Operator-precedence parser

From Wikipedia, the free encyclopedia

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 with order of operations format into an internally optimized computer-readable format like Reverse Polish notation (RPN).

Edsger Dijkstra's shunting yard algorithm is commonly used to implement operator precedence parsers which convert from infix notation to RPN.

Contents

Relationship to other parsers

An operator-precedence parser is a simple shift-reduce parser 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 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.

Perl 6 "sandwiches" an operator-precedence parser in 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.

To show that one grammar is **operator precedence**, first it should be **operator grammar.** Operator precedence grammar is the only grammar which can construct the parse tree even though the given grammar is ambiguous.

Example algorithm known as precedence climbing to parse infix

notation

An EBNF grammar that parses infix notation 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 the grammar, until we reach *primary*).

An operator-precedence parser can do the same more efficiently. 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

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

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

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

Example execution of the algorithm

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 next token is +, with precedence 1. the while loop is entered.

- *op* is + (precedence 1)
- *rhs* is 3
- the next token is *, with precedence 2. recursive invocation.
  parse_expression_1 (*lhs* = 3, *min_precedence* = 2)
    - the next token is *, with precedence 2. the while loop is entered.
      - *op* is * (precedence 2)
      - *rhs* is 4
      - the next token is +, with precedence 1. no recursive invocation.
      - *lhs* is assigned 3*4 = 12
      - the next token is +, with precedence 1. the while loop is left.
    - 12 is returned.
- the next token is +, with precedence 1. no recursive invocation.
- *lhs* is assigned 2+12 = 14
- the next token is +, with precedence 1. the while loop is not left.
- *op* is + (precedence 1)
- *rhs* is 5
- the next token is ==, with precedence 0. no recursive invocation.
- *lhs* is assigned 14+5 = 19
- the next token is ==, with precedence 0. the while loop is not left.
- *op* is == (precedence 0)
- *rhs* is 19
- the next token is *end-of-line*, which is not an operator. no recursive invocation.
- *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 while loop is left.

1 is returned.

Alternatives to Dijkstra's Algorithm

There are alternative ways to apply operator precedence rules which do not involve the Shunting Yard algorithm.

One is to build a tree of the original expression, 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. As an example, such an approach is used in the *Mathematical Formula Decomposer*.[1]

Another approach is to first fully parenthesise 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>

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 command line with parameters

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

it produces

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

as output on the console.

References

1. ^ Mathematical Formula Decomposer (http://herbert.gandraxa.com/herbert/mfd.asp)

External links

- Parsing Expressions by Recursive Descent (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm) Theodore Norvell (C) 1999-2001. Access data September 14, 2006.
- Sequential formula translation. (http://portal.acm.org/citation.cfm?id=366968) Samelson, K. and Bauer, F. L. 1960. Commun. ACM 3, 2 (Feb. 1960), 76-83. DOI= http://doi.acm.org/10.1145/366959.366968