Simple Precedence Climbing Parsing Algorithms based on Binding Powers and Token Insertion

joekreu

February 2022

This 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.

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. For example, instead of simply fetching the next atomic operand from the token sequence, a whole parenthesized subexpression, or another *primary expression*, can be parsed recursively. This should be investigated separately.

Generally, precedence climbing parsing of expressions can be controlled by *precedence* (a number) and *associativity* (*left* or *right*) of operators, or alternatively by *binding powers*. In the latter case, 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*. Typically, *lbp* and *rbp* are 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 a way, the definition of general binding power based parsing is in the code of the parsers.

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
```

where `A0`, `A1`, ... are *atomic operands* and `Op1`, `Op2`, ... are *infix operators*. 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; e.g., from a text 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` will be parsed as `a + (b * c)`, not as `(a + b) * c`.

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 left associative. The exponentiation operator `^` is 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*.

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' binding powers are required to be less than 100.

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

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

User defined binding powers should be integers in range `6 to 99`. 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.

The upper bound of the allowed range, i.e., the number 99, results from the fake binding powers for unary operators (i.e., the value `100`). A lower bound is useful as well. For example, a negative *lbp* is assigned to the artificial `$END` token (see section 2). The benefit of other 'internal' binding powers becomes visible in more elaborate parsers. For example, the *comma* can possibly be parsed as an *infix operator* with very small binding powers.

The parsers return nested lists that represent parse trees; these lists can be formatted as Lisp-like *S-expressions*. Parsing `5 + 3 ! * 4` will create the list `[+, 5, [*, [!, 3, $POST], 4]]`, or `(+ 5 (* (! 3 $POST) 4))` as S-expression; omitting the fake operand `$POST` this will become `(+ 5 (* (! 3) 4))`.

Goals of this repository

1. Find and compare implementations of precedence climbing algorithms based on binding powers. Encourage experimentation.
2. Separate definition of parsing rules from the implementation of the parsers.
3. Better understand 'precedence correct' parsing.

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

2. Tokenization

In a first step, a *tokenizer* (*lexcial scanner*) creates a sequence of *tokens* from the input. In this repository, a token is always an atomic operand or an operator. A token may consist of one or more characters. In a way, the tokenizers in this repo are very primitive: Tokens must always be separated by spaces.

On the other hand, the tokenizers are also responsible for inserting the fake operands `$PRE` and `$POST`. 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 the `$BEGIN` and a negative *lbp* to the `$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 requirement for space-separation of tokens can be greatly relaxed.
> *Note 2:* The `$BEGIN` token is only referenced by the iterative parsers.

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.

In summary, tokenization in this repo splits the code at spaces, inserts the fake tokens `$PRE`, `$POST`, `$BEGIN` and `$END`, inserts missing binding powers, and provides an interface for the actual parsing.

3. Overview on the parsers. Notes on use

3.1 The individual parsers

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

1. `pcp_ir_0` is based on iteration (loops) *and* recursion.
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.
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), otherwise it 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 *linked list* of tokens.
8. `pcp_rec_0_2` is a recursive and purely functional parser. It uses a linked list of tokens. 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`.

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.

Analysis of the code and test results support this claim:
The nine 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.

The remaining parser is `direct_pcp_ir_0`. It has no dependencies; it parses some ‘hard coded’ examples. Its algorithm is that of `pcp_ir_0`.

3.2 Usage of the parsers

Use Python 3.5 or later. Put 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.

The parser `direct_pcp_ir_0.py` is simply run by

```
python direct_pcp_ir_0.py
```

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

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

A parser can be run by

```
python PARSER_MODULE 'CODE'
```

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

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

Use `python3` instead of `python` if required.

This input will generate detailed output – among others, a two-dimensional representation of the parse tree and indications of the *correctness* of the parsing.

> *Note:* The terms *correctness* of parsing, *root operator weight* and *range*, that may occur in the output, are not defined here. *Correctness* is modelled after (but not idential to) the definition of this term in *User Defined Syntax* by *Annika Aasa* (1992).

Enclose the code in single quotes (Linux) or double quotes (Windows). Place spaces between the tokens. Operators must be defined in `binding_powers.json`.

The call syntax `./PARSER_MODULE 'CODE'` may work, depending on the operating system and the shell. Set the *executable* flag of the parser module. An example:

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

Use option `-h` to find out all ways to run the parsers: `python pcp_ir_0.py -h`

The `bash` shell script `run_tests.sh` parses test codes from the file `basic_tests.txt` by the nine standard basic parsers. It should work on systems that support `bash` scripts. Run the script without arguments:

```
./run_tests.sh
```

3.2.1 Randomly generated expressions (option -r)

The command line syntax

```
python 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. 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`, … . E.g.,

```
python 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.

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

The command line syntax

```
python 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,

```
python 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`.

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`, the shell script `run_tests.sh` and the data file `basic_tests.txt`.

The parser modules are independent of each other. The standard 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`.

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

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

Binding power definitions are loaded from `binding_powers.json` and stored in the global Python dictionaries `LBP` and `RBP`.

Comments in the code and data files provide additional information.
