Precedence Climbing Parsing based on Binding Powers and Token Insertion

1. Introduction

The repository contains several demo implementations of 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 parsing algorithms are iterative, recursive, or mixed (based on loops *and* on recursion).

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
```

1.1 Parsing basics

1.1.1 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*. Typically, *lbp* and *rbp* are integers. Binding powers indicate the strength of binding in the corresponding direction. 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 do not have to be greater than the binding powers of infix operators in the same expression.

An infix operator will be right associative if its *rbp* is less than its *lbp*.

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.

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 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

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 The main goals of the project

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 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.

2. Prerequisites

Python 3.8 or higher. This is enough for the actual 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 and more or less *functional* (in the sense of functional programming).

All these parsers accept the same 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

Use Python 3.8 or later. Put all project files in one directory, or clone the repo. There is only minimal error handling. Run the parsers from the command line.

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 basic parsers.

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

A basic 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. Enclose the code in single quotes (Linux) or double quotes (Windows?). Place spaces between the tokens.

This input 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 the *correctness* of the parsing. This is to facilitate experimentation.

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'
```

The output can be more verbose or more concise. Use the option `-h` (with any basic parser module) to find out all ways to run the parsers:

```
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
```

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

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

Because of the random character results are not reproducible. See also

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

4.2 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`. An 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
```

Using unary operators is also possible. For more details see

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

5. License

6. Acknowledgements

This project was inspired by works of *T. S. Norvell*, *Aleksey Kladov* (*matklad*), *Andy Chu*, *Eli Bendersky*, *Jean-Marc Bourguet*, *Fredrik Lundh* (*effbot*), *Bob Nystrom* and others.

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

7. More information

See the text Detailed Guide in this repo.