Precedence Climbing Parsing based on Binding Powers and Token Insertion

joekreu

March 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.

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.
>
> 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, for example, from a `csv`-file 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`.

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*, 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
```

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 possibly be parsed 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`.

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]]`, or `(+ 5 (* (! 3 $POST) 4))` as S-expression. Without the fake operand `$POST` this is `(+ 5 (* (! 3) 4))`. Fake operands can easily be removed from the parse tree.

Goals of this repository. Limitations

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.

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

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. In this repository, a token is always an atomic operand or an operator. A token may consist of one or more characters.

Tokens are separated by whitespace, or by transition from an alphanumeric to a special character or vice versa. A minus sign that is followed by a digit is considered alphanumeric.

Also, the four characters `_`, `(`, `)`, `;` are considered alphanumeric. This is because operators of type `(23;12)` or `(10;_)` will be generated if the parsers are run with options `-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` 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`. 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`.

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 sepearation of tokens (by whitespace or transition to another kind of characters) can be improved.
> *Note 2:* Only the iterative parsers (see 3.1) 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.

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.
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, i.e., it uses the `yield` statement instead of `return`. 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 *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. 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.

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' (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.

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.

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

```
python 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

```
python PARSER_MODULE 'CODE'
```

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

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

Use the correct interpreter name (e.g., `python3` instead of `python` 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 that is followed by a digit is also considered alphanumeric.

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

> *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 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 shorter 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 the option `-h` (with any basic parser) to find out all ways to run the parsers. There are severval options that control the output.

```
python pcp_ir_0.py -h
```

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. Run the script without arguments:

```
./run_tests.sh
```

3.2.1 Randomly generated expressions (option `-r`)

The command

```
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`. 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`, ... 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

```
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`.

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

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 (`DETAILED_GUIDE.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`.

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`, `math.inf`, `os.path`, `collections.namedtuple`, `functools.reduce`, `random.randint`, `json.load`.

Comments in the code and data files provide additional information.
