**The Shunting-Yard Algorithm**

December 11, 2011 · [Coding](https://www.reedbeta.com/blog/category/coding/) · [6 Comments](https://www.reedbeta.com/blog/the-shunting-yard-algorithm/#comments)

In game development (or programming in general) it’s not uncommon to have a situation where you’d like to let a user enter an arithmetic formula that your code parses and evaluates. For example, in a shader you might like to have an annotation that specifies how a parameter is to be computed in the main application. In various kinds of authoring tools you might like to create a shape, image, or animation based on a mathematical function. Embedding a full-fledged scripting language like Python or Lua is a bit overkill for these kinds of tasks. So how can we handle arithmetic expressions without a large amount of infrastructure?

In textbooks and university computer-science courses, we often hear about a few classic approaches to parsing formulas. One is the so-called [reverse Polish notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation), where we write formulas in postfix form, with operators following their operands:

2 3 + *// means 2 + 3*

a b + c d + \* *// means (a + b) \* (c + d)*

pi 4 / sin *// means sin(pi/4)*

The nice things about RPN are that (a) parentheses are not needed, nor the concepts of operator precedence and associativity, since the order of operations is fully specified by the notation; and (b) it can be parsed by a simple algorithm: scan the formula left-to-right, when you see an operand push it on a stack, and when you see an operator, pop the required operand(s) off the stack, apply the operator, and push the result back on. When you’re done, the result is the only item left on the stack (assuming well-formed input).

That’s quite easy to add to an application; there’s almost no infrastructure needed. But it has the disadvantage that it requires users to work in this unfamiliar and awkward notation. Of course, users can be trained to work in RPN, and with experience it no longer appears unfamiliar or awkward. But let’s take pity on the poor users and let them work with standard mathematical notation. What options are there?

The standard computer-science cirriculum at this point would start talking about context-free grammars, abstract syntax trees, and syntax-directed parsers. There are two main approaches to building parsers that are used in practice, i.e. for parsing programming languages: top-down (also known as recursive descent or LL) and bottom-up (aka shift-reduce, LR). Unfortunately, neither of these is a good fit for embedding a simple arithmetic language in an application. Top-down parsing isn’t a good fit for arithmetic in general, since each level of operator precedence requires its own nonterminal symbol in the grammar (each of which corresponds to a function call in the parser), and right-associative operators can’t be expressed in the grammar without breaking the LL constraint, necessitating some sort of extragrammatical fixup. Bottom-up parsing works by using a large state machine whose transition rules are usually impractical to work out by hand, requiring a parser generator tool such as [Bison](http://www.gnu.org/s/bison/) to compute. Again, that’s a lot of infrastructure to throw at what is not such a complicated problem.

Fortunately, there is another way: the [shunting-yard algorithm](http://en.wikipedia.org/wiki/Shunting-yard_algorithm). It is due to Edsger Dijkstra, and so named because it supposedly resembles the way [trains are assembled and disassembled](http://www.youtube.com/watch?v=vop7UTPhdoc) in a railyard. This algorithm processes infix notation efficiently, supports precedence and associativity well, and can be easily hand-coded.

[**How It Works**](https://www.reedbeta.com/blog/the-shunting-yard-algorithm/#how-it-works)

As in RPN, we scan the formula from left to right, processing each operand and operator in order. However, we now have two stacks: one for operands and another for operators. Then, we proceed as follows:

* If we see an operand, push it on the operand stack.
* If we see an operator:
  + While there’s an operator on top of the operator stack of precedence *higher than or equal to* that of the operator we’re currently processing, pop it off and apply it. (That is, pop the required operand(s) off the stack, apply the operator to them, and push the result back on the operand stack.)
  + Then, push the current operator on the operator stack.
* When we get to the end of the formula, apply any operators remaining on the stack, from the top down. Then the result is the only item left on the operand stack (assuming well-formed input).

Note that “applying” an operator can mean a couple of different things in this context. You could actually execute the operators, in which case the operands would be numerical values of all the terms and subexpressions; you could also build a syntax tree, in which case the operands would be subtrees. The algorithm works the same way in either case.

That’s basically all there is to it, aside from some bells and whistles! As you can see, it has a lot in common with the RPN algorithm, and is just a little more complicated.

[**Advanced Usage**](https://www.reedbeta.com/blog/the-shunting-yard-algorithm/#advanced-usage)

I described the algorithm above in its simplest form, but there are several enhancements that can be made to handle more complicated formulas.

**Associativity.** Above, I said that when processing an operator, any operators of *equal* precedence at the top of the stack should be popped and applied. This makes those operators left-associative, since the leftmost of the two operators will be applied first. You can implement right-associativity by leaving equal-precedence operators on the stack.

**Parentheses.** Parens are a bit of a special case. When you see a left paren, push it on the operator stack; no other operators can pop a paren (so it’s as if it has the *lowest* precedence). Then when you see a right paren, pop-and-apply any operators on the stack until you get back to a left paren, which is popped and discarded.

**Unary operators.** These generally work just like any binary operators except that they only pop one operand when they’re applied. There is one extra rule that needs to be followed, though: when processing a unary operator, it’s only allowed to pop-and-apply other unary operators—never any binary ones, regardless of precedence. This rule is to ensure that formulas like a ^ -b are handled correctly, where ^ (exponentiation) has a higher precedence than - (negation). (In a ^ -b there’s only one correct parse, but in -a^b you want to apply the ^ first.)

Both prefix and postfix unary operators can be used. The way to tell whether you’re in a position to allow prefix or postfix operators is to look at the previous token; if it’s an operand, you’re looking for binary and postfix unary operators, and if the previous token is an operator (or there’s no previous token) you’re looking for prefix unary operators. Note that a left parent counts as an operator and a right paren as an operand for this purpose. This rule also allows you to tell whether - is a negation (unary) or a subtraction (binary)—it’s a negation if it appears when looking for a prefix unary operator, and a subtraction otherwise.

**Function calls.** The prefix/postfix rule also allows you to tell when an open paren designates a function call rather than grouping a subexpression (a grouping paren is like a prefix operator while a function-call one is like a postfix operator). When a function-call paren is encountered, the operand at the top of the stack is the function to be called. Push the paren on the operator stack as before, but also set up a list somewhere to hold the function arguments, and maintain a mapping that lets you find that list again from the paren on the stack. (Note that with nested function calls, you could have multiple left parens on the stack.)

Then, when a comma is encountered, pop-and-apply operators back to a left paren; the operand on the top of the stack is then the next argument, and should be popped and added to the argument list. When the right paren is encountered, do the same, then pop and discard the left paren. (Note that the arguments shouldn’t be left on the operand stack, at least not without some sort of sentinel between them; this would allow an ill-formed call like f(a, b, +) to be parsed as f(a + b)).

Array subscripts using square brackets can be handled in the same way as function calls.

After all this, the algorithm has grown a bit, and is a little trickier to get right in all the corner cases—but it’s still pretty simple, and *certainly* less work than a full-blown syntax-directed parser! In my opinion, it’s a shame that the shunting-yard algorithm isn’t more widely discussed as part of standard texts and computer-science university courses. Even the [Dragon Book](http://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools) doesn’t so much as mention it! I never heard of this algorithm until I happened to see a forum post that referenced it, but its simplicity, elegance, and efficiency make it a superior solution for many use-cases of processing arithmetic expressions.