Operator-precedence parser

From Wikipedia, the free encyclopedia

In computer science, an **operator precedence parser** is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation relying on order of operations to a format that is optimized for evaluation such as Reverse Polish notation (RPN).

Edsger Dijkstra's shunting yard algorithm is commonly used to implement operator precedence parsers. Other algorithms include the precedence climbing method and the top down operator precedence method.[1]

Relationship to other parsers [edit]

An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive nonterminals never appear in the right-hand side of any rule.

Operator-precedence parsers are not used often in practice; however they do have some properties that make them useful within a larger design. First, they are simple enough to write by hand, which is not generally the case with more sophisticated right shift-reduce parsers. Second, they can be written to consult an operator table at run time, which makes them suitable for languages that can add to or change their operators while parsing. (An example is Haskell, which allows user-defined infix operators with custom associativity and precedence; consequentially, an operator-precedence parser must be run on the program *after* parsing of all referenced modules.)

Perl 6 sandwiches an operator-precedence parser between two Recursive descent parsers in order to achieve a balance of speed and dynamism. This is expressed in the virtual machine for Perl 6, Parrot, as the Parser Grammar Engine (PGE). GCC's C and C++ parsers, which are hand-coded recursive descent parsers, are both sped up by an operator-precedence parser that can quickly examine arithmetic expressions. Operator precedence parsers are also embedded within compiler compiler-generated parsers to noticeably speed up the recursive descent approach to expression parsing.[2]

Precedence climbing method [edit]

The precedence climbing method is a compact, efficient, and flexible algorithm for parsing expressions that was first described by Martin Richards and Colin Whitby-Stevens.[3]

An infix-notation expression grammar in EBNF format will usually look like this:

```
    expression ::= equality-expression
    equality-expression ::= additive-expression ( ( '==' | '!=' ) additive-expression
) *
    additive-expression ::= multiplicative-expression ( ( '+' | '-' ) multiplicative-
expression ) *
    multiplicative-expression ::= primary ( ( '*' | '/' ) primary ) *
    primary ::= '(' expression ')' | NUMBER | VARIABLE | '-' primary
```

With many levels of precedence, implementing this grammar with a predictive recursive-descent parser can become inefficient. Parsing a number, for example, can require five function calls: one for each non-terminal in

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
فارسی
Polski
Edit links

the grammar until reaching *primary*.

An operator-precedence parser can do the same more efficiently.[2] The idea is that we can left associate the arithmetic operations as long as we find operators with the same precedence, but we have to save a temporary result to evaluate higher precedence operators. The algorithm that is presented here does not need an explicit stack; instead, it uses recursive calls to implement the stack.

The algorithm is not a pure operator-precedence parser like the Dijkstra shunting yard algorithm. It assumes that the *primary* nonterminal is parsed in a separate subroutine, like in a recursive descent parser.

Pseudo-code [edit]

The pseudo-code for the algorithm is as follows. The parser starts at function *parse_expression*. Precedence levels are greater than or equal to 0.

```
parse_expression ()
    return parse_expression_1 (parse_primary (), 0)
```

```
parse_expression_1 (lhs, min_precedence)
    lookahead := peek next token
    while lookahead is a binary operator whose precedence is >= min_precedence
        op := lookahead
        advance to next token
        rhs := parse_primary ()
        lookahead := peek next token
        while lookahead is a binary operator whose precedence is greater
                than op's, or a right-associative operator
                whose precedence is equal to op's
            rhs := parse_expression_1 (rhs, lookahead's precedence)
            lookahead := peek next token
        lhs := the result of applying op with operands lhs and rhs
    return lhs
```

Note that in the case of a production rule like this (where the operator can only appear once):

```
equality-expression ::= additive-expression  ( '==' | '!=' ) additive-expression
```

the algorithm must be modified to accept only binary operators whose precedence is > *min_precedence*.

Example execution of the algorithm [edit]

An example execution on the expression 2 + 3 * 4 + 5 == 19 is as follows. We give precedence 0 to equality expressions, 1 to additive expressions, 2 to multiplicative expressions.

parse_expression_1 (*lhs* = 2, *min_precedence* = 0)

- the lookahead token is +, with precedence 1. the outer while loop is entered.
- *op* is + (precedence 1) and the input is advanced
- *rhs* is 3
- the lookahead token is *, with precedence 2. the inner while loop is entered.
  parse_expression_1 (*lhs* = 3, *min_precedence* = 2)
  - the lookahead token is *, with precedence 2. the outer while loop is entered.
    - *op* is * (precedence 2) and the input is advanced
    - *rhs* is 4
    - the next token is +, with precedence 1. the inner while loop is not entered.
    - *lhs* is assigned 3*4 = 12
    - the next token is +, with precedence 1. the outer while loop is left.
  - 12 is returned.
- the lookahead token is +, with precedence 1. the inner while loop is not entered.
- *lhs* is assigned 2+12 = 14
- the lookahead token is +, with precedence 1. the outer while loop is not left.
- *op* is + (precedence 1) and the input is advanced
- *rhs* is 5

- the next token is ==, with precedence 0. the inner while loop is not entered.
- *lhs* is assigned 14+5 = 19
- the next token is ==, with precedence 0. the outer while loop is not left.
- *op* is == (precedence 0) and the input is advanced
- *rhs* is 19
- the next token is *end-of-line*, which is not an operator. the inner while loop is not entered.
- *lhs* is assigned the result of evaluating 19 == 19, for example 1 (as in the C standard).
- the next token is *end-of-line*, which is not an operator. the outer while loop is left.

1 is returned.

Alternative methods [edit]

There are other ways to apply operator precedence rules. One is to build a tree of the original expression and then apply tree rewrite rules to it.

Such trees do not necessarily need to be implemented using data structures conventionally used for trees. Instead, tokens can be stored in flat structures, such as tables, by simultaneously building a priority list which states what elements to process in which order.

Another approach is to first fully parenthesize the expression, inserting a number of parentheses around each operator, such that they lead to the correct precedence even when parsed with a linear, left-to-right parser. This algorithm was used in the early FORTRAN I compiler.[*citation needed*]

Example code of a simple C application that handles parenthesisation of basic math operators (+, -, *, /, ^ and parentheses):

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
  int i;
  printf("((((");
  for(i=1;i!=argc;i++){
    if(argv[i] && !argv[i][1]){
      switch(*argv[i]){
          case '(': printf("((((("); continue;
          case ')': printf("))))"); continue;
          case '^': printf(")^("); continue;
          case '*': printf("))*(("); continue;
          case '/': printf("))/(("); continue;
          case '+':
            if (i == 1 || strchr("(^*/+-", *argv[i-1]))
              printf("+");
            else
              printf(")))+(((");
            continue;
          case '-':
            if (i == 1 || strchr("(^*/+-", *argv[i-1]))
              printf("-");
            else
              printf(")))-(((");
            continue;
      }
    }
    printf("%s", argv[i]);
  }
  printf("))))\n");
  return 0;
}
```

For example, when compiled and invoked from the command line with parameters

```
a * b + c ^ d / e
```

it produces

```
((((a))*((b)))+(((c)^(d))/((e))))
```

as output on the console.

A limitation to this strategy is that unary operators must all have higher precedence than infix operators. The "negative" operator in the above code has a higher precedence than exponentiation. Running the program with this input

```
- a ^ 2
```

produces this output

```
((((-a)^(2))))
```

which is probably not what is intended.

References [edit]

1. ^ Norvell, Theodore (2001). "Parsing Expressions by Recursive Descent". Retrieved 2012-01-24.
2. ^ [a] [b] Harwell, Sam (2008-08-29). "Operator precedence parser". ANTLR3 Wiki. Retrieved 2012-01-24.
3. ^ Richards, Martin; Whitby-Stevens, Colin (1979). *BCPL — the language and its compiler*. Cambridge University Press. ISBN 9780521219655.

External links [edit]

- Clarke, Keith (1992-05-26). "Re: compact recursive-descent parsing of expressions". Retrieved 2012-01-24.
- Example C++ code by Keith Clarke for parsing infix expressions using the precedence climbing method
- Samelson, Klaus; Friedrich L. Bauer (February 1960). "Sequential formula translation". *Communications of the ACM* **3** (2): 76–83. doi:10.1145/366959.366968.

Categories: Parsing algorithms