_powers.json` (syntax), the shell script `run_tests.sh` and the file `basic_tests.txt` (test data).

Documentation is in this guide (`PARSING.md`), in `README.md` and in `LICENSE.txt`.

The parser modules are independent of each other. The basic parsers import functions and other definitions from the module `helpers`, e.g., the tokenizers and the test driver function `run_parser`. The `helpers` module in turn imports the class `FormatBinaryTree` from module `bintree`.

However, most of the Python code in `helpers.py` is not needed for the actual parsing. It is used for processing of the input, for presentation of the results, for correctness checks. For shorter, self-contained parser code, see `direct_pcp_ir_0.py` or the files in the gist `infix_parser.py` at https://gist.github.com/joekreu.

The parser modules invoke the test driver, passing the parse function and the corresponding tokenizer as parameters.

The `helpers` module uses the following items from system modules: `sys.argv`, `sys.executable`, `math.inf`, `os.path`, `collections.namedtuple`, `functools.reduce`, `random.randint`, `json.load`.

Comments in the code and data files provide additional information.

5. Acknowledgements and References

This repository was inspired by works on *precedence climbing* and *Pratt* parsing by *Theodore Norvell*, *Aleksey Kladov* (*matklad*), *Andy Chu*, *Eli Bendersky*, *Fredrik Lundh* (*effbot*), *Olivier Breuleux*, *Bob Nystrom* (*munificent*), *Dmitry A. Kazakov*, *Annika Aasa* and others.

The earliest reference to the simple iterative and recursive algorithm in `pcp_ir_0` that I know is *Keith Clarke* [8].

In the gist `op.py` *Olivier Breuleux* uses *dummy operands* and artificial (high) binding powers to virtually convert unary operators to infix operators (see [5]). This idea is explained in his text [6].

Dmitry Kazakov emphasizes the benefit of *two* priority values (i.e., what is called *lbp* and *rbp* here) instead of one priority and associativity. See posts of *Dmitry A. Kazakov* and *James Harris* at `compilers.iecc.com` [10].

The computer algebra systems *Maxima* and (now historic) *muMATH* use Pratt parsers based on binding powers. The assignment operator in these systems has an *lbp* of 180 and an *rbp* of 20.

The test for *precedence correctness* and the definitions of *operator ranges* (see the functions `_is_prec_correct`, `_lrange`, `_rrange` in the module `helpers.py`) are adapted from definitions by *Annika Aasa* [1], [2].

An incomplete list of references

[1] Annika Aasa, *Precedences in specifications and implementations of programming languages* (1995),
https://core.ac.uk/download/pdf/82260562.pdf

[2] Annika Aasa, *User defined syntax* (1992),
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3542

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

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

[5] Olivier Breuleux, `op.py`, https://gist.github.com/breuleux/6147321/.

[6] Olivier Breuleux, *Insert Language Name Here. How to make interesting little languages*, http://breuleux.net/blog/language-howto.html.

[7] Andy Chu, *Pratt Parsing and Precedence Climbing Are the Same Algorithm* (2016),
https://www.oilshell.org/blog/2016/11/01.html

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

[9] *Common operator notation* (*Wikipedia*, June 2022).

[10] James Harris, Dmitry A. Kazakov, *Compiling expressions* (2013),
posts on compilers.iecc.com, https://compilers.iecc.com/comparch/article/13-01-013

[11] Robert Jacobson, *Making a Pratt Parser Generator*,
https://www.robertjacobson.dev/designing-a-pratt-parser-generator

[12] Aleksey Kladov (matklad), *Simple but Powerful Pratt Parsing* (2020),
https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html

[13] Aleksey Kladov (matklad), *From Pratt to Dijkstra* (2020),
https://matklad.github.io/2020/04/15/from-pratt-to-dijkstra.html

[14] Fredrik Lundh (effbot), *Simple Top-Down Parsing in Python* (2008)

[15] Computer Algebra System Maxima, *Maxima Manual*, *Version 5.45.0*,
https://maxima.sourceforge.io/docs/manual/maxima.pdf
See especially section 7 (*Operators*).

[16] Theodore S. Norvell, *Parsing Expressions by Recursive Descent* (1999),
https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

[17] Theodore S. Norvell, *From Precedence Climbing to Pratt Parsing* (2016),
https://www.engr.mun.ca/~theo/Misc/pratt_parsing.htm

[18] Bob Nystrom, *Pratt Parsers: Expression Parsing Made Easy* (2011),
http://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/. Java code at https://github.com/munificent/bantam, C# code by John Cardinal at https://github.com/jfcardinal/BantamCs, and Python code by Ravener at https://github.com/ravener/bantam.py