Towards representing mathematical expressions in plain text

Patrolin

July 20, 2022

Abstract

Standard math notation often relies on spacing and/or 2D layout for legibility, this translates poorly to 1D (often monospace or nearly monospace) text used in programming and text files. Moreover the standard math notation is hard to explain in a few sentences and leads to confusion about operator precedence when you start adding many (domain-specific) operators (`&&` *,* `||` *,* `<<` *,* `>>` *,* `&` *,* `|` *,* `^` *,* `@` *).*

I. Standard notation (SN)

Brackets are a good fallback, they let you express any expression you want although very verbosely.

$$(a+b)\cdot c$$
$$a+(b\cdot c)$$
$$a+(((b\cdot c)\cdot d)\cdot e)$$

You probably don't want to write brackets all day, so SN has many alternative representations.

E(MD)(AS) says that exponentiation a^b takes precedence over multiplication $a\cdot b$ and division $\frac{a}{b}$, both of which have the same precedence, and all of them take precedence over addition $a+b$ and subtraction $a-b$ (both of which have the same precedence).

Therefore:

$$a+b\cdot c = a+(b\cdot c)$$
$$(a+b)\cdot c = (a+b)\cdot c$$

You may notice how inconsistent the presentation of these operators is, to write these in plain text we have to use someting like:
`a ** b` for a^b, $a*b$ for $a\cdot b$, a/b for $\frac{a}{b}$

You can also choose to write $a\cdot b$ as ab (or really $a\ b$ so we know where one name ends and the other begins), however this now has higher precedence than a/b, but still lower precedence than `a ** b`.

Not to mention that exponentiation is right-associative, meaning $a^{b^c} = a^{(b^c)}$, unlike the rest of the operators, since the left-associative version can be represented differently $(a^b)^c = a^{bc}$, however in practice nobody uses the right-associative version because

a) it is unintuitive
b) nobody wants to or has to deal with numbers of this magnitude anyway.

You may also encounter functions like $f(a,b) = a\cdot b + b^2$, then you can evaluate the function at some point $f(3,5) = 3\cdot 5 + 5^2 = 40$

If a function only has one variable you can write it without brackets (usually this is only done with sin and cos):

$$\sin x = \sin(x)$$

This prefix operator then requires its own precedence; below ab, but above $a\cdot b$:

$$\sin xe^x = \sin(x\cdot(e \texttt{ ** } x))$$
$$\sin x\cdot e^x = \sin(x)\cdot(e \texttt{ ** } x)$$

This means that SN actually uses EIP(MD)(AS):

1. Exponentiation
2. Implicit multiplication
3. Prefix operators
4. Multiplication/Division
5. Addition/Subtraction

II. Quick notation (QN)

Parsing long expressions can be complicated (especially for humans), so why not just make left-to-right the default.

a) every operator has the same precedence
b) every operator is left-associative

Therefore:

$$\begin{aligned} a + b * c &= (a + b) * c \\ a + (b * c) &= a + (b * c) \end{aligned}$$

This makes adding new operators trivial and hopefully reduces the brackets needed per expression, like with polyfilling the modulo operator in Javascript:

$$a \text{ mod } b = a \mathbin{\%} b + b \mathbin{\%} b = ((a \mathbin{\%} b) + b) \mathbin{\%} b$$

A consequence of this system is that you may have to write more brackets for operators with higher precedence in SN, however this really only applies to addition/subtraction followed by multiplication/division:

$$\begin{aligned} a + bx &= a + (b * x) \\ a - \tfrac{b}{x} &= a - (b/x) \\ a * b^x &= a * (b \texttt{ ** } x) \end{aligned}$$

You could also include implicit multiplication $a\ b$, but this doesn't actually save you any characters over $a * b$.

III. Results

Lerp SN	`lerp(t, a, b) = (1 - t) * a + t * b`
Lerp QN	`lerp(t, a, b) = 1 - t * a + (t * b)`
Modulo SN	`a mod b = ((a rem b) + b) rem b`
Modulo QN	`a mod b = a rem b + b rem b`
Modulo2 SN	`a mod b = (a rem b) + (b * (a < 0))`
Modulo2 QN	`a mod b = (a rem b) + (b * (a < 0))`
Remainder SN	`a rem b = (a mod b) - (b * (a < 0))`
Remainder QN	`a rem b = a mod b - (b * (a < 0))`
Quadratic formula SN	`(-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)`
Quadratic formula QN	`(-b + (b ** 2 - (4 * a * c)) ** 0.5) / (2 * a)`
Modular exponentiation SN	`x[n] = (x[n-1] * (b[i] ? x[n-1] : x[0])) mod p`
Modular exponentiation QN	`x[n] = x[n-1] * (b[i] ? x[n-1] : x[0]) mod p`
Xorshift excerpt SN	`(x << k) | (x >> (64 - k))`
Xorshift excerpt QN	`x << k | (x >> (64 - k))`
Mersenne twister excerpt SN	`y | ((y >> u) & d)`
Mersenne twister excerpt QN	`y | (y >> u & d)`

Notice how "Mersenne twister excerpt SN" has brackets for `(y >> u) & d`, this may or may not be required depending on the programming language used, as everyone makes up their own precedence rules.

In conclusion QN provides an easy to understand, easy to extend, perhaps shorter notation for mathematical expressions.