Equation (expression) parser with precedence? [closed]

Asked 16 years, 3 months ago Modified 10 months ago

Viewed 104k times

122

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 11 months ago.

Improve this question

I've developed an equation parser using a simple stack algorithm that will handle binary (+, -, |, &, *, /, etc) operators, unary (!) operators, and parenthesis.

Using this method, however, leaves me with everything having the same precedence - it's evaluated left to right regardless of operator, although precedence can be enforced using parenthesis.

So right now "1+11*5" returns 60, not 56 as one might expect.

While this is suitable for the current project, I want to have a general purpose routine I can use for later projects.

Edited for clarity:

What is a good algorithm for parsing equations with precedence?

I'm interested in something simple to implement and understand that I can code myself to avoid licensing issues with available code.

Grammar:

I don't understand the grammar question - I've written this by hand. It's simple enough that I don't see the need for YACC or Bison. I merely need to calculate strings with equations such as "2+3 * (42/13)".

Language:

I'm doing this in C, but I'm interested in an algorithm, not a language specific solution. C is low level enough that it'll be easy to convert to another language should the need arise.

Code Example

I posted the test code for the simple expression parser I was talking about above. The project requirements altered and so I never needed to optimize the code for performance or space as it wasn't incorporated into the

project. It's in the original verbose form, and should be readily understandable. If I do anything further with it in terms of operator precedence, I'll probably choose the macro hack because it matches the rest of the program in simplicity. If I ever use this in a real project, though, I'll be going for a more compact/speedy parser.

Related question

> Smart design of a math parser?

algorithm parsing equation

Share

Improve this question

Follow

edited Feb 2 at 14:28

desertnaut

60.2k ● 31 ● 151 ● 176

asked Aug 26, 2008 at 14:52

Adam Davis

93.5k ● 60 ● 271 ● 333

I have writen an expression parser in C# on my blog. It does infix to postfix without the stack in the shunting yard algorithm. It only uses an array. – Guge Dec 5, 2008 at 21:35

As i understand you need parse only arithmetic expression. Use Reverse Polish Notation – mishadoff Nov 1, 2011 at 11:41

16 Answers Sorted by: Highest score (default)

184

The shunting yard algorithm is the right tool for this. Wikipedia is really confusing about this, but basically the algorithm works like this:

Say, you want to evaluate 1 + 2 * 3 + 4. Intuitively, you "know" you have to do the 2 * 3 first, but how do you get this result? The key is to realize that when you're scanning the string from left to right, you will evaluate an operator when the operator that **follows** it has a lower (or equal to) precedence. In the context of the example, here's what you want to do:

1. Look at: 1 + 2, don't do anything.
2. Now look at 1 + 2 * 3, still don't do anything.
3. Now look at 1 + 2 * 3 + 4, now you know that 2 * 3 has to to be evaluated because the next operator has lower precedence.

How do you implement this?

You want to have two stacks, one for numbers, and another for operators. You push numbers onto the stack all the time. You compare each new operator with the one at the top of the stack, if the one on top of the stack has higher priority, you pop it off the operator stack, pop the operands off the number stack, apply the operator and push the result onto the number stack. Now you repeat the comparison with the top of stack operator.

Coming back to the example, it works like this:

N = [ ] Ops = [ ]

- Read 1. N = [1], Ops = [ ]
- Read +. N = [1], Ops = [+]
- Read 2. N = [1 2], Ops = [+]
- Read `*`. N = [1 2], Ops = [+ *]
- Read 3. N = [1 2 3], Ops = [+ *]
- Read +. N = [1 2 3], Ops = [+ *]
  - Pop 3, 2 and execute 2`*`3, and push result onto N. N = [1 6], Ops = [+]
  - `+` is left associative, so you want to pop 1, 6 off as well and execute the +. N = [7], Ops = [].
  - Finally push the [+] onto the operator stack. N = [7], Ops = [+].
- Read 4. N = [7 4]. Ops = [+].
- You're run out off input, so you want to empty the stacks now. Upon which you will get the result 11.

There, that's not so difficult, is it? And it makes no invocations to any grammars or parser generators.

edited Jun 24, 2009 at 22:27

Dour High Arch

21.7k ● 30 ● 77 ● 93

answered Sep 6, 2008 at 18:38

Pramod
9,456 ● 4 ● 27 ● 28

8 You don't actually need two stacks, as long as you can see the second thing on the stack without popping the top. You can instead use a single stack that alternates numbers and operators. This in fact corresponds to exactly what an LR parser generator (such as bison) does. – Chris Dodd Apr 28, 2009 at 21:19

2 Really nice explanation of the algorithm I just implemented right now. Also you are not converting it to postfix which is also nice. Adding support for parenthesis is very easy too. – Giorgi Sep 29, 2010 at 21:05

4 A simplified version for the shunting-yard algorithm can be found here: andreinc.net/2010/10/05/… (with implementations in Java and python) – Andrei Ciobanu Oct 6, 2010 at 6:53

Thanks a lot for mention about left - associative. I stuck with ternary operator: how to parse complex expressions with nested "?:". I realized that both '?' and ':' have to have the same priority. And if we interpret '?' as right - associative and ':' as left - associative this algorithm works with them very well. Also, we can collapse 2 operators only when both of them are left - associative. – Vladislav Aug 8, 2019 at 21:25

71

The hard way

You want a recursive descent parser.

To get precedence you need to think recursively, for example, using your sample string,

```
1+11*5
```

to do this manually, you would have to read the `1`, then see the plus and start a whole new recursive parse "session" starting with `11`... and make sure to parse the `11 * 5` into its own factor, yielding a parse tree with `1 + (11 * 5)`.

This all feels so painful even to attempt to explain, especially with the added powerlessness of C. See, after parsing the 11, if the * was actually a + instead, you would have to abandon the attempt at making a term and instead parse the `11` itself as a factor. My head is already exploding. It's possible with the recursive decent strategy, but there is a better way...

The easy (right) way

If you use a GPL tool like Bison, you probably don't need to worry about licensing issues since the C code generated by bison is not covered by the GPL (IANAL but I'm pretty sure GPL tools don't force the GPL on generated code/binaries; for example Apple compiles code like say, Aperture with GCC and they sell it without having to GPL said code).

Download Bison (or something equivalent, ANTLR, etc.).

There is usually some sample code that you can just run bison on and get your desired C code that demonstrates this four function calculator:

http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html

Look at the generated code, and see that this is not as easy as it sounds. Also, the advantages of using a tool like Bison are 1) you learn something (especially if you read the Dragon book and learn about grammars), 2) you avoid NIH trying to reinvent the wheel. With a real parser-generator tool, you actually have a hope at scaling up later, showing other people you know that parsers are the domain of parsing tools.

Update:

People here have offered much sound advice. My only warning against skipping the parsing tools or just using the Shunting Yard algorithm or a hand rolled recursive decent parser is that little toy languages[1] may someday turn into big actual languages with functions (sin, cos, log) and variables, conditions and for loops.

Flex/Bison may very well be overkill for a small, simple interpreter, but a one off parser+evaluator may cause trouble down the line when changes need to be made or features need to be added. Your situation will vary and you will need to use your judgement; just don't punish other people for your sins [2] and build a less than adequate tool.

My favorite tool for parsing

The best tool in the world for the job is the Parsec library (for recursive decent parsers) which comes with the programming language Haskell. It looks a lot like BNF, or like some specialized tool or domain specific language for parsing (sample code [3]), but it is in fact just a regular library in Haskell, meaning that it compiles in the same build step as the rest of your Haskell code, and you can write arbitrary Haskell code and call that within your parser, and you can mix and match other libraries *all in the same code*. (Embedding a parsing language like this in a language other than Haskell results in loads of syntactic cruft, by the way. I did this in C# and it works quite well but it is not so pretty and succinct.)

Notes:

1 Richard Stallman says, in Why you should not use Tcl

> The principal lesson of Emacs is that a language for extensions should not be a mere "extension language". It should be a real programming language, designed for writing and maintaining substantial programs. Because people will want to do that!

[2] Yes, I am forever scarred from using that "language".

Also note that when I submitted this entry, the preview was correct, but **SO's less than adequate parser ate my close anchor tag on the first paragraph**, proving that parsers are not something to be trifled with because

if you use regexes and one off hacks **you will probably get something subtle and small wrong**.

[3] Snippet of a Haskell parser using Parsec: a four function calculator extended with exponents, parentheses, whitespace for multiplication, and constants (like pi and e).

```
aexpr   =   expr `chainl1` toOp
expr    =   optChainl1 term addop (toScalar 0)
term    =   factor `chainl1` mulop
factor  =   sexpr  `chainr1` powop
sexpr   =   parens aexpr
        <|> scalar
        <|> ident

powop   =   sym "^" >>= return . (B Pow)
        <|> sym "^-" >>= return . (\x y -> B Pow x (B

toOp    =   sym "->" >>= return . (B To)

mulop   =   sym "*" >>= return . (B Mul)
        <|> sym "/" >>= return . (B Div)
        <|> sym "%" >>= return . (B Mod)
        <|>             return . (B Mul)

addop   =   sym "+" >>= return . (B Add)
        <|> sym "-" >>= return . (B Sub)

scalar = number >>= return . toScalar

ident  = literal >>= return . Lit

parens p = do
             lparen
             result <- p
             rparen
             return result
```

Share Improve this answer Follow

edited Feb 14, 2016 at 12:49

community wiki
18 revs, 13 users 52%
Jared Updike

9 To emphasize my point, note that the markup in my post is not getting parsed correctly (and this varies between the markup rendered statically and that rendered in the WMD preview). There have been several attempts to fix it but I think THE PARSER IS WRONG. Do everyone a favor and get parsing right! – Jared Updike Jun 23, 2009 at 23:47

16

Long time ago, I made up my own parsing algorithm, that I couldn't find in any books on parsing (like the Dragon Book). Looking at the pointers to the Shunting Yard algorithm, I do see the resemblance.

About 2 years ago, I made a post about it, complete with Perl source code, on http://www.perlmonks.org/?node_id=554516. It's easy to port to other languages: the first implementation I did was in Z80 assembler.

It's ideal for direct calculation with numbers, but you can use it to produce a parse tree if you must.

Update Because more people can read (or run) Javascript, I've reimplemented my parser in Javascript, after the code has been reorganized. The whole parser is under 5k of Javascript code (about 100 lines for the

parser, 15 lines for a wrapper function) including error reporting, and comments.

You can find a live demo at http://users.telenet.be/bartl/expressionParser/expressionParser.html.

```
// operator table
var ops = {
   '+'  : {op: '+', precedence: 10, assoc: 'L', exec:
l+r; } },
   '-'  : {op: '-', precedence: 10, assoc: 'L', exec:
r; } },
   '*'  : {op: '*', precedence: 20, assoc: 'L', exec:
l*r; } },
   '/'  : {op: '/', precedence: 20, assoc: 'L', exec:
l/r; } },
   '**' : {op: '**', precedence: 30, assoc: 'R', exec:
Math.pow(l,r); } }
};

// constants or variables
var vars = { e: Math.exp(1), pi: Math.atan2(1,1)*4 };

// input for parsing
// var r = { string: '123.45+33*8', offset: 0 };
// r is passed by reference: any change in r.offset is
// functions return the parsed/calculated value
function parseVal(r) {
    var startOffset = r.offset;
    var value;
    var m;
    // floating point number
    // example of parsing ("lexing") without aid of re
    value = 0;
    while("0123456789".indexOf(r.string.substr(r.offse
r.string.length) r.offset++;
    if(r.string.substr(r.offset, 1) == ".") {
        r.offset++;
        while("0123456789".indexOf(r.string.substr(r.o
r.offset < r.string.length) r.offset++;
```

```
    }
    if(r.offset > startOffset) {  // did that work?
        // OK, so I'm lazy...
        return parseFloat(r.string.substr(startOffset,
    } else if(r.string.substr(r.offset, 1) == "+") {
        r.offset++;
        return parseVal(r);
    } else if(r.string.substr(r.offset, 1) == "-") {
        r.offset++;
        return negate(parseVal(r));
    } else if(r.string.substr(r.offset, 1) == "(") {
        r.offset++;   // eat "("
        value = parseExpr(r);
        if(r.string.substr(r.offset, 1) == ")") {
            r.offset++;
            return value;
        }
        r.error = "Parsing error: ')' expected";
        throw 'parseError';
    } else if(m = /^[a-z_][a-z0-9_]*/i.exec(r.string.s
variable/constant name
        // sorry for the regular expression, but I'm t
a varname lexer
        var name = m[0];  // matched string
        r.offset += name.length;
        if(name in vars) return vars[name];  // I know
        r.error = "Semantic error: unknown variable '"
        throw 'unknownVar';
    } else {
        if(r.string.length == r.offset) {
            r.error = 'Parsing error at end of string:
            throw 'valueMissing';
        } else  {
            r.error = "Parsing error: unrecognized val
            throw 'valueNotParsed';
        }
    }
}

function negate (value) {
    return -value;
}

function parseOp(r) {
```

```
    if(r.string.substr(r.offset,2) == '**') {
        r.offset += 2;
        return ops['**'];
    }
    if("+-*/".indexOf(r.string.substr(r.offset,1)) >=
        return ops[r.string.substr(r.offset++, 1)];
    return null;
}

function parseExpr(r) {
    var stack = [{precedence: 0, assoc: 'L'}];
    var op;
    var value = parseVal(r);  // first value on the le
    for(;;){
        op = parseOp(r) || {precedence: 0, assoc: 'L'}
        while(op.precedence < stack[stack.length-1].pr
              (op.precedence == stack[stack.length-1].
'L')) {
            // precedence op is too low, calculate wit
left, first
            var tos = stack.pop();
            if(!tos.exec) return value;  // end  reach
            // do the calculation ("reduce"), producin
            value = tos.exec(tos.value, value);
        }
        // store on stack and continue parsing ("shift
        stack.push({op: op.op, precedence: op.preceden
exec: op.exec, value: value});
        value = parseVal(r);  // value on the right
    }
}

function parse (string) {   // wrapper
    var r = {string: string, offset: 0};
    try {
        var value = parseExpr(r);
        if(r.offset < r.string.length){
          r.error = 'Syntax error: junk found at offse
            throw 'trailingJunk';
        }
        return value;
    } catch(e) {
        alert(r.error + ' (' + e + '):\n' + r.string.s
'<*>' + r.string.substr(r.offset));
```

```
            return;
        }
    }
```

Share Improve this answer

Follow

edited May 12, 2014 at 7:44

wilx

18.2k ● 7 ● 64 ● 120

answered Sep 22, 2008 at 13:51

bart

7,759 ● 3 ● 35 ● 40

14

It would help if you could describe the grammar you are currently using to parse. Sounds like the problem might lie there!

Edit:

The fact that you don't understand the grammar question and that 'you've written this by hand' very likely explains why you're having problems with expressions of the form '1+11*5' (i.e., with operator precedence). Googling for 'grammar for arithmetic expressions', for example, should yield some good pointers. Such a grammar need not be complicated:

```
<Exp> ::= <Exp> + <Term> |
          <Exp> - <Term> |
          <Term>

<Term> ::= <Term> * <Factor> |
           <Term> / <Factor> |
           <Factor>
```

```
<Factor> ::= x | y | ... |
             ( <Exp> ) |
             - <Factor> |
             <Number>
```

would do the trick for example, and can be trivially augmented to take care of some more complicated expressions (including functions for example, or powers,...).

I suggest you have a look at this thread, for example.

Almost all introductions to grammars/parsing treat arithmetic expressions as an example.

Note that using a grammar does not at all imply using a specific tool (*a la* Yacc, Bison,...). Indeed, you most certainly are already using the following grammar:

```
<Exp>  :: <Leaf> | <Exp> <Op> <Leaf>

<Op>   :: + | - | * | /

<Leaf> :: <Number> | (<Exp>)
```

(or something of the kind) without knowing it!

Share Improve this answer Follow

edited May 23, 2017 at 11:54

Community Bot

1 ● 1

answered Aug 26, 2008 at 14:59

OysterD
6,750 ● 5 ● 35 ● 33

8

Have you thought about using Boost Spirit? It allows you to write EBNF-like grammars in C++ like this:

```
group       = '(' >> expression >> ')';
factor      = integer | group;
term        = factor >> *(('*' >> factor) | ('/' >> fa
expression  = term >> *(('+' >> term) | ('-' >> term))
```

Share Improve this answer

Follow

edited May 12, 2014 at 7:57

wilx
18.2k ● 7 ● 64 ● 120

answered Apr 28, 2009 at 20:36

Zifre
27k ● 12 ● 86 ● 106

1 +1 And the upshot is, everything is part of Boost. The grammar for the calculator is here: spirit.sourceforge.net/distrib/spirit_1_8_5/libs/spirit/example/…. The implementation of the calculator is here: spirit.sourceforge.net/distrib/spirit_1_8_5/libs/spirit/example/…. And the documentation is here: spirit.sourceforge.net/distrib/spirit_1_8_5/libs/spirit/doc/…. I will never understand why people still implement there own mini-parsers. – stephan Jun 20, 2009 at 13:54

As you put your question there is no need for recursion whatsoever. The answer is three things: Postfix notation

6

plus Shunting Yard algorithm plus Postfix expression evaluation:

1). Postfix notation = invented to eliminate the need for explicit precedence specification. Read more on the net but here is the gist of it: infix expression (1 + 2) * 3 while easy for humans to read and process not very efficient for computing via machine. What is? Simple rule that says "rewrite expression by caching in precedence,then always process it left-to-right". So infix (1 + 2) * 3 becomes a postfix 12+3*. POST because operator is placed always AFTER the operands.

2). Evaluating postfix expression. Easy. Read numbers off postfix string. Push them on a stack until an operator is seen. Check operator type - unary? binary? tertiary? Pop as many operands off stack as needed to evaluate this operator. Evaluate. Push result back on stack! And u r almost done. Keep doing so until stack has only one entry = value u r looking for.

Let's do (1 + 2) * 3 which is in postfix is "12+3*". Read first number = 1. Push it on stack. Read next. Number = 2. Push it on stack. Read next. Operator. Which one? +. What kind? Binary = needs two operands. Pop stack twice = argright is 2 and argleft is 1. 1 + 2 is 3. Push 3 back on stack. Read next from postfix string. Its a number. 3.Push. Read next. Operator. Which one? *. What kind? Binary = needs two numbers -> pop stack twice. First pop into argright, second time into argleft. Evaluate operation - 3 times 3 is 9.Push 9 on stack. Read

next postfix char. It's null. End of input. Pop stack onec = that's your answer.

3). Shunting Yard is used to transform human (easily) readable infix expression into postfix expression (also human easily readable after some practice). Easy to code manually. See comments above and net.

Share Improve this answer

Follow

answered Jun 9, 2012 at 0:07

CisBestLanguageOfAllTimes

61 ● 1 ● 1

5

I would suggest cheating and using the Shunting Yard Algorithm. It's an easy means of writing a simple calculator-type parser and takes precedence into account.

If you want to properly tokenise things and have variables, etc. involved then I would go ahead and write a recursive descent parser as suggested by others here, however if you simply require a calculator-style parser then this algorithm should be sufficient :-)

Share Improve this answer

Follow

answered Aug 28, 2008 at 18:53

ljs

37.8k ● 36 ● 109 ● 124

Another resource for precedence parsing is the Operator-precedence parser entry on Wikipedia. Covers Dijkstra's

5

shunting yard algorithm, and a tree alternate algorithm, but more notably covers a really simple macro replacement algorithm that can be trivially implemented in front of any precedence ignorant parser:

```
#include <stdio.h>
int main(int argc, char *argv[]){
  printf("((((");
  for(int 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;
      }
    }
    printf("%s", argv[i]);
  }
  printf("))))\n");
  return 0;
}
```

Invoke it as:

```
$ cc -o parenthesise parenthesise.c
$ ./parenthesise a \* b + c ^ d / e
((((a))*((b)))+(((c)^(d))/((e))))
```

Which is awesome in its simplicity, and very understandable.

Share Improve this answer

Follow

edited May 12, 2014 at 7:43

wilx

18.2k ● 7 ● 64 ● 120

answered Apr 23, 2009 at 19:10

Adam Davis
93.5k ● 60 ● 271 ● 333

4 That's quite a nice little pearl. But extending it (say, with function application, implicit multiplication, prefix and postfix operators, optional type annotations, anything) would break the whole thing. In other words, it's an elegant hack. – Jared Updike Nov 3, 2009 at 18:47

I don't see the point. All this does is change an operator-precedence parsing problem into a parentheses-precedence parsing problem. – user207421 Jun 30, 2015 at 2:05

2 @EJP sure, but the parser in the question handles parenthesis just fine, so this is a reasonable solution. If you have a parser that doesn't, though, then you are correct that this just moves the problem to another area. – Adam Davis Jun 30, 2015 at 2:18

4

It depends on how "general" you want it to be.

If you want it to be really really general such as be able to parse mathematical functions as well like sin(4+5)*cos(7^3) you will probably need a **parse tree.**

In which, I do not think that a complete implementation is proper to be pasted here. I'd suggest that you check out one of the infamous "Dragon book".

But if you just want precedence support, then you could do that by first converting the expression to postfix form in which an algorithm that you can copy-and-paste

should be available from google or I think you can code it up yourself with a binary tree.

When you have it in postfix form, then it's piece of cake from then on since you already understand how the stack helps.

Share Improve this answer

Follow

answered Aug 26, 2008 at 15:06

chakrit

61.5k ● 25 ● 136 ● 163

The dragon book might be a little excessive for an expression evaluator - a simple recursive descent parser is all that's needed, but it's a must read if you want to do anything more extensive in compilers. – Eclipse Jun 23, 2009 at 23:22

1 Wow - it's nice to know that the "Dragon book" is still discussed. I remember studying it - and reading it all through - at university, 30 years ago. – Schroedingers Cat Apr 8, 2015 at 8:01

4

I found this on the PIClist about the Shunting Yard algorithm:

> Harold writes:
>
> > I remember reading, a long time ago, of an algorithm that converted algebraic expressions to RPN for easy evaluation. Each infix value or operator or parenthesis was represented by a railroad car on a

> track. One type of car split off to another track and the other continued straight ahead. I don't recall the details (obviously!), but always thought it would be interesting to code. This is back when I was writing 6800 (not 68000) assembly code.

This is the "shunting yard algorythm" and it is what most machine parsers use. See the article on parsing in Wikipedia. An easy way to code the shunting yard algorythm is to use two stacks. One is the "push" stack and the other the "reduce" or "result" stack. Example:

pstack = () // empty rstack = () input: 1+2*3 precedence = 10 // lowest reduce = 0 // don't reduce

start: token '1': isnumber, put in pstack (push) token '+': isoperator set precedence=2 if precedence < previous_operator_precedence then reduce() // see below put '+' in pstack (push) token '2': isnumber, put in pstack (push) token '*': isoperator, set precedence=1, put in pstack (push) // check precedence as // above token '3': isnumber, put in pstack (push) end of input, need to reduce (goal is empty pstack) reduce() //done

to reduce, pop elements from the push stack and put them into the result stack, always swap the

top 2 items on pstack if they are of the form 'operator' 'number':

pstack: '1' '+' '2' '' *'3' rstack: () ... pstack: () rstack: '3' '2'* '' '1' '+'

if the expression would have been:

1*2+3

then the reduce trigger would have been the reading of the token '+' which has lower precendece than the '*' already pushed, so it would have done:

pstack: '1' '' *'2' rstack: () ... pstack: () rstack: '1' '2'* ''

and then pushed '+' and then '3' and then finally reduced:

pstack: '+' '3' rstack: '1' '2' '' *... pstack: () rstack: '1' '2'* '' '3' '+'

So the short version is: push numbers, when pushing operators check the precedence of the previous operator. If it was higher than the operator's that is to be pushed now, first reduce, then push the current operator. To handle parens simply save the precedence of the 'previous' operator, and put a mark on the pstack that tells the reduce algorythm to stop reducing when solving the inside of a paren pair. The closing

> paren triggers a reduction as does the end of input, and also removes the open paren mark from the pstack, and restores the 'previous operation' precedence so parsing can continue after the close paren where it left off. This can be done with recursion or without (hint: use a stack to store the previous precedence when encountering a '(' ...). The generalized version of this is to use a parser generator implemented shunting yard algorythm, f.ex. using yacc or bison or taccle (tcl analog of yacc).

Share Improve this answer

Follow

edited Feb 2 at 14:34

desertnaut

60.2k ● 31 ● 151 ● 176

answered Dec 9, 2008 at 20:35

Adam Davis

93.5k ● 60 ● 271 ● 333

2

A Python solution using pyparsing can be found here. Parsing infix notation with various operators with precedence is fairly common, and so pyparsing also includes the `infixNotation` (formerly `operatorPrecedence`) expression builder. With it you can easily define boolean expressions using "AND", "OR", "NOT", for example. Or you can expand your four-function arithmetic to use other operators, such as ! for factorial, or '%' for modulus, or add P and C operators to compute permutations and combinations. You could write

an infix parser for matrix notation, that includes handling of '-1' or 'T' operators (for inversion and transpose). The operatorPrecedence example of a 4-function parser (with '!' thrown in for fun) is here and a more fully featured parser and evaluator is here.

Share Improve this answer Follow

edited Sep 6, 2018 at 20:31

answered Sep 4, 2009 at 8:02

PaulMcG

63.6k ● 16 ● 96 ● 132

1

I know this is a late answer, but I've just written a tiny parser that allows all operators (prefix, postfix and infix-left, infix-right and nonassociative) to have arbitrary precedence.

I'm going to expand this for a language with arbitrary DSL support, but I just wanted to point out that one doesn't need custom parsers for operator precedence, one can use a generalized parser that doesn't need tables at all, and just looks up the precedence of each operator as it appears. People have been mentioning custom Pratt parsers or shunting yard parsers that can accept illegal inputs - this one doesn't need to be customized and (unless there's a bug) won't accept bad input. It isn't complete in a sense, it was written to test the algorithm and its input is in a form that will need some preprocessing, but there are comments that make it clear.

Note some common kinds of operators are missing for instance the sort of operator used for indexing ie table[index] or calling a function function(parameter-expression, ...) I'm going to add those, but think of both as postfix operators where what comes between the delimeters '[' and ']' or '(' and ')' is parsed with a different instance of the expression parser. Sorry to have left that out, but the postfix part is in - adding the rest will probably almost double the size of the code.

Since the parser is just 100 lines of racket code, perhaps I should just paste it here, I hope this isn't longer than stackoverflow allows.

A few details on arbitrary decisions:

If a low precedence postfix operator is competing for the same infix blocks as a low precedence prefix operator the prefix operator wins. This doesn't come up in most languages since most don't have low precedence postfix operators. - for instance: ((data a) (left 1 +) (pre 2 not) (data b)(post 3 !) (left 1 +) (data c)) is a+not b!+c where not is a prefix operator and ! is postfix operator and both have lower precedence than + so they want to group in incompatible ways either as (a+not b!)+c or as a+(not b!+c) in these cases the prefix operator always wins, so the second is the way it parses

Nonassociative infix operators are really there so that you don't have to pretend that operators that return different types than they take make sense together, but without having different expression types for each it's a kludge.

As such, in this algorithm, non-associative operators refuse to associate not just with themselves but with any operator with the same precedence. That's a common case as < <= == >= etc don't associate with each other in most languages.

The question of how different kinds of operators (left, prefix etc) break ties on precedence is one that shouldn't come up, because it doesn't really make sense to give operators of different types the same precedence. This algorithm does something in those cases, but I'm not even bothering to figure out exactly what because such a grammar is a bad idea in the first place.

```
#lang racket
;cool the algorithm fits in 100 lines!
(define MIN-PREC -10000)
;format (pre prec name) (left prec name) (right prec n
(post prec name) (data name) (grouped exp)
;for example "not a*-7+5 < b*b or c >= 4"
;which groups as: not ((((a*(-7))+5) < (b*b)) or (c >=
;is represented as '((pre 0 not)(data a)(left 4 *)(pre
(data 5)(nonassoc 2 <)(data b)(left 4 *)(data b)(right
>=)(data 4))
;higher numbers are higher precedence
;"(a+b)*c" is represented as ((grouped (data a)(left 3
(data c))

(struct prec-parse ([data-stack #:mutable #:auto]
                    [op-stack #:mutable #:auto])
  #:auto-value '())

(define (pop-data stacks)
  (let [(data (car (prec-parse-data-stack stacks)))]
    (set-prec-parse-data-stack! stacks (cdr (prec-pars
    data))

(define (pop-op stacks)
```

```
  (let [(op (car (prec-parse-op-stack stacks)))]
    (set-prec-parse-op-stack! stacks (cdr (prec-parse-
    op))

(define (push-data! stacks data)
    (set-prec-parse-data-stack! stacks (cons data (pre
stacks))))

(define (push-op! stacks op)
    (set-prec-parse-op-stack! stacks (cons op (prec-pa

(define (process-prec min-prec stacks)
  (let [(op-stack (prec-parse-op-stack stacks))]
    (cond ((not (null? op-stack))
           (let [(op (car op-stack))]
             (cond ((>= (cadr op) min-prec)
                    (apply-op op stacks)
                    (set-prec-parse-op-stack! stacks (
                    (process-prec min-prec stacks)))))

(define (process-nonassoc min-prec stacks)
  (let [(op-stack (prec-parse-op-stack stacks))]
    (cond ((not (null? op-stack))
           (let [(op (car op-stack))]
             (cond ((> (cadr op) min-prec)
                    (apply-op op stacks)
                    (set-prec-parse-op-stack! stacks (
                    (process-nonassoc min-prec stacks)
                   ((= (cadr op) min-prec) (error "mul
associative operator"))
                   ))))))

(define (apply-op op stacks)
  (let [(op-type (car op))]
    (cond ((eq? op-type 'post)
           (push-data! stacks `(,op ,(pop-data stacks)
          (else ;assume infix
           (let [(tos (pop-data stacks))]
             (push-data! stacks `(,op ,(pop-data stack

(define (finish input min-prec stacks)
  (process-prec min-prec stacks)
  input
  )
```

```
(define (post input min-prec stacks)
  (if (null? input) (finish input min-prec stacks)
      (let* [(cur (car input))
             (input-type (car cur))]
        (cond ((eq? input-type 'post)
               (cond ((< (cadr cur) min-prec)
                      (finish input min-prec stacks))
                     (else
                      (process-prec (cadr cur)stacks)
                      (push-data! stacks (cons cur (li
                      (post (cdr input) min-prec stack
              (else (let [(handle-infix (lambda (proc-
                                                (cond ((< (c
                                                       (fini
stacks))
                                                      (else
                                                       (proc
stacks)
                                                       (push
                                                       (star
stacks)))))]
                      (cond ((eq? input-type 'left) (h
0))
                            ((eq? input-type 'right) (
1))
                            ((eq? input-type 'nonassoc
nonassoc 0))
                            (else error "post op, infi
expected here"))))))))

;alters the stacks and returns the input
(define (start input min-prec stacks)
  (if (null? input) (error "expression expected")
      (let* [(cur (car input))
             (input-type (car cur))]
        (set! input (cdr input))
        ;pre could clearly work with new stacks, but c
one?
        (cond ((eq? input-type 'pre)
               (let [(new-stack (prec-parse))]
                 (set! input (start input (cadr cur) n
                 (push-data! stacks
                             (cons cur (list (pop-data
```

```
                    ;we might want to assert here that th
null
                    (post input min-prec stacks)))
                ((eq? input-type 'data)
                 (push-data! stacks cur)
                 (post input min-prec stacks))
                ((eq? input-type 'grouped)
                 (let [(new-stack (prec-parse))]
                   (start (cdr cur) MIN-PREC new-stack)
                   (push-data! stacks (pop-data new-stac
                 ;we might want to assert here that the
null
                 (post input min-prec stacks))
                (else (error "bad input"))))))

(define (op-parse input)
  (let [(stacks (prec-parse))]
    (start input MIN-PREC stacks)
    (pop-data stacks)))

(define (main)
  (op-parse (read)))

(main)
```

Share Improve this answer Follow

edited Oct 31, 2014 at 9:55

answered Mar 10, 2014 at 10:17

Josh S

147 ● 1 ● 7

1

Here is a simple case recursive solution written in Java. Note it does not handle negative numbers but you can do add that if you want to:



```
public class ExpressionParser {

public double eval(String exp){
    int bracketCounter = 0;
    int operatorIndex = -1;

    for(int i=0; i<exp.length(); i++){
        char c = exp.charAt(i);
        if(c == '(') bracketCounter++;
        else if(c == ')') bracketCounter--;
        else if((c == '+' || c == '-') &&
bracketCounter == 0){
            operatorIndex = i;
            break;
        }
        else if((c == '*' || c == '/') &&
bracketCounter == 0 && operatorIndex < 0){
            operatorIndex = i;
        }
    }
    if(operatorIndex < 0){
        exp = exp.trim();
        if(exp.charAt(0) == '(' &&
exp.charAt(exp.length()-1) == ')')
            return eval(exp.substring(1,
exp.length()-1));
        else
            return Double.parseDouble(exp);
    }
    else{
        switch(exp.charAt(operatorIndex)){
            case '+':
                return eval(exp.substring(0,
operatorIndex)) +
eval(exp.substring(operatorIndex+1));
            case '-':
                return eval(exp.substring(0,
operatorIndex)) -
```

}

Share Improve this answer answered Oct 13, 2017 at 4:54

Follow

user4617883

1,479 ● 1 ● 13 ● 23

1

Algorithm could be easily encoded in C as recursive descent parser.

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

/*
 *  expression -> sum
 *  sum -> product | product "+" sum
 *  product -> term | term "*" product
 *  term -> number | expression
 *  number -> [0..9]+
 */

typedef struct {
    int value;
    const char* context;
} expression_t;

expression_t expression(int value, const char* context
    return (expression_t) { value, context };
}

/* begin: parsers */

expression_t eval_expression(const char* symbols);

expression_t eval_number(const char* symbols) {
    // number -> [0..9]+
    double number = 0;
    while (isdigit(*symbols)) {
        number = 10 * number + (*symbols - '0');
        symbols++;
    }
    return expression(number, symbols);
}
```

```
expression_t eval_term(const char* symbols) {
    // term -> number | expression
    expression_t number = eval_number(symbols);
    return number.context != symbols ? number : eval_e
}

expression_t eval_product(const char* symbols) {
    // product -> term | term "*" product
    expression_t term = eval_term(symbols);
    if (*term.context != '*')
        return term;

    expression_t product = eval_product(term.context +
    return expression(term.value * product.value, prod
}

expression_t eval_sum(const char* symbols) {
    // sum -> product | product "+" sum
    expression_t product = eval_product(symbols);
    if (*product.context != '+')
        return product;

    expression_t sum = eval_sum(product.context + 1);
    return expression(product.value + sum.value, sum.c
}

expression_t eval_expression(const char* symbols) {
    // expression -> sum
    return eval_sum(symbols);
}

/* end: parsers */

int main() {
    const char* expression = "1+11*5";
    printf("eval(\"%s\") == %d\n", expression,
eval_expression(expression).value);

    return 0;
}
```

next libs might be useful: yupana - strictly arithmetic operations; tinyexpr - arithmetic operations + C math functions + one provided by user; mpc - parser combinators

Explanation

Let's capture sequence of symbols that represent algebraic expression. First one is a number, that is a decimal digit repeated one or more times. We will refer such notation as production rule.

```
number -> [0..9]+
```

Addition operator with its operands is another rule. It is either `number` or any symbols that represents `sum "*" sum` sequence.

```
sum -> number | sum "+" sum
```

Try substitute `number` into `sum "+" sum` that will be `number "+" number` which in turn could be expanded into `[0..9]+ "+" [0..9]+` that finally could be reduced to `1+8` which is correct addition expression.

Other substitutions will also produce correct expression: `sum "+" sum` -> `number "+" sum` -> `number "+" sum "+" sum` -> `number "+" sum "+" number` -> `number "+" number "+" number` -> `12+3+5`

Bit by bit we could resemble set of production rules aka grammar that express all possible algebraic expression.

```
expression -> sum
sum -> difference | difference "+" sum
difference -> product | difference "-" product
product -> fraction | fraction "*" product
fraction -> term | fraction "/" term
term -> "(" expression ")" | number
number -> digit+
```

To control operator precedence alter position of its production rule against others. Look at grammar above and note that production rule for `*` is placed below `+` this will force `product` evaluate before `sum`. Implementation just combines pattern recognition with evaluation and thus closely mirrors production rules.

```
expression_t eval_product(const char* symbols) {
    // product -> term | term "*" product
    expression_t term = eval_term(symbols);
    if (*term.context != '*')
        return term;

    expression_t product = eval_product(term.context +
    return expression(term.value * product.value, prod
}
```

Here we eval `term` first and return it if there is no `*` character after it this is left choise in our production rule otherwise - evaluate symbols after and return `term.value * product.value` this is right choise in our production rule i.e. `term "*" product`

Share Improve this answer Follow

edited Jan 26, 2018 at 7:01

answered Jan 25, 2018 at 15:05

Viktor Shepel

191 ● 1 ● 8

0

Actually there's a way to do this without recursion, which allows you to go through the entire expression once, character by character. This is O(n) for time and space. It takes all of 5 milliseconds to run even for a medium-sized expression.

First, you'd want to do a check to ensure that your parens are balanced. I'm not doing it here for simplicity. Also, I'm acting as if this were a calculator. Calculators do not apply precedence unless you wrap an expression in parens.

I'm using two stacks, one for the operands and another for the operators. I increase the priority of the operation whenever I reach an opening '(' paren and decrease the priority whenever I reach a closing ')' paren. I've even revised the code to add in numbers with decimals. This is in c#.

NOTE: This doesn't work for signed numbers like negative numbers. Probably is just a simple revision.

```
  internal double Compute(string sequence)
    {
        int priority = 0;
        int sequenceCount = sequence.Length;
        for (int i = 0; i < sequenceCount; i++) {
            char s = sequence[i];
            if (Char.IsDigit(s)) {
                double value =
ParseNextNumber(sequence, i);
                numberStack.Push(value);
                i = i + value.ToString().Length -
1;
            } else if (s == '+' || s == '-' || s
== '*' || s == '/')  {
               Operator op =
ParseNextOperator(sequence, i, priority);
                CollapseTop(op, numberStack,
operatorStack);
                operatorStack.Push(op);
            } if (s == '(') { priority++; ;
continue; }
            else if (s == ')') { priority--;
continue; }
        }
        if (priority != 0) { throw new
ApplicationException("Parens not balanced"); }
        CollapseTop(new Operator(' ', 0),
numberStack, operatorStack);
        if (numberStack.Count == 1 &&
operatorStack.Count == 0) {
            return numberStack.Pop();
        }
        return 0;
    }
```

Then to test this out:

```
Calculator c = new Calculator();
double value = c.Compute("89.8+((9*3)+8)+
(9*2)+1");
Console.WriteLine(string.Format("The sum of the
```

```
expression is: {0}", (float)value));
//prints out The sum of the expression is: 143.8
```

Share Improve this answer Follow

edited Oct 17, 2020 at 14:28

answered Oct 17, 2020 at 4:55

Charles Owen
2,792 ● 2 ● 22 ● 29

0

Pure javascript, no dependencies needed

I very like bart's answer.

and I do some modifications to read it easier, and also add support some function(and easily extend)

```
function Parse(str) {
  try {
    return parseExpr(str.replaceAll(" ", "")) // Imple
  } catch (e) {
    alert(e.message)
  }
}

Parse("123.45+3*22*4")
```

It can support as below

```
const testArray = [
  // 👇 Basic Test
  ["(3+5)*4", ""],
  ["123.45+3*22*4", ""],
  ["8%2", ""],
```

```
  ["8%3", ""],
  ["7/3", ""],
  ["2*pi*e", 2 * Math.atan2(0, -1) * Math.exp(1)],
  ["2**3", ""],

  // 👇 unary Test
  ["3+(-5)", ""],
  ["3+(+5)", ""],

  // 👇 Function Test
  ["pow{2,3}*2", 16],
  ["4*sqrt{16}", 16],
  ["round{3.4}", 3],
  ["round{3.5}", 4],
  ["((1+e)*3/round{3.5})%2", ((1 + Math.exp(1)) * 3 /
  ["round{3.5}+pow{2,3}", Math.round(3.5)+Math.pow(2,3
]
```

Full code

```
// 👇 Main
(() => {
  window.onload = () => {
    const nativeConsoleLogFunc = window.console.erro
    window.console.error = (...data) => { // Overrid
test.
      const range = document.createRange()
      const frag = range.createContextualFragment(`<
      document.querySelector("body").append(frag)
      nativeConsoleLogFunc(...data)
    }

    // Add Enter event
    document.querySelector(`input`).onkeyup = (keybo
      if (keyboardEvent.key === "Enter") {
        const result = Parse(document.getElementById
        if (result !== undefined) {
          alert(result)
        }
      }
```

```
    }

    const testArray = [
      // 👇 Basic Test
      ["(3+5)*4", ""],
      ["123.45+3*22*4", ""],
      ["8%2", ""],
      ["8%3", ""],
      ["7/3", ""],
      ["2*pi*e", 2 * Math.atan2(0, -1) * Math.exp(1)
      ["2**3", ""],

      // 👇 unary
      ["3+(-5)", ""],
      ["3+(+5)", ""],

      // 👇 Function Test
      ["pow{2,3}*2", 16],
      ["4*sqrt{16}", 16],
      ["round{3.4}", 3],
      ["round{3.5}", 4],
      ["((1+e)*3/round{3.5})%2", ((1 + Math.exp(1))
2],
      ["round{3.5}+pow{2,3}", Math.round(3.5) + Math

      // 👇 error test
      ["21+", ValueMissingError],
      ["21+*", ParseError],
      ["(1+2", ParseError], // miss ")"
      ["round(3.12)", MissingParaError], // should b
      ["help", UnknownVarError],
    ]

    for (let [testString, expected] of testArray) {
      if (expected === "") {
        expected = eval(testString) // Why don't you
writing the function yourself? Because the browser m
policy considerations. [CSP](https://content-securit
      }
      const actual = Parse(testString, false)

      if (actual !== expected) {
        if (actual instanceof Error && actual instan
          continue
```

```
        }
        console.error(`${testString} = ${actual}, va
</code> expected`)
      }
    }
  }
})()

// 👇 Script
class UnknownVarError extends Error {
}

class ValueMissingError extends Error {
}

class ParseError extends Error {
}

class MissingParaError extends Error {
}

/**
 * @description Operator
 * @param {string} sign "+", "-", "*", "/", ...
 * @param {number} precedence
 * @param {"L"|"R"} assoc associativity  left or rig
 * @param {function} exec
 * */
function Op(sign, precedence, assoc, exec = undefine
  this.sign = sign
  this.precedence = precedence
  this.assoc = assoc
  this.exec = exec
}

const OpArray = [
  new Op("+", 10, "L", (l, r) => l + r),
  new Op("-", 10, "L", (l, r) => l - r),
  new Op("*", 20, "L", (l, r) => l * r),
  new Op("/", 20, "L", (l, r) => l / r),
  new Op("%", 20, "L", (l, r) => l % r),
  new Op("**", 30, "R", (l, r) => Math.pow(l, r))
]
```

```
const VarTable = {
  e: Math.exp(1),
  pi: Math.atan2(0, -1), // https://developer.mozill
US/docs/Web/JavaScript/Reference/Global_Objects/Math
  pow: (x, y) => Math.pow(x, y),
  sqrt: (x) => Math.sqrt(x),
  round: (x) => Math.round(x),
}

/**
 * @param {Op} op
 * @param {Number} value
 * */
function Item(op, value = undefined) {
  this.op = op
  this.value = value
}

class Stack extends Array {
  constructor(...items) {
    super(...items)
    this.push(new Item(new Op("", 0, "L")))
  }

  GetLastItem() {
    return this[this.length - 1] // fast then pop //
https://stackoverflow.com/a/61839489/9935654
  }
}

function Cursor(str, pos) {
  this.str = str
  this.pos = pos
  this.MoveRight = (step = 1) => {
    this.pos += step
  }
  this.PeekRightChar = (step = 1) => {
    return this.str.substring(this.pos, this.pos + s
  }

  /**
   * @return {Op}
   * */
  this.MoveToNextOp = () => {
```

```
    const opArray = OpArray.sort((a, b) => b.precede
    for (const op of opArray) {
      const sign = this.PeekRightChar(op.sign.length
      if (op.sign === sign) {
        this.MoveRight(op.sign.length)
        return op
      }
    }
    return null
  }
}

/**
 * @param {Cursor} cursor
 * */
function parseVal(cursor) {
  let startOffset = cursor.pos

  const regex = /^(?<OpOrVar>[^\d.])?(?<Num>[\d.]*)/
  const m = regex.exec(cursor.str.substr(startOffset
  if (m) {
    const {groups: {OpOrVar, Num}} = m
    if (OpOrVar === undefined && Num) {
      cursor.pos = startOffset + Num.length

      if (cursor.pos > startOffset) {
        return parseFloat(cursor.str.substring(start
cursor.pos - startOffset)) // do not use string.subs
in the future. https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Deprecated_and_obso
      }
    }

    if ("+-(".indexOf(OpOrVar) !== -1) {
      cursor.pos++
      switch (OpOrVar) {
        case "+": // unary plus, for example: (+5)
          return parseVal(cursor)
        case "-":
          return -(parseVal(cursor))
        case "(":
          const value = parseExpr(cursor)
          if (cursor.PeekRightChar() === ")") {
            cursor.MoveRight()
```

```
            return value
          }
          throw new ParseError("Parsing error: ')' e
        }
      }
    }


    // 👇 below is for Variable or Function
    const match = cursor.str.substring(cursor.pos).mat
https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Stri

    if (match) {
      // 👇 Variable
      const varName = match[0]
      cursor.MoveRight(varName.length)
      const bracket = cursor.PeekRightChar(1)
      if (bracket !== "{") {
        if (varName in VarTable) {
          const val = VarTable[varName]
          if (typeof val === "function") {
            throw new MissingParaError(`${varName} is
curly brackets`)
          }
          return val
        }
      }

      // 👇 is function
      const regex = /{(?<Para>[^{]*)}/gm
      const m = regex.exec(cursor.str.substring(cursor
      if (m && m.groups.Para !== undefined) {
        const paraString = m.groups.Para
        const para = paraString.split(',')
        cursor.MoveRight(paraString.length + 2) // 2 =
        return VarTable[varName](...para)
      }
      throw new UnknownVarError(`unknown variable ${va
    }

    // 👇 Handle Error
    if (cursor.str.length === cursor.pos) { // example
      throw new ValueMissingError(`Parsing error at en
```

```
expected.`)
  } else { // example: 1+2+*
    throw new ParseError("Parsing error: unrecognize
  }
}

/**
 * @param {string|Cursor} expr
 * */
function parseExpr(expr) {
  const stack = new Stack()
  const cursor = (expr instanceof Cursor) ? expr : n
  while (1) {
    let rightValue = parseVal(cursor)
    const op = cursor.MoveToNextOp() ?? new Op("", 0

    while (
      op.precedence < stack.GetLastItem().op.precede
      (op.precedence === stack.GetLastItem().op.prec
'L')) {
      const lastItem = stack.pop()
      if (!lastItem.op.exec) { // end reached
        return rightValue
      }
      rightValue = lastItem.op.exec(lastItem.value,
    }

    stack.push(new Item(op, rightValue))
  }
}

function Parse(str, alertError = true) {
  try {
    return parseExpr(str.replaceAll(" ", ""))
  } catch (e) {
    if (alertError) {
      alert(e.message)
      return undefined
    }
    return e
  }
}
```

```
<input type="text" id="expr" name="expr" placeholder
<button onclick="const x = Parse(document.getElement
!= null) alert(x);">
  Calculate!
</button>
```

Run code snippet Expand snippet

Share Improve this answer Follow

edited Jul 31, 2021 at 16:07

answered Jul 29, 2021 at 14:31

Carson

7,810 ● 2 ● 56 ● 54