# Parsing Expressions with the Shunting-Yard Algorithm

Fritz Ruehr • Willamette CS 353 • *[draft 2017]*

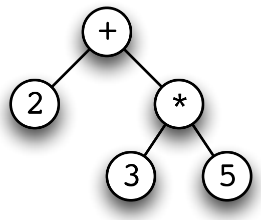
**Overview:** this is a description of a simple algorithm to parse expression trees from string input. It can handle simple operator-based syntax as in our compiler language for CS 353, but not a lot more. However, the basic approach it uses, and some of the terminology, are similar to more sophisticated algorithms used for real-world parser generators. So, studying these techniques is both useful for our labs and also foreshadows more complex approaches you might study later. An even simpler approach can be used to convert infix to post-fix forms, if that is all you need, or to evaluate the expressions as you go, if you don't need the trees. (We need the trees because we have dynamic constructs like READ that are meant to evaluate repeatedly when we "run" the expression, not just once.)

**Introduction:** Say that we wish to parse arithmetic or other algebraic expressions with infix binary operators and atomic terms. For example, consider this grammar:

```
<exp> ::=  <var> | <lit> | <exp> <opr> <exp> | ( <exp> )

<var> ::= [a-z]+

<lit> ::= [0-9]+

<opr> ::= * | +
```

By "parsing" we mean an algorithm that transforms strings into trees, thereby recognizing an implicit sub-expression structure. For example:

```
" 2 + 3 * 5 "   ➤
```



One idea here is to use precedence rules for the operators to disambiguate concrete syntax like the following, where two different parses are possible:

```
2 + 3 * 5       =?    (2 + 3) * 5     ✗
                =?    2 + (3 * 5)     ✔
```

Here we usually prefer the second interpretation, given the "PEMDAS" rules for precedence of arithmetic operators, where multiplication (*) is "before" addition (+).

> **Note:** We say "before", but this has nothing to do with *order of evaluation,* which is the order in which the sub-expressions of a binary operator term are evaluated. This is just an issue of how the expression (or tree) itself is built. Here, "before" just means "lower in the tree".

Of course, we also want to recognize overt uses of parentheses, in order to allow users to override the precedence rules—but honoring the precedence rules is one of the tricky parts of the algorithm.

**Basic strategy:** as with just about all parsing algorithms, we will use a stack (actually two) to help in the conversion of the linear string into a tree. This follows the aphorism:

> "A stitch in time saves nine … but a *stack over time* is a *tree.*" ™

The idea is that the stack holds items temporarily that we might like to combine into a (sub-)tree while we look at what follows, since we might need to consolidate items further to the right in the input first, then incorporate the material from the left "above" that to the right in the final tree.

The specific algorithm we will describe is called the *shunting-yard algorithm:* it was devised by Edsgar Dijkstra (one of the pioneers of programming languages and software more generally); the name comes from its connection to how trains are maneuvered around a train yard by "shunting" groups of cars off onto a side-track before re-incorporating them into the full train. (The shunting/side-track is stack-like in its behavior.)

The overall process of the algorithm involves looking at a new token of input (perhaps we call a tokenizer to peel off several characters at a time to get this token) and deciding, based on this new token and its relationship to the tops of the stacks, whether to (a) push the token onto a stack (sometimes called a *shift)* or (b) combine some stack items together into a new sub-tree (sometimes called a *reduce).* The process repeats over and over for each new token, ultimately leaving (we hope) just a single, final result tree on one of the stacks: if we don't have exactly one tree when we're done, then something went wrong with the syntax, and we should generate an error message. (Typically, either there were too many arguments and not enough operators, or the other way 'round, or things came in all together the wrong order.)

Thus the basic question becomes: given a new token, plus the stack tops, should we *shift* or should we *reduce?* In fact, in some situations we will make multiple successive reductions (i.e., several reduce steps at once). So the overall algorithm looks like this:

```
while (more tokens) {
    get next token;
    compare token to stack tops and either:
        shift token onto stack;
     OR repeatedly reduce stack(s) until certain conditions;
}

finally: reduce remaining stack(s);

if (one tree left)
    return that tree;
else
    error (depending on what was left over);
```

The main dependence on which we base our shift/reduce decisions is on the *precedence* of the incoming token if it's an operator or similar (if it's an argument, i.e., an atomic term like a literal number or variable, we shift it).

> **Note:** the original algorithm is not necessarily intended to handle things like the pre- and post-increment operators we have in our CS 353 language, or conditional expressions or a few others. But we can extend the basic ideas to handle these cases.

**The details:** as mentioned above, we will use two stacks: one of these will be for *arguments* (either atomic terms or sub-trees that have already been reduced or consolidated) and one for *operators* (things like (+) and (`*`), but also left parentheses, and perhaps a few more things if we extend the algorithm). In terms of actual implementation, you want the argument stack to be able to hold trees or expressions, so if you use a Java generic stack, you want to instantiate it to the overall `Expression` superclass (presuming you have sub-classes for things like literal numbers, atomic variables, binary operator applications, etc.). The operator stack could be made to just hold characters, although tokens would be better (for example, in an extended version you might need to push something like a (++) or (—) operator on). You could also use a Java *enumerated type* for your token type.

As mentioned above, you might want to call a simple lexer or tokenizer to strip off spaces, gather up the characters of the next token, and perhaps convert them into some kind of overt `Token` objects or enumerated values. (You could also just leave them as strings, but you'll probably want to convert at least the numeric literals to integers.)

How do we use the two stacks, the “argument stack” and the “operator stack”? When we get a new token (call it the “current one”), we take one of the following actions, based on what kind of token it is:

- If it’s an *atomic entity* like a numeric constant (but eventually also variables), we push it on the argument stack.

- If it’s an *operator,* we would *like* to push it on the operator stack—but we have some things to check first:
    - if the operator stack is empty, or has a left parenthesis on top, just push it;
    - if there is an operator on the op-stack with *lower precedence* than the current one, again just push the current one;
    - if there is already an operator on the op-stack with *higher precedence* than the current one, we should reduce the stack first, consolidating that operator and its arguments, and then continue to check again (i.e., compare the current operator to the one on the new stack top).
    - [What about equal precedence? Depending on whether the current operator is left- or right-associative, we either reduce or push, resp.]

- If the current token is a *left parenthesis,* we should push it on the stack—it will wait there for the corresponding right one to come along;
  but if it is a *right parenthesis,* we should reduce the stack repeatedly until we find the matching left parenthesis, popping it off before we continue. (At this point, the top of the argument stack should be the tree corresponding to the parenthesized sub-expression.)

- Finally, if we reach the end of the expression, we should repeatedly reduce the stacks to leave exactly one expression tree on the argument stack: this is the final result. (Of course, it may happen that there are two or more things left on the argument stack and none on the op-stack, or that there is only one argument but still some operators left on their stack: these correspond to two different error conditions, either too many arguments or too many operators.)

**Note**: most on-line descriptions of the algorithm differ from this one in two respects: (1) they generally address *conversion of infix to postfix,* rather than the production of an actual expression tree: so they just emit literal constants to an output buffer (or “print” them) rather than keeping them in a stack and reducing them to a final tree. (2) they generally seem not to address an important issue for parsers (even toy ones) which is *fully validating* the form of the input. More specifically, the algorithm as stated doesn’t enforce the obvious alternation between arguments and operators. This can be effected with a simple boolean “toggle”, i.e., a boolean variable or switch which is toggled (or inverted) every time we see the category of item (*argument* or *operator)* we are currently seeking. (Left parentheses should only be found when seeking an argument, and right when seeking an operator—neither toggles.) *[Disclaimer: I have not proved that this is a sufficient and correct approach, but I suspect that it is.]*