Implicit Multiplication Problem On Calculators

halof

October 2025

You probably have seen an image or meme showcasing how calculators often make some miscalculations when dealing with parenthesis and implicit multiplication.

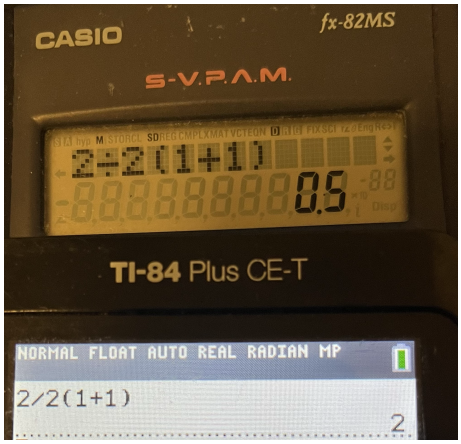

Figure 1: Difference between calculators dealing with implicit multiplication

In fact, this error not only occurs but it is widely known. Let's dive deep into why this happens.
First, it is important to distinguish three main categories of calculators:

1. Basic calculators.
2. Scientific calculators.
3. Graphic/advanced calculators.

Some calculators do not parse the full expression and therefore cannot evaluate operator precedence the same way a full parser or a human would.

The following image represents how a human would interpret the expression `2/2(1+1)` step by step:

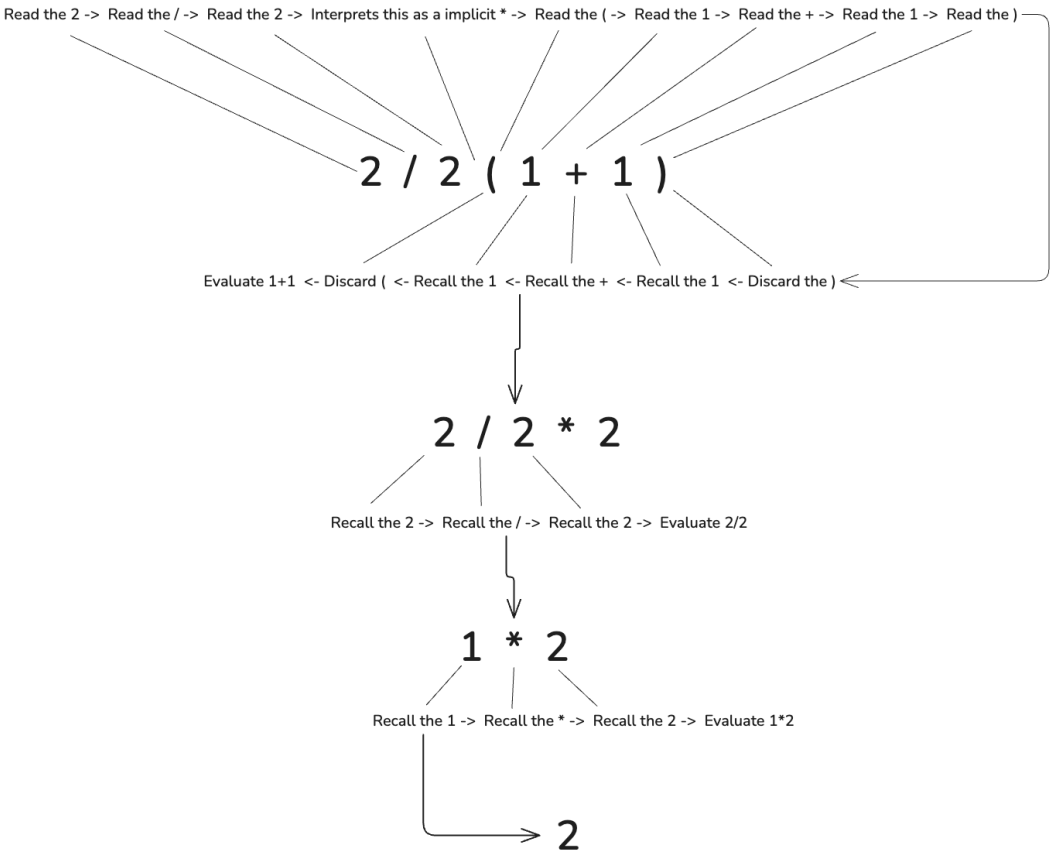

Figure 2: Human infix interpretation

As you can see, making a simple calculation requires a few steps and is not always straightforward. There are some ways a calculator can interpret a calculation:

1. Infix (standard algebraic) evaluation.
2. Reverse Polish Notation (postfix, RPN).
3. Compilation or internal interpretation (lexer + parser + evaluator).

Some calculators internally convert infix notation to postfix (Reverse Polish Notation) to evaluate operator precedence. However, this process depends entirely on how the expression is tokenized.

For example, the expression `2/2(1+1)` is ambiguous. Two valid parses are:

- `(2/2)*(1+1)` → postfix: `2 2 / 1 1 + *` → evaluates to 2.
- `2/(2*(1+1))` → postfix: `2 2 1 1 + * /` → evaluates to 0.5.

To keep this document short, the full postfix translation steps are omitted; a code implementation is available at `github.com/halofmayor/calculator`.

The following image represents how a human would interpret the postfix expression `2 2 / 1 1 + *` step by step:

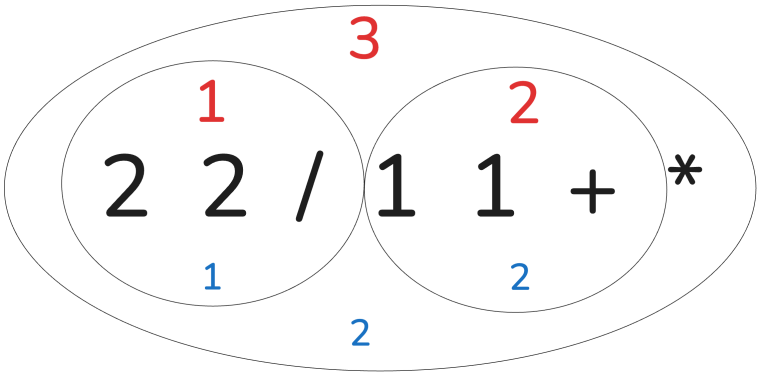

Figure 3: Human postfix interpretation (red = step, blue = result of the step)

But if the machine can interpret this correctly, why does the error still happen?

Historically, early calculators had limited memory and simpler input models: they expected explicit operators (for example `2/2*(1+1)`) rather than implicit multiplication (`2/2(1+1)`). When converting from infix to postfix, a parser that does not detect implicit multiplication will treat tokens incorrectly. As a result it may build the wrong operator sequence — effectively interpreting the expression as `2/(2*(1+1))` (result 0.5) instead of `(2/2)*(1+1)` (result 2).

Nowadays it is trivial to handle implicit multiplication by inserting an explicit `*` during tokenization (e.g. between a number or `)` and a following `(` or number). However, for compatibility and legacy behavior, some calculators, including certain scientific models, still follow older parsing rules that ignore this case.