# Precedence Climbing Parsing based on Binding Powers and Token Insertion in Python

## 1. Introduction

The repository contains several demo implementations of expression parsers based on *binding powers*, *precedence climbing* and *insertion of fake operands*.

The term *precedence climbing* is used here in a generic sense. The algorithms can be iterative, recursive, or mixed (with loops *and* recursion).

Very few lines of Python code are enough for the core of a parser that creates parse trees from operands and operators with virtually arbitrary binding powers.

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
```

The text PARSING is an extended version of most parts of this README.

### 1.1 Basics of precedence climbing parsing based on binding powers

#### 1.1.1 Binding powers

An infix operator has a *left* and a *right binding power* (*lbp* and *rbp*). Prefix operators have an *rbp*, and postfix operators have an *lbp*. Binding powers are numbers (often integers); they indicate the strength of binding (the *precedence*) to the left or to the right. Frequently, *lbp* and *rbp* of a specific operator are equal or differ only by one. They can, however, differ by any number, as long as they are in an allowed range (here: 6 to 99). Binding powers of unary operators can be smaller than the binding powers of infix operators in the same expression.

Parsing based on *lbp* and *rbp* (two numbers) can be more powerful than the approach based on *precedence* (in the specific sense: one number) and *associativity* (two values: *left* or *right*). Associativity can be expressed in terms of binding powers: An infix operator will be right associative if its *rbp* is less than its *lbp*.

#### 1.1.2 Unary operators, token insertion

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. The result is an alternating sequence of operands and infix operators, starting and ending with an operand.

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. A negative *rbp* is assigned to `$BEGIN` and a negative *lbp* to `$END`.

Insertion of tokens is done during the tokenization, i.e., by the *tokenizer*. The complete token sequence generated by the tokenizer for the example (1) is

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

#### 1.1.3 Results of parsing. Parse trees

The parsers return nested lists that represent parse trees. These lists can be formatted as Lisp-like *S-expressions*. E.g., parsing `5 + 3 ! * 4` will create the list `[+, 5, [*, [!, 3, $POST], 4]]`, or `(+ 5 (* (! 3 $POST) 4))` as S-expression. Without the fake operand `$POST` this is `(+ 5 (* (! 3) 4))`.

### 1.2 Main goals of the project

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.

### 1.3 Limitations

Exploring the full potential of precedence climbing parsing based on binding powers and token insertion is not the goal of this project. *Pratt parsing* (although related), *context-free grammars*, *Backus normal forms* and the like are not considered. The software does not contain *evaluators* for the parse trees.

## 2. Prerequisites

The Python interpreter (version 3.8 or higher) with the standard library is enough for the parsers. The `bash` shell is required for a test script.

## 3. Overview on the parsers

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

`pcp_ir_0` and `pcp_ir_0_no_ins` are *iterative* and *recursive*. Contrary to the general setting, the latter it is not based on token insertion. Instead, special code takes care of prefix and postfix operators.

`pcp_it_0_1w`, `pcp_it_0_1wg` and `pcp_it_0_2w` implement iterative algorithms.

`pcp_rec_0_0`, `pcp_rec_0_1`, `pcp_rec_0_2` and `pcp_rec_03` are recursive (without loops) and more or less *functional* (in the sense of functional programming).

All these parsers accept the same syntax for operator definitions.

Analysis of the code and test results support this claim:

*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*.

The remaining parser, `direct_pcp_ir_0`, uses the algorithm of `pcp_ir_0` to parse some 'hard coded' examples. It is 'self-contained' (no imports).

The software is contained in ten parser modules, 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 for the shell script).

## 4. Usage of the parsers

All project files should be placed in the same directory. Note that there is only minimal error handling. Run the parsers from the command line.

The parser `direct_pcp_ir_0.py` is simply run by

```
python3 direct_pcp_ir_0.py
```

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

The syntax definition is loaded from the file `binding_powers.json` unless one of the options `-r`, `-d` is in effect (see below). Edit the syntax definition if desired.

A parser can be run by

```
python3 PARSER_MODULE 'CODE'
```

where `PARSER_MODULE` is one of the basic parser modules. Example:

```
python3 pcp_rec_0_0.py '3 + 5! * 6^2'
```

Use `python` instead of `python3` if required. Enclose the code in single quotes or (on Windows?) double quotes. Tokens are separated by whitespace, or by transition from an alphanumeric to a special character or vice versa. In this regard, *underscore*, *parentheses* and *semicolon* are considered alphanumeric. A minus sign followed by a digit is also considered alphanumeric. Operands should be integers or identifiers.

This above command will parse the code by the specified parser and generate detailed output – among others, a two-dimensional representation of the parse tree and indications of *correctness* of the parsing. The output can be more concise or more verbose. Use the option `-h` (with any basic parser) to find out all ways to run the parsers, e.g., `python3 pcp_ir_0.py -h`

The call syntax `./PARSER_MODULE 'CODE'` may work, depending on the operating system and the shell. An example:

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

The `bash` shell script `run_tests.sh` reads test codes from the file `basic_tests.txt` and processes them with the nine basic parsers. It should work on systems that support `bash` scripts. Run the script without arguments, or provide the option `-q` (*quiet*, i.e., less verbose output).

### 4.1 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`. Non-specified values default to 6. An example:

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

Results of calls with option `-r` are not reproducible.

### 4.2 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
```

Unary operators can also be included. Use the help option (`-h`) for details.

## 5. License

The project is licensed under the terms of the MIT License. See LICENSE.

## 6. Acknowledgements

This project was inspired by works on precedence climbing and Pratt parsing by *Theodore Norvell*, *Aleksey Kladov* (*matklad*), *Andy Chu*, *Eli Bendersky*, *Fredrik Lundh* (*effbot*), *Annika Aasa* and others. See section 5 in the text PARSING.

The *correctness test* and the definitions of *operator ranges* (see the functions `_is_prec_correct`, `_lrange`, `_rrange` in `helpers.py`) are adapted from definitions by *Annika Aasa* in *Precedences in Specifications and Implementations of Programming Languages* (1995).