The content has been updated and moved to `https://github.com/joekreu/`. 2022-04-22.

`minimalpcp`: Expression parsing by precedence climbing and token insertion

Last revised on 2018-06-05, refers to version 2017-04-23 of `minimalpcp.py` (for Python version 3.* or 2.7.*)

This document contains notes on the Python demo module `minimalpcp.py` (see j.e49.de/parsing).

Using *minimalpcp*, expressions consisting of atomic operands (numbers, identifiers) and operators (infix, unary prefix, unary postfix) can be parsed. Concepts of *Vaughan Pratt*, *Keith Clarke* and others are used.

Two input components are required: a character string containing the expression to be parsed, and a Python dictionary containing the *syntax*. The *syntax* consists of the operators together with their binding powers (numbers controlling the tightness of binding between operators and operands). The properties of particular operators are not hard-coded in the Python program. The operator methods `led` and `nud` that are often used in *Pratt parsing* (see below) are not needed here.

The *minimalpcp* parser returns a *parse tree* in the form of a nested Python list.

This implementation of *minimalpcp* cannot process control structures, definition and invocation of functions, subexpressions enclosed in parentheses (i.e., non-atomic primary expressions). It is not hard to extend the module in such a way that these parsing tasks can be performed.

The algorithm used in *minimalpcp* is now usually called *precedence climbing*. It is described in a paper by *Keith Clarke* (1986), see www.antlr.org/papers/Clarke-expr-parsing-1986.pdf. However, Clarke uses *precedence* (a number) and *associativity* (a boolean value) of operators instead of two binding powers (i.e. two numbers), and he treats parsing of unary operators differently.

Precedence climbing is similar to *Pratt parsing*, invented by *Vaughan Pratt* (see *Top Down Operator Precedence*, 1973; tdop.github.io). Pratt uses left and right binding powers. See also crockford.com/javascript/tdop/tdop.html by *Douglas Crockford*. Precedence climbing is also similar to the *shunting yard algorithm* invented by *Edsger W. Dijkstra*. Precedence climbing, Pratt parsing and the shunting yard algorithm are variants of *operator precedence parsing*.

Andy Chu's current blog contains a review of Pratt and precedence climbing parsing tutorials, see www.oilshell.org/blog/2016/11/02.html or www.oilshell.org/blog/2017/03/31.html.

Operators in *minimalpcp* have a *left* and a *right* binding power (LBP, RBP). The greater the LBP (or RBP), the tighter is the binding to the left (resp. right) operand.

A very simple tokenizer class (called `Tokenizer`) is included. The tokens are required to be white-space-separated in the string so that the list of tokens can be obtained by splitting the string at whitespace characters. This approach was chosen to make the code of this demo program short and self-contained. See usage examples in the source.

A *tokenizer* (an instance of the `Tokenizer` class) is the only input for the actual parser. The tokenizer encapsulates the string to be parsed and the dictionary of operator definitions. Global variables are not used. This should facilitate the coexistence of several tokenizers in one program.

The interface of the tokenizer (as used by the parse function `parseExpr`) consists of the attribute `Current` (current token), the method `Next` (advance by one token, and return the new current token), and the methods LBP and RBP (return binding powers of operators). A tokenizer instance is stateful. Its state is defined by the *current position* in the token sequence.
This tokenizer class can easily be replaced by a smarter one with the same interface.

Prefix and *postfix* operators add to the complexity of expression parsing. In *minimalpcp*, even prefix and postfix operators have *two* binding powers. A prefix operator has a certain very high LBP, and a postfix operator has a certain very high RBP. See test data in the source. The tokenizer inserts *dummy* (*virtual*) *operands* before prefix and after postfix operators. Thus an alternating sequence of

atomic operands and operators is obtained, and every operator can be parsed as an infix operator. The high binding power of prefix and postfix operators towards their respective dummy operands and the “real” binding power towards their “real” operands provide for the desired parsing.

This is one place where the use of *two* binding powers for an operator (instead of precedence and associativity) is more flexible. Another advantage of this approach is mentioned in *Dmitry A. Kazakov's* post (1/4/2013) in compgroups.net/comp.compilers/compiling-expressions/2145696: Sometimes “very” different LBP and RBP values of a certain infix operator are desirable (e.g., for assignment operators that return the assigned value).

Still another advantage of using two binding powers is the possibility to easily parse “bracketing constructs” with operands before or after them; e.g., in index expressions like `a[3]` or in “ternary operator” expressions like `a<10 ? b : c`. Here *dummy operators* with appropriate LBP and RBP can be inserted by the tokenizer (in the examples, between '`a`' and '`[`', between '`10`' and '`?`', and between '`:`' and '`c`'). In a post-processing step the resulting substructures in the parse tree can be converted to the desired form. - Insertion of *dummy operators* is not implemented in *minimalpcp*.

> ***Caution:*** I do not know an easy general definition of *correct parsing* of expressions containing infix, prefix and postfix operators, controlled by binding powers. The definition would probably be easier if prefix and postfix operators always had binding powers higher than any infix operator's binding power; but I think this would be too restrictive. Anyway, the actual parser, i.e. the code of `parseExpr`, produces results that seem to be at least reasonable. If some obvious conditions are met (e.g., *an atomic token is not followed by another atomic token*, or *an infix operator is not followed by an infix or a postfix operator*) a certain solution will be obtained. So, in a sense, one way to define correct parsing is given by this algorithm. A deeper analysis could probably reveal links to formal language theory. Can the parsing results of *minimalpcp* always be described as the result of an appropriately defined unambiguous grammar? See, e.g., *Precedences in specifications and implementations of programming languages* by *Annika Aasa* (1995), at www.sciencedirect.com/science/article/pii/030439759590680J.

The expression parser `parseExpr` consists of seven lines of code. It employs iteration and recursion. It needs to know the subsequent operator token when processing an atomic token. No further token lookahead is required. In the parse tree returned by `parseExpr` the dummy operands are kept to make the functionality of parsing better visible. They can easily be removed in post-processing.

The utility function `parse` combines tokenizer generation, parsing and output formatting.

The code in `minimalpcp.py` is meant to demonstrate only the basics. Tokenizer and parser together consist of about 35 lines of Python code (not counting comment lines, code for utility functions and test data). Here is a list of possible extensions, modifications and improvements:

1. Use a better tokenizer.
2. Add support for parsing of functions, parenthesized subexpressions and control structures. Add *operator insertion* to enable parsing of bracketing constructs with pre- or post operators. Enable the simultaneous use of the minus sign ('`-`') for prefix minus and infix minus.
3. Add some syntactic sugar. For example, insert an asterisk operator tokens ('`*`') between a number and a following identifier, or an `apply` token between two identifiers. That is, parse '3 x' as '3*x', or 'sin x' as 'sin`(x)`'.
4. Add error reporting: indicate the kind of error, the offending token and its position.
5. Obtain the syntax definition from an external source (e.g., a text, JSON or XML document). This syntax definition should be independent of the implementation language of the parser.

JoeCreu, 2018-06-05