Abstract Syntax Tree in Java

Talk about the example about evaluating Math Expressions, it builds the understanding of lexer, parser, and visitor's role.

Talk about tail-end recursion: it optimizes the stack frame of the recursion function. It will reduce the number of stack frame generated by your function, removing the limitation on StackOverFlow and will be bounded by the amount of memory in your machine.

It was particularly confusion for the implementation of the math expression AST because parseExpression, parseTerm, and parseFactor uses while loop for Tail end recursion but it wasn't clear to me how it was optimizing the stack frame. The below explanation helps clear it up.

> You might be thinking: "*Wait, `parseExpression` still calls `parseTerm`, and `parseTerm` calls `parseFactor`. Isn't that recursion?*"
>
> Yes, t still uses the stack. However:
>
> 1. The **depth** of that recursion is limited by the **precedence levels** of your math (e.g., Expression -> Term -> Factor). That’s only 3-4 frames deep.
> 2. The **length** of the expression (how many `+` signs you have) is handled by the `while` loop.
>
> Without those `while` loops, a long math equation would crash your program. With them, your parser can handle an expression a million characters long without breaking a sweat.

Revision #2
Created 2026-04-24 02:44:38 UTC by Tamarine
Updated 2026-04-24 02:48:55 UTC by Tamarine