Computer Algebra System with Graphical Interface

Jordan Jacobson

November 7, 2025

1 Overview

This document outlines the complete pipeline of the light computer algebra system (CAS). The narrative follows a user expression from the initial keystroke through parsing, evaluation, and LaTeX rendering, highlighting the responsibilities of the principal classes and the contracts at each boundary.

2 User Interaction Layer

1. The application boots via `Main`, which instantiates the Swing shell containing the input field, evaluation trigger, history list, and LaTeX preview pane. Event listeners dispatch expressions when the user presses Enter or clicks Evaluate.

2. The UI performs sanitation by trimming whitespace, normalizing glyphs (e.g., $\times$, $\div$, $-$), and validating characters and structure before invoking the parser.

3. A session controller co-located with `Main` constructs a `Parser` using the sanitized string and packages metadata (request id, precision settings, evaluation mode) into a `ComputationRequest`. Work is forwarded to the evaluator on a background thread to keep the UI responsive.

3 Lexing and Parsing

1. `Parser` handles lexing with a cursor over the source string, emitting `Token`⟨`Types`, Object⟩ instances for numbers, identifiers, delimiters, and operators such as $+ - * / \% ! \sqrt{} = < > \leq \geq$.

2. Tokens carry their enumerated type, payload (`BigDecimal`, `Symbol`, canonical operator char), and span offsets for diagnostics.

3. Parsing uses a Pratt-style precedence climber supporting literals, parentheses, unary operators, precedence-tiered binary operators, and postfix constructs. The output is a typed abstract syntax tree (AST) of lightweight nodes (`BinaryNode`, `CallNode`, etc.).

4 Symbol and Numeric Core

1. `Symbol` represents identifiers and constants, guaranteeing canonical ordering and hashing so structurally identical subtrees deduplicate.

2. `Number` forms the scalar backbone with tagged representations for arbitrary-precision integer, rational, real (`BigDecimal`), and complex values. Each variant enforces immutability and normalized form.

3. `Vector` and `Matrix` extend arithmetic to collections while preserving dimension metadata and deferring expensive operations until required.

4. Utility structures (`LinkedList`, `Pair`, `TokenStream`) support hot paths where allocations and ordering semantics matter.

5 Evaluation Pipeline

1. The evaluator consumes the AST and a `Context` built from user preferences. Traversal proceeds depth-first with normalization and simplification passes.

2. Normalization aligns operand domains (e.g., upgrading integers to rationals when mixed) and consults the symbol table for function definitions and derivative rules.

3. Simplification applies prioritized rule sets covering arithmetic identities, structural canonicalization, and function-specific reductions. Ordering (canonicalization, constant folding, structural contraction, function rewrites, numeric evaluation) prevents oscillation.

4. Numeric evaluation runs after symbolic simplification when requested, relying on `Number`'s arbitrary-precision routines such as exponentiation by squaring, Newton iteration, and series approximations. Exact forms are preserved whenever possible.

5. Diagnostics propagate through structured exceptions detailing node context and recovery hints; warnings (loss of significance, truncation) attach to the `ComputationResult`.

6 Rendering and Output

1. The simplified AST passes to a renderer facade. Using `TeXFormula/TeXIcon`, the renderer emits LaTeX fragments with correct parentheses, fractions, radicals, and piecewise layouts.

2. `ImageDisplay` paints the icon onto a buffered image, refreshes the preview window, and exposes the rendered asset for export (e.g., `formula.png`).

3. The UI updates the output panel, history list, and status indicators (runtime, simplification depth, warnings) from the Swing event thread once evaluation completes.

7 Control Flow Summary

1. `Main` captures input, sanitizes it, and issues a computation request.

2. `Parser` tokenizes the expression and builds the AST using `Token`, `Types`, and `Symbol`.

3. The evaluator normalizes domains, simplifies the tree, and performs arbitrary-precision arithmetic via `Number`, `Vector`, and related classes.

4. The renderer converts the result to LaTeX, which `ImageDisplay` presents; warnings and errors return to the UI for display.

5. The cycle repeats for each user submission, reusing cached symbols, tokens, and numeric contexts.