Stop Mowing The Lawn!

A Better Way to Understand Expressions

by Steven T Abell
Software Designer
© 2022 brising.com

A different version of this article appeared in *The ComMuniCator*, a publication of the California Mathematics Council, Volume 47 Number 1, September 2022. The content here is similar, subject to different editorial policies. Thanks to CMC for publication in their journal.
Click here to download the PowerPoint slides:
http://www.brising.com/OperatorPrecedenceParser.pptx

One of the cornerstones of math education is the Lawnmower Algorithm. The Lawnmower has been around for a long time. You learned it. You teach it. Some of your students get it. Some think they get it. Some give up trying to get it.

It's great when they get it. But there are a lot of failures: some outright egregious failures, with more near misses that don't show themselves until later. And yet no one can afford to fail at this. If you can't Mow The Lawn correctly and reliably, you can't read or write math beyond a very elementary level.

If you don't know what all this talk about lawnmowers is about, you'll recognize it in just a moment. Here's how it goes:

You have let your lawn grow wild for a long time, and now you need to mow it before the Homeowners Association sues you. But the grass is very tall. If you simply push your lawnmower into this mess, it will choke and stall. Instead of doing that, fiddle with the mower's controls to set the cutting height up as far as it will go. Now push the mower over the entire lawn, cutting off just the tops of the grass. Then lower your mower a notch and do it again, making the grass a little shorter this time. Do this again and again until you finally get down to ground level. Now you are finished.

In your classroom, you may know the Lawnmower Algorithm by a different name, or use a different analogy (or no analogy at all) to explain it. But by this name or any other, this is how we teach our students to read expressions. This is how you learned it when you were in school. It is how your grandparents learned it. It is probably how their grandparents learned it, too.

The Lawnmower Algorithm is augmented by the acronym PEMDAS (**P**arentheses, **E**xponents, **M**ultiplication, **D**ivision, **A**ddition, **S**ubtraction), or by the infamous mnemonic "**P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally", where each letter represents a different level of Mowing The Lawn. Except of course that isn't really true. It is really more like PE{MD}{AS}. Except that isn't really true either. How many of your students really know how PEMDAS works? How many of their parents or siblings or future co-workers do? That number is smaller than we'd like.

Most of my working life was deep in the trenches of Silicon Valley as a software developer. My first *real* programming language was FORTRAN, learned in Math 29 at UC Davis in 1975. A few years later I taught Math 19, which was BASIC. In either of these languages, and in many others, here is the entire lecture in which expressions are explained: "*It's just like what you learned back in Algebra I.*" And if you allow for a few slightly different symbols, it is. Except that it isn't.

Most programming languages are taught this way. But *no programming language works this way*. And yes, *it matters.*

Because I have been around so long, I usually end up being the unofficial mentor in a development team. Other engineers come to me with obscure questions about their code. Several times, I have had to explain to people what a programming language compiler does when it reads an expression. The usual response is: "*No, no, Steve, you don't understand: that's not how it works. It's like we learned in Algebra class. You know… the Lawnmower Algorithm*" or whatever name they were given for the process. Then I have to explain to them: "*No, no,* ***you*** *don't understand: that's not how it works.*" Maybe their teachers didn't know the difference. Or maybe they did know but didn't think it mattered. So here we are with a real-world problem, with these engineers, mostly college-degreed software professionals, who don't know what's happening or how to fix it.

What's really shocking is that I know some of these people had a class in compiler construction at some point. They should have known better. And yet it is the Lawnmower Algorithm they learned forever ago that is stuck in their heads.

The actual behavior of actual computers actually matters to an increasingly large portion of the population, and not just professional programmers. So we have to root this thing out.

Let's start by looking at something familiar, intentionally pointing out the obvious. Then we'll look at other things that aren't so obvious.

Think about this expression and how to evaluate it:

```
a*b+c*d+e*f
```

Using the Lawnmower Algorithm:

1) Multiply **`a*b`**.
2) Multiply **`c*d`**.
3) Multiply **`e*f`**.
4) Add results from **Steps 1** and **2**.
5) Add results from **Steps 4** and **3**.

A compiler does it differently:

1) Multiply **`a*b`**.
2) Multiply **`c*d`**.
3) Add results from **Steps 1** and **2**.
4) Multiply **`e*f`**.
5) Add results from **Steps 3** and **4**.

To show this more simply, I have numbered the operators in their order of execution, first as a lawnmower does it, then as a compiler does it.

```
a*b+c*d+e*f
 1 4 2 5 3      lawnmower
 1 3 2 5 4      compiler
```

Here is an example using parentheses:

```
a*b*(c*d-(e+f))
 4 5  2 3  1   lawnmower
 1 5  2 4  3   compiler
```

Troublesome examples are easy to find. It is harder to find examples that don't diverge like this.

What about evaluation? Are the results of these two pathways through the operators the same or different? If you are doing algebra, the results are *always the same*. (Feeling relieved? Wait a bit.) If you are working with computers, the results *might be different*. (Are you nervous yet? Good.)

Why are the results different *sometimes*? Because algebra is *stateless*. But real-world computer software, while entirely deterministic, can also be *state-dependent*. To show one example, I will introduce an idea that, while simple, might still be new to you.

A *stream* is a programming language artifact used in expressions. Streams can be written using traditional function syntax, although most languages write them differently. A stream's distinctive feature is that it delivers the *next* element in a sequence on demand. If a stream **s** contains the sequence **(2, 4, 6, 8)** and **n** is the function-like request for the next element, successive invocations of **n(s)** deliver those numbers in that order, one number for each invocation of **n(s)**. The expression **n(s) + n(s) + n(s)** is equivalent to **2 + 4 + 6** for that particular stream **s**, but would have a different result for streams containing other sequences. In this example, the number **8** still waits in **s**, unused.

Streams appear prominently in many modern computer programs. Their use often simplifies code significantly. But they also have *state*, which is silently introduced into your expression. A stream waiting to deliver its first element is in a *different state* from when it is waiting to deliver its second element, etc.

Now consider an example where the stream **s** contains the sequence **(8, 4, 2, 1)**. Evaluate this:

```
n(s)*(n(s)-n(s)/n(s))
```

What did you get? Using the Lawnmower Algorithm, many educated people begin inside the parentheses to obtain this:

```
1*(8-4/2) = 6
```

Keeping precedence in mind, some insist it must be:

```
1*(2-8/4) = 0
```

But if you ask a compiler, the only entity whose opinion matters in programming, the result is:

```
8*(4-2/1) = 16
```

If your programming language also allows you to define operators (as you can in C++ or Swift, for example), you can easily write code that is yet more mysterious to lawnmowers. And even if you are careful to write stateless code, a simple everyday trip through your debugger can be a mind-bending experience if you have the wrong expectations.

So knowing only how a compiler understands expressions will not hurt you when doing algebra. But knowing only how to Mow The Lawn *can* hurt you when programming, and it will be a complete surprise to you when it does. If you only want to teach or learn one way, which is the wiser choice?

Why don't programming languages work like you learned back in Algebra I? Could they be made to do so?

Yes, as a matter of fact, they could. But they aren't made that way, and they won't be, because *that would be too hard*.

If you know how to program, even a little bit, here is a challenge for you: take what you know about evaluating expressions *as you teach your students how to do it* and code it up. Remember: you don't get to use other knowledge or personal discoveries about how this could be done. It has to be what you teach your students in class. Be sure your program handles everything in the PEMDAS stack. Is your exponentiation operator left-associative or right-associative? As extra credit, make it handle the unary minus operator.

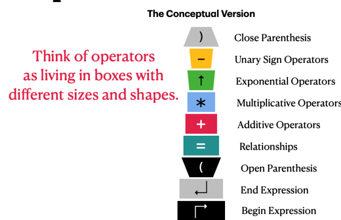
If you are determined and careful, you may eventually develop code that works for all cases. But even if you do, you will hate what you have created. That is some really ugly code there. If you want to reply here that I haven't yet seen this code you haven't yet written, I will tell you: that doesn't matter. I know the problem. I know the code. It is ugly. And here is something every good software developer knows:

Code that is ugly is code that is wrong. Period… end of story… doesn't matter if it works or not. *It's wrong!* Ugly code is not only about subjective aesthetics. Ugly code is telling you something important about your understanding of the problem. Be sure to listen.

There are other ways of reading expressions that are objectively simpler than Mowing The Lawn. That is why compilers use them. So… why are we teaching this hard Lawnmower stuff to our students?

If the word *compiler* makes you squeamish, relax. I could write a short article telling you how to make a compiler, and you would even think it's kind of fun. But, better than that, I made an illustrated PowerPoint deck you can get by clicking here:

http://www.brising.com/OperatorPrecedenceParser.pptx

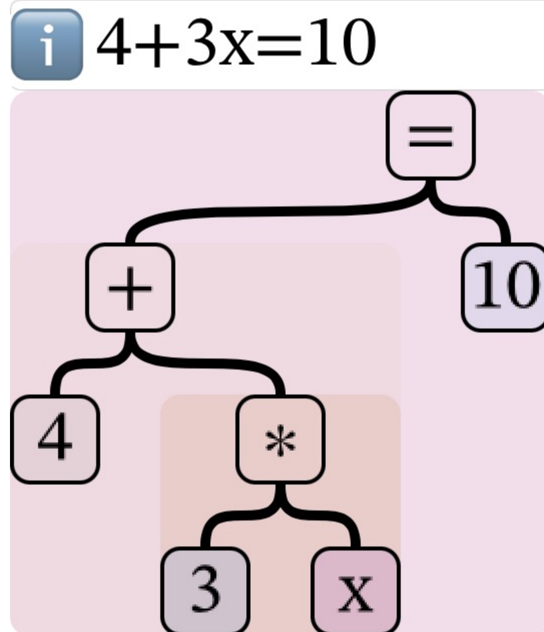


There is no code in these slides. A few simple diagrams make the process even more clear than my short article would. Feel free to use this deck in your classroom. Contact me if you want to talk about it.

The kind of compiler shown in the slides is called an Operator Precedence Parser. It is stunningly simple. Unlike the Lawnmower, which has to read and modify an expression multiple times, the Operator Precedence Parser only has to read it once.

If words are good and pictures are better, what about interactive animations?

If you or your students want to have a dynamic video game experience with expressions, then look at ***DragginMath***™. This iPad/iPhone app is on the App Store now. Expression parsing is just the beginning of what it can help you teach.



Now that I have told you about this, do I expect you to throw the Lawnmower in the trash and make it go away by tomorrow? Of course not. That thing has been with us for a long time. The culture of math education and curriculum development doesn't often turn so quickly, and the result can be regrettable when it tries. But it will be better if the Lawnmower goes away eventually. So I want to put this idea into your head now and see what good things you can do with it. Then it will be true when some near-future class of programmers is told "*It's just like what you learned back in Algebra I,*" and they won't struggle later with elementary problems they didn't know they could have. ***Better yet:*** **everyone *will have a simpler way to read, understand, and evaluate expressions.***