Calculator Parser: Design Document

Overview

This system parses and evaluates mathematical expressions, transforming a string like "2 + 3 * (4 - 1)" into a numeric result. The key architectural challenge is designing a parser that correctly interprets operator precedence (multiplication before addition) and associativity (left-to-right versus right-to-left), requiring a structured approach to dissect and evaluate expressions.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

1. Context and Problem Statement

> **Milestone(s):** All Milestones (1-3) — This foundational section explains the core parsing challenge that all subsequent milestones build upon.

At its heart, this project is about teaching a computer to understand and follow the rules of arithmetic that you learned in grade school. Consider the expression `"2 + 3 * 4"`. A human immediately knows to multiply 3 and 4 first, then add 2, arriving at 14. However, a computer receives this as a flat string of characters: `'2'`, `' '`, `'+'`, `' '`, `'3'`, `' '`, `'*'`, `' '`, `'4'`. There is no inherent structure or hierarchy; it's just a linear sequence. The fundamental challenge—and the core educational goal of this project—is designing a system that can reconstruct the intended hierarchical meaning from this flat sequence, correctly applying the rules of operator precedence and associativity.

Mental Model: The Order of Operations Puzzle

Think of evaluating a mathematical expression as a chef following a complex recipe with nested instructions. The recipe might read: *"Take 2 cups of flour, then add the result of mixing 3 eggs with 4 cups of sugar."* A chef doesn't just combine ingredients left-to-right; they recognize that the instruction "mixing 3 eggs with 4 cups of sugar" forms a sub-task that must be completed before its result can be used in the main recipe. The phrase "the result of mixing..." acts like parentheses, creating a nested operation.

Similarly, consider the expression `2 + 3 * 4`. The multiplication symbol (`*`) has a higher "priority" than the addition (`+`), much like the mixing sub-task in the recipe has priority over the final addition. A naive approach that processes the string strictly left-to-right would perform `2 + 3 = 5`, then `5 * 4 = 20`, yielding an incorrect result. The system must instead recognize that `3 * 4` is a cohesive sub-expression that must be evaluated first, forming an intermediate result (12) that then participates in the addition.

This mental model extends to all the rules you'll implement:

- **Precedence:** Some operators (like multiplication) are "more binding" than others (like addition). In our recipe, mixing (multiplication) happens before combining with other ingredients (addition).
- **Associativity:** When operators have the same precedence, like in `8 / 4 / 2`, the order matters. Is it `(8 / 4) / 2 = 1`, or `8 / (4 / 2) = 4`? Most operators are left-associative (group left-to-right), but exponentiation (`^`) is typically right-associative (`2 ^ 3 ^ 2` is `2 ^ (3 ^ 2)`).
- **Grouping:** Parentheses `( )` explicitly create a sub-expression, overriding the default precedence rules, just as the phrase "the result of..." in the recipe creates an explicit sub-task.

The key insight is that **the linear text is a serialized encoding of a tree-shaped thought process**. The parser's job is to deserialize it—to reconstruct the tree from the linear form.

Core Problem: From String to Structure

The core technical problem is **syntax analysis**: converting a sequence of characters into a hierarchical representation that encodes the intended order of operations. This representation is called an **Abstract Syntax Tree (AST)**.

Let's trace the transformation for the expression `2 + 3 * 4`:

1. **Input String (Linear):** `"2 + 3 * 4"`
2. **Lexical Analysis (Tokenization):** The string is broken into meaningful chunks called *tokens*.

Token Type	Lexeme (Text)
`NUMBER`	`2`
`PLUS`	`+`
`NUMBER`	`3`
`ASTERISK`	`*`
`NUMBER`	`4`

3. **Syntactic Analysis (Parsing):** The flat token sequence is transformed into a tree structure based on grammar rules. The AST for `2 + 3 * 4` must encode that `*` has higher precedence than `+`.

```
    +
   / \
  2   *
     / \
    3   4
```

`This tree is the *abstract syntax* because it abstracts away the concrete punctuation (spaces, parentheses) and focuses solely on t`

4. **Evaluation:** The tree is traversed depth-first. The evaluator sees the `*` node first, computes `3 * 4 = 12`, then replaces that subtree with the value `12`. Next, it evaluates the `+` node: `2 + 12 = 14`.

The challenge intensifies with nested structures. For `2 * (3 + 4)`:

- The parentheses are tokens (`LPAREN`, `RPAREN`) that tell the parser to treat `3 + 4` as a primary unit.
- The resulting AST correctly shows multiplication applied to the *result* of the addition:

```
    *
   / \
  2   +
     / \
    3   4
```

The parser must therefore solve a **grammar ambiguity**. The same token sequence could, in theory, be structured in multiple ways. The parser uses a set of **grammar rules** (defining precedence and associativity) to choose the single, correct tree structure. Defining these rules and implementing an algorithm to apply them is the central puzzle of this project.

Existing Parsing Strategies

There are several well-established algorithms for parsing expression grammars. For a beginner-friendly calculator, two primary strategies are most relevant: **Recursive Descent Parsing** and **Pratt Parsing** (also called Operator-Precedence or Top-Down Operator Precedence parsing).

Decision: Parser Algorithm Selection

- **Context:** We need to choose a parsing algorithm that is implementable by beginners, clearly demonstrates precedence and associativity concepts, and can handle the required operators (binary, unary, parentheses, function calls).
- **Options Considered:**
  1. **Shunting Yard Algorithm:** Converts infix notation to postfix (RPN) using a stack, then evaluates the RPN. Very efficient but abstracts away the tree-building step, making it less educational for understanding ASTs.
  2. **Recursive Descent Parsing:** A top-down method where each grammar rule becomes a function. Closely mirrors the grammar, making it intuitive to read and write. Explicitly shows precedence via the call chain.
  3. **Pratt Parsing:** A more generalized recursive descent that uses binding power tables. Extremely elegant and compact, especially for handling mixfix operators, but its conceptual model is slightly more abstract for beginners.
- **Decision:** Use **Recursive Descent Parsing** for the core implementation guidance in this document.
- **Rationale:**
  1. **Conceptual Transparency:** The call stack directly mirrors the precedence hierarchy. A function `parse_expression` calls `parse_term` (for `+` / `-`), which calls `parse_factor` (for `*` / `/`), which calls `parse_primary` (for numbers/parentheses). This directly teaches that "term is higher precedence than expression."
  2. **Incremental Development:** It's easy to extend. Adding a new precedence level (like exponentiation in Milestone 2) involves adding one new function and adjusting the call chain.
  3. **Excellent Learning Resource:** The recommended "Crafting Interpreters" book uses this method, providing a perfect complementary resource.
- **Consequences:**
  - **Pro:** The code is verbose but self-documenting; the structure of the parser directly reflects the grammar. Easy to debug by tracing function calls.
  - **Con:** Can be more code to write than Pratt parsing, and the manual precedence climbing via function calls can seem repetitive. However, this repetition reinforces the learning objective.

The following table compares these strategies to help you understand the landscape and why recursive descent is the recommended starting point.

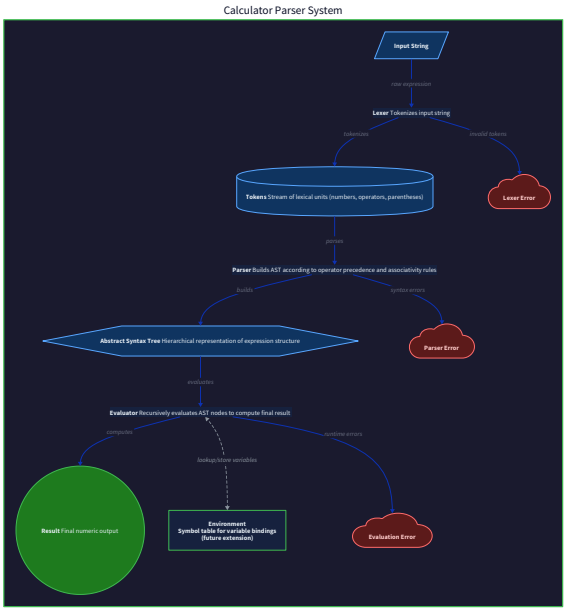
Parsing Strategy	How It Works (Mental Model)	Pros (Why Choose It)	Cons (Trade-offs)	Best For This Project?
Recursive Descent	A committee of experts: each function (`parse_expression`, `parse_term`) is an expert for one precedence level. The lowest-precedence expert (`parse_expression`) delegates sub-tasks to higher-precedence experts (`parse_term`), who may delegate further.	**Intuitive:** Code reads like the grammar rules. **Educational:** Explicitly shows precedence hierarchy in function calls. **Debuggable:** Function call stack shows parsing progress. **Incremental:** Easy to add new operator levels.	**Verbose:** Requires a function for each precedence level. **Rigid:** Grammar is hard-coded into the function structure, making it less flexible for dynamic changes.	✅ **Recommended.** Maximizes learning value by making precedence explicit. The verbosity is a feature for understanding.
Pratt Parsing	A single dispatcher with a rulebook: One parsing function consults a table that defines each operator's "binding power" (precedence) and whether it's prefix (like unary `-`) or infix (like `+`). It uses these rules to decide how to group tokens.	**Concise:** One core parsing loop handles all precedence levels. **Flexible:** Easy to add new operators by updating the table. **Elegant:** Beautiful separation of grammar definition (table) and parsing logic.	**Abstract:** The binding power model is less immediately intuitive than recursive functions. **Debugging:** The flow of control within the single loop can be harder to trace.	🟡 **Alternate Path.** An excellent second implementation after mastering recursive descent. The linked "Pratt Parsing" article is a superb resource.
Shunting Yard	A sorting yard for tokens: Operators are placed onto a stack and shuffled to the output queue based on their precedence, mimicking how a railroad shunting yard organizes train cars.	**Efficient:** Single pass, stack-based. Classic algorithm. **Practical:** Directly produces Reverse Polish Notation (RPN), which is trivial to evaluate.	**Opaque:** The AST construction (if done) is indirect. The algorithm doesn't naturally build a visible tree structure, which is a key learning goal. **Less Educational:** Abstracts away the tree-building and recursion concepts.	🔴 **Not Recommended for Learning.** While efficient, it skips over the core educational objective of constructing and evaluating an AST.

> The critical insight is that all these strategies are solving the same problem: **correctly grouping tokens according to precedence and associativity.** Choosing Recursive Descent prioritizes clarity of concept over code compactness, which is the right trade-off for a learning project.

With the problem defined and the high-level strategy chosen, the subsequent sections will dive into the detailed design of each component—the lexer that creates tokens, the parser that builds the AST, and the evaluator that computes the result—following the pipeline shown in the System Component Diagram:

System Component Diagram

Shows the three main components (Lexer, Parser, Evaluator) and their data flow: Input String → Tokens → AST → Number. Also shows the Environment (symbol table) interacting with the Evaluator.



.

2. Goals and Non-Goals

> **Milestone(s):** All Milestones (1-3) — This section defines the complete feature scope for the entire calculator project, providing clear boundaries for what will and won't be implemented across all milestones.

Think of building this calculator as constructing a **specialized kitchen appliance** rather than a full professional kitchen. The kitchen appliance has specific, well-defined functions: it can bake, broil, and toast, but it cannot sous-vide, ferment, or dehydrate. Similarly, our calculator will handle arithmetic, variables, and basic functions but won't attempt to be a general-purpose programming language. Defining these boundaries upfront prevents "feature creep" where the project grows endlessly and helps learners focus on mastering core parsing concepts without being overwhelmed by complexity.

Goals (Must-Have Features)

The calculator must transform mathematical expressions written as strings into computed numeric results, correctly interpreting the intended mathematical meaning. This requires supporting the operators, precedence rules, grouping mechanisms, and storage capabilities detailed below. The features are organized into three sequential milestones, each building on the previous one.

Milestone	Feature Category	Specific Features	Precedence Level (Highest to Lowest)	Associativity	Example Input	Expected Result
Milestone 1	Basic Arithmetic	Integer and floating-point numbers (including negative values via unary minus in expression)	N/A (operands)	N/A	`-3.14`	`-3.14`
		Addition (`+`)	Level 1 (lowest)	Left	`2 + 3 + 4`	`9`
		Subtraction (`-`)	Level 1	Left	`10 - 3 - 2`	`5`
		Multiplication (`*`)	Level 2	Left	`2 * 3 * 4`	`24`
		Division (`/`)	Level 2	Left	`12 / 3 / 2`	`2`
		Parentheses `( )`	Overrides all precedence	N/A	`(2 + 3) * 4`	`20`
Milestone 2	Unary Operators	Unary minus (negation)	Level 3	Right (prefix)	`-5` , `--5`	`-5` , `5`
	Exponentiation	Power (`^`)	Level 4	Right	`2 ^ 3 ^ 2`	`512` (not 64)
Milestone 3	Variables	Assignment (`=`)	Level 0 (lowest, forms statements)	Right	`x = 5`	Stores `5` , returns `5`
		Variable reference	N/A (identifier)	N/A	`x + 2` after above	`7`
	Built-in Functions	`sin(x)` , `cos(x)` , `sqrt(x)` , `abs(x)`	Level 5 (highest, via function call)	N/A	`sqrt(16)`	`4`
	Expression Integration	Variables and functions in expressions	Respects existing precedence	N/A	`sqrt(x * 2)` with `x=8`	`4`

> **Design Insight:** The precedence hierarchy is crucial. Imagine operators as workers in a factory assembly line: high-precedence operators (like `^`) work first on tightly-bound operands, then mid-level operators (`*` , `/`), then low-level operators (`+` , `-`). Parentheses act as management directives that reorder any worker's task immediately.

Detailed Feature Specifications:

1. **Numeric Literals:** The calculator must parse both integer (`42` , `-7`) and floating-point (`3.14` , `-0.5` , `.5` , `2.`) representations. Internally, these will be stored as a floating-point type (like Python's `float`) to handle all cases uniformly, though integer results should display without decimal points when possible for cleaner output.
2. **Binary Arithmetic Operators:** Each operator follows standard mathematical behavior:
   - Addition and subtraction at precedence level 1, evaluated left-to-right: `10 - 3 - 2` equals `((10 - 3) - 2) = 5` .
   - Multiplication and division at precedence level 2, evaluated left-to-right: `12 / 3 / 2` equals `((12 / 3) / 2) = 2` .
   - Division by zero must be detected and reported as a runtime error, not cause a program crash.
3. **Parentheses:** Parentheses `(` and `)` create explicit grouping that overrides the default precedence hierarchy. Nested parentheses must be supported to arbitrary depth (limited only by system stack in recursive implementations). Mismatched parentheses (more opens than closes or vice versa) must be detected as syntax errors.
4. **Unary Minus Operator:** The unary minus (negation) has higher precedence than multiplication/division but lower than exponentiation. It binds to the immediately following expression: `-2^2` means `-(2^2) = -4` . Multiple unary minuses can chain: `--5` means `-(-5) = 5` . The lexer will produce the same `MINUS` token for both unary and binary minus; the parser distinguishes them based on context.
5. **Exponentiation Operator:** The power operator `^` has the highest precedence among arithmetic operators and is **right-associative**: `2^3^2` must evaluate as `2^(3^2) = 2^9 = 512` , not `(2^3)^2 = 64` . This matches conventional mathematical notation.
6. **Variable Assignment and Storage:** The assignment operator `=` has the lowest precedence and forms an **assignment expression** that both stores a value and returns it. It is right-associative, allowing chained assignments: `x = y = 5` stores `5` in both `x` and `y` . The left operand must be an identifier (variable name). Variables persist in an **environment** (symbol table) for the duration of the calculator session.
7. **Variable References:** When an identifier appears in an expression (not as the left side of assignment), its current value is retrieved from the environment. References to undefined variables must produce a clear semantic error message naming the missing variable.
8. **Built-in Mathematical Functions:** The calculator provides four standard functions:
   - `sin(x)` : Returns sine of `x` (radians)
   - `cos(x)` : Returns cosine of `x` (radians)
   - `sqrt(x)` : Returns square root of `x` (error for negative inputs)

- `abs(x)` : Returns absolute value of `x` Function calls have the highest precedence (like parentheses) and support exactly one argument. The argument can be any expression.

9. **Expression Integration:** All features compose naturally: variables can appear in function arguments (`sqrt(x)`), functions can appear in expressions (`2 * sin(0.5)`), assignments can contain complex expressions (`y = (2 + 3) * -sqrt(16)`).

Architecture Decision Record: Feature Sequencing Across Milestones

- **Context:** Learners need gradual complexity introduction while ensuring each milestone delivers a working, testable system.
- **Options Considered:**
  1. **Vertical slices per milestone:** Each milestone implements all components (lexer, parser, evaluator) for a subset of features. This creates working calculators at each step but requires rewriting components.
  2. **Horizontal layers per milestone:** Milestone 1 builds complete lexer, Milestone 2 builds complete parser, Milestone 3 builds complete evaluator. This teaches components in isolation but delays end-to-end functionality.
  3. **Incremental feature addition:** Each milestone adds features to all components, building a progressively more capable calculator (chosen approach).
- **Decision:** Incremental feature addition across all components.
- **Rationale:** This approach maintains motivation by providing a working calculator at each milestone while allowing natural extension of each component's capabilities. Learners see how new features integrate into the existing architecture rather than building isolated subsystems.
- **Consequences:** Each milestone requires updating multiple components, but the changes are localized to specific feature additions. The final system emerges organically without major redesigns.

Option	Pros	Cons	Chosen?
Vertical slices per milestone	Working end-to-end system at each stage; clear separation of feature sets	Component code rewritten multiple times; doesn't teach component evolution	✗
Horizontal layers per milestone	Deep focus on one component at a time; clean separation of concerns	No complete system until final milestone; demotivating for some learners	✗
Incremental feature addition	Progressive complexity; working system at each stage; natural component evolution	Features can interact in complex ways; requires careful planning of extensions	✓

Non-Goals (Explicitly Out of Scope)

This calculator is **not a general-purpose programming language**. It lacks features that would transform it from a calculation tool into a Turing-complete language. These exclusions are deliberate to maintain focus on parsing and expression evaluation fundamentals.

Category	Specific Exclusions	Why Excluded	Example That Won't Work
Control Flow	Loops (`for` , `while`), conditionals (`if` , `else`), branching	Turns calculator into programming language; requires statement sequencing and execution control	`if x > 0 then x else -x`
User-Defined Functions	Function definitions, parameters, return statements	Requires symbol table scoping, call stack management, and more complex parsing	`f(x) = x * 2`
Complex Data Types	Strings, booleans, arrays, dictionaries, custom objects	Arithmetic calculator focuses on numeric computation only	`"hello" + "world"`
Advanced Operators	Modulo (`%`), bitwise operators (`&` , `|`), comparison chains (`x > y > z`)	Keep operator set minimal for learning precedence fundamentals	`17 % 5`
Multiple Statements	Semicolon separation, line breaks as statement separators	Single expression evaluation per input; no sequencing needed	`x=5; y=10; x+y`
Implicit Multiplication	Juxtaposition-based multiplication (`2π` , `3(4+5)`)	Adds lexical ambiguity; explicit `*` operator teaches precedence clearly	`2(3+4)`
Advanced Function Features	Multiple arguments, variable arguments, named arguments	Single-argument functions simplify parsing and evaluation	`pow(2, 3)`
Error Recovery	Continuing after errors, suggesting fixes, partial evaluation	Beginner project focuses on correct parsing, not robust IDE features	After `2 +* 3` , continue parsing
Persistence	Saving variables between sessions, loading scripts from files	Environment exists only for current evaluation session	Loading `vars.txt`
Alternative Number Bases	Hexadecimal, binary, octal literals	Focus on decimal arithmetic parsing	`0xFF + 1`
Constants	Predefined constants (`π` , `e`)	Can be approximated with variables; not core to parsing	`2 * π`

> **Design Insight:** The "non-goals" act as **guardrails** for the project scope. Each exclusion represents a potential rabbit hole that could consume weeks of development time while teaching little about the core parsing concepts. By explicitly stating these boundaries, learners can confidently say "that's out of scope" when tempted to add features.

Rationale Behind Key Exclusions:

1. **No Control Flow:** Adding conditionals or loops would require implementing a **statement/expression distinction**, **execution control flow**, and potentially a **virtual machine** or **interpreter loop**. These are fascinating topics but belong in a compiler/interpreter project, not a focused calculator parser.
2. **No User-Defined Functions:** Supporting `def f(x): return x*2` would necessitate:
   - Function definition parsing separate from expressions
   - Parameter scoping and local variable environments
   - Call stack management for recursion
   - Return value semantics This effectively creates a programming language, far beyond the "expression evaluator" goal.
3. **No Multiple Statements:** The calculator evaluates one expression at a time. This simplifies the interface and avoids needing to handle statement sequencing, which would require modifying the pipeline to accept multiple expressions and return multiple results.
4. **No Implicit Multiplication:** While `2(3+4)` is mathematically conventional, parsing it requires lookahead or backtracking to distinguish between function calls `f(3)` and implicit multiplication `2(3)` . Requiring explicit `*` eliminates this ambiguity and teaches operators explicitly.
5. **No Error Recovery:** Professional compilers attempt to continue parsing after errors to report multiple issues. This requires sophisticated error recovery strategies (panic mode, phrase-level recovery) that are beyond beginner scope. Our calculator will stop at the first error with a clear message.

⚠️ **Common Pitfall: Scope Creep During Implementation**

- **Symptom:** The implementation grows to include comparison operators, logical operators, or string concatenation because "they seem easy to add."
- **Why It's Problematic:** Each new feature interacts with existing precedence rules, associativity, and type system. Adding comparisons (`>` , `<`) requires boolean types and truth values, which then begs for logical operators (`&&` , `||`), and suddenly you're implementing a full predicate language.
- **How to Avoid:** Refer to this Non-Goals table when tempted to add features. If a feature isn't listed in Goals, don't implement it. Complete the three milestones as specified, then create a separate "extension" project if desired.

The bounded scope ensures learners can complete a functional, well-designed calculator parser within a reasonable timeframe while mastering the core concepts of lexical analysis, recursive descent parsing, abstract syntax trees, and tree-walking evaluation.

---

3. High-Level Architecture

> **Milestone(s):** All Milestones (1-3) — This foundational section describes the system's overall structure, which remains consistent across all three milestones. The architecture decomposes the complex problem of evaluating expressions into three manageable stages: breaking text into tokens, building a structured tree, and walking that tree to compute results.

Architecture Pipeline: Lex, Parse, Evaluate

Mental Model: The Three-Stage Factory Assembly Line Imagine you're tasked with assembling a complex piece of furniture from a flat-pack instruction sheet. You can't just start bolting parts together randomly. Instead, you follow a systematic pipeline:

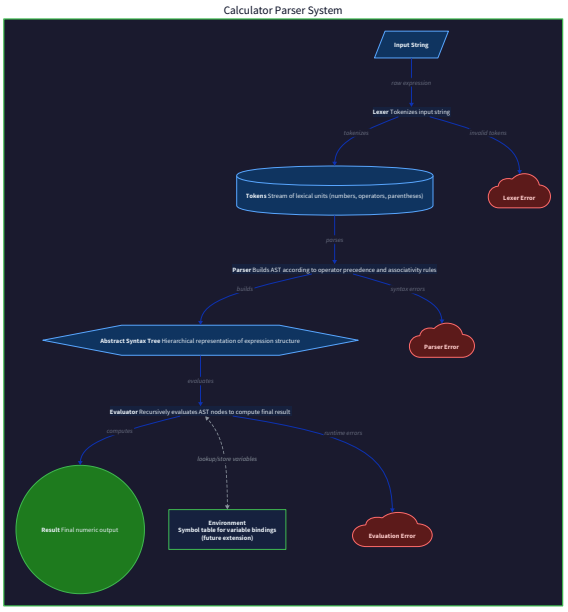
1. **Inventory Checker (Lexer):** First, you meticulously go through the instruction sheet, cataloging every distinct part (screw, plank, bracket) and tool (wrench, hammer) mentioned. You ignore the empty space and comments. Your output is a numbered list of recognized items.
2. **Blueprint Builder (Parser):** Next, you take that list of parts and interpret the assembly instructions. The order matters! "Attach bracket A to plank B using screw C" implies a hierarchy: the bracket and plank are joined *by* the screw. You translate the flat list into a structured assembly diagram showing what connects to what and in what order.
3. **Assembly Worker (Evaluator):** Finally, you follow the completed blueprint step-by-step. For each join in the diagram, you physically connect the corresponding parts. The final output is the assembled piece of furniture.

This is exactly how our calculator works. The input string (`"2 + 3 * 4"`) is the flat instruction sheet. The **Lexer** produces the list of tokens (parts). The **Parser** builds the Abstract Syntax Tree (AST), which is the hierarchical blueprint. The **Evaluator** walks this tree to perform the actual arithmetic and produce the final number.

The data flows in a single direction through this pipeline, with each component transforming the output of the previous one. This separation of concerns makes the system easier to understand, test, and modify.

System Component Diagram

Shows the three main components (Lexer, Parser, Evaluator) and their data flow: Input String → Tokens → AST → Number. Also shows the Environment (symbol table) interacting with the Evaluator.



The diagram above illustrates this linear data flow and the key data structures exchanged. The **Environment** (symbol table) is a shared data store that persists across evaluations, allowing variables to retain their values. It is read and written by the Evaluator but is conceptually separate from the main pipeline.

Core Components and Their Responsibilities

Component	Primary Responsibility	Input	Output	Key Data Structure
Lexer (Tokenizer)	Converts a raw string of characters into a sequence of meaningful tokens.	`str` (e.g., `"x = 2 + 3"`)	`list[Token]` (e.g., `[IDENTIFIER("x"), EQUAL("="), NUMBER("2"), PLUS("+"), NUMBER("3")]`)	`Token` (type, lexeme, position)
Parser	Consumes the token stream and builds a hierarchical tree structure that encodes operator precedence and grouping.	`list[Token]`	`ASTNode` (root of the tree)	Hierarchy of `Number`, `BinaryOp`, `UnaryOp`, etc.
Evaluator	Recursively traverses the AST, executing the operations described by each node to compute a final numeric value.	`ASTNode`, `Environment`	`float` (the final result)	`Environment` (dictionary of variable names to values)
Environment	Acts as a symbol table, providing a mutable mapping from variable names (`str`) to their current numeric values (`float`).	Read/Write requests from Evaluator	Current value for a given name	`dict[str, float]`

End-to-End Data Flow (Concrete Example) Let's trace the journey of a simple expression, `"2 + 3 * 4"`, through the system.

1. **Lexing:** The Lexer scans the string left-to-right.
   - Sees `'2'`, recognizes it as the start of a number, reads the whole integer `"2"`, emits a `Token(type=NUMBER, lexeme="2", position=0)`.
   - Skips the space.
   - Sees `'+'`, emits `Token(type=PLUS, lexeme="+", position=3)`.
   - Skips space.
   - Sees `'3'`, emits `Token(type=NUMBER, lexeme="3", position=5)`.
   - Skips space.
   - Sees `'*'`, emits `Token(type=STAR, lexeme="*", position=7)`.
   - Skips space.
   - Sees `'4'`, emits `Token(type=NUMBER, lexeme="4", position=9)`.
   - Reaches end of string. Output: A list of 5 tokens.
2. **Parsing:** The Parser receives this token list. Using recursive descent (detailed in Section 6), it builds an AST.
   - It recognizes multiplication (`*`) has higher **precedence** than addition (`+`).
   - It therefore groups `3 * 4` as a `BinaryOp` node *before* combining it with the `2`.
   - The resulting AST is:

```
       BinaryOp('+',
           left=Number(2.0),
           right=BinaryOp('*',
               left=Number(3.0),
               right=Number(4.0)
           )
       )
```

`*   This tree structure makes the intended order of operations explicit and unambiguous.`

3. **Evaluation:** The Evaluator receives the root `BinaryOp('+', ...)` node and an empty `Environment`.
   - To evaluate a `BinaryOp`, it must first evaluate its left and right child nodes.
   - It evaluates the left child: `Number(2.0)` simply returns `2.0`.
   - It evaluates the right child: This is another `BinaryOp('*', ...)`. It recursively evaluates *its* children (`3.0` and `4.0`) and then applies the multiplication operator, returning `12.0`.
   - Now it has both operands (`2.0` and `12.0`). It applies the addition operator, producing the final result: `14.0`.

This three-stage design is remarkably robust. Adding a new operator (like `^` for power in Milestone 2) primarily involves extending the Lexer to recognize the new token and updating the Parser's precedence tables and grammar rules. The Evaluator simply gets a new case to handle. Adding variables and functions (Milestone 3) introduces new AST node types (`Variable`, `Assign`, `FunctionCall`) and modifies the data flow to include the `Environment`, but the core pipeline remains unchanged.

> **Design Insight:** The AST is the heart of the system. It serves as an **intermediate representation (IR)** that completely decouples the syntax of the input language from the semantics of its execution. This allows you to change how you evaluate expressions (e.g., interpret vs. compile) or even generate code for a different machine, all without touching the parser.

Recommended File/Module Structure

A well-organized codebase is crucial for maintainability and clarity, especially for learners. We recommend a Python project structure that mirrors the architectural components, promoting **separation of concerns** and making it easy to locate and test each part.

Project Layout

```
calculator_project/          # Project root directory
├── calculator/              # Main package
│   ├── __init__.py          # Makes 'calculator' a Python package
│   ├── lexer.py             # Lexer class and Token definition
│   ├── parser.py            # Parser class and AST node definitions
│   ├── evaluator.py         # Evaluator class and Environment definition
│   └── builtins.py          # Definitions for built-in functions (sin, cos, sqrt, etc.)
├── tests/                   # Unit and integration tests
│   ├── __init__.py
│   ├── test_lexer.py
│   ├── test_parser.py
│   ├── test_evaluator.py
│   └── test_integration.py
├── main.py                  # Command-line interface (CLI) entry point
├── requirements.txt         # Project dependencies (likely empty for this project)
└── README.md                # Project documentation
```

Module Responsibilities and Dependencies

Module	Exports	Key Imports	Purpose
`lexer.py`	`Token` (dataclass), `Lexer` class	—	Defines token types and implements the string-to-token stream conversion.
`parser.py`	`ASTNode` (ABC), `Number`, `BinaryOp`, `UnaryOp`, `Assign`, `Variable`, `FunctionCall`, `Parser` class	`lexer` (for `Token`)	Defines the AST data model and implements the token-stream-to-AST conversion.
`evaluator.py`	`Environment` class, `Evaluator` class	`parser` (for AST nodes), `builtins` (for function implementations)	Defines the symbol table and implements the tree-walking evaluation logic.
`builtins.py`	`BUILTINS` (dict mapping names to function objects)	`math` (standard library)	Provides the implementations of built-in mathematical functions.
`main.py`	—	`calculator.lexer`, `calculator.parser`, `calculator.evaluator`	Glues everything together: reads input, runs the pipeline, prints results/errors.

Dependency Flow The dependency direction is strictly one-way, following the data pipeline: `main.py` → `evaluator.py` → `parser.py` → `lexer.py`. `builtins.py` is a utility module used by `evaluator.py`. This acyclic structure prevents circular imports and reinforces the architectural model.

- `evaluator.py` depends on `parser.py` because it needs to know about the AST node types (like `BinaryOp`) to evaluate them.
- `parser.py` depends on `lexer.py` because it consumes the `Token` objects the lexer produces.
- `main.py` depends on all three core components to orchestrate the pipeline.

Key Benefits of This Structure

- **Clarity:** Each file has a single, well-defined purpose.
- **Testability:** You can write unit tests for the `Lexer` in isolation by feeding it strings and checking token lists. You can test the `Parser` by feeding it a pre-made token list and checking the AST.
- **Maintainability:** To fix a bug in variable assignment, you know to look in `evaluator.py`. To add a new operator, you modify `lexer.py` and `parser.py`.
- **Reusability:** The `parser` module could be reused in a different project (e.g., a compiler) without dragging along the evaluation logic.

Architecture Decision Record: Monolithic vs. Modular Design

- **Context:** We need to organize the code for a beginner-friendly yet professionally structured calculator project.
- **Options Considered:**
  1. **Single Script:** Put all code (lexer, parser, evaluator, AST nodes) in one `.py` file.
  2. **Modular by Component:** Split code into separate modules based on architectural responsibility (as described above).
  3. **Modular by Feature:** Split code into files like `arithmetic.py`, `variables.py`, `functions.py`.
- **Decision:** Use **Modular by Component**.
- **Rationale:** While a single file is simpler initially, it becomes unwieldy beyond ~200 lines and conflates distinct concepts. Modular by feature leads to tight coupling and scattered logic (e.g., parsing variable assignment would touch both `parser.py` and `variables.py`). Modular by component aligns perfectly with the system's architecture, reinforces separation of concerns for the learner, and scales cleanly as features are added. It is the standard practice for interpreters/compilers.
- **Consequences:**
  - **Positive:** Clear separation, easier testing, better code navigation, professional structure.
  - **Negative:** Slightly more overhead for beginners (multiple files to navigate), requires understanding of Python imports.

Options Comparison Table

Option	Pros	Cons	Suitable for this project?
Single Script	- Simplest to start - No import complexity	- Becomes a "big ball of mud" - Hard to test components in isolation - Poor separation of concerns	❌ No
Modular by Component	- Aligns with architecture - Excellent separation of concerns - Industry-standard practice	- Slight initial complexity	✅ **Yes**
Modular by Feature	- Groups related user-facing functionality	- Leads to cross-cutting concerns (parsing logic is in multiple files) - Harder to reason about data flow	❌ No

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Beginner-Friendly)	Advanced Option (For Extension)
Token Representation	`dataclass` with `type` (string) and `lexeme` (string)	`Enum` for token types, `@dataclass(frozen=True)` for immutability
AST Nodes	Simple Python classes with fields, using `isinstance()` checks in evaluator	Visitor pattern with a dedicated `ASTVisitor` class for evaluation/traversal
Symbol Table	Python `dict` wrapped in a simple `Environment` class	Hierarchical `Environment` with scopes (for future function arguments)
Error Reporting	Raise custom exceptions (`SyntaxError`, `RuntimeError`) with messages	Use `rich` or `colorama` for colored, formatted error output in CLI
CLI Interface	Simple `while` loop with `input()` / `print()`	`argparse` for command-line flags, `readline` support for history

B. Recommended File/Module Structure (Expanded) Here is the detailed skeleton for the `calculator/` package directory. Create these files before writing any substantial code.

`calculator/__init__.py`

```python
# This file can be empty. Its existence makes the 'calculator' directory a Python package.

# Optionally, you can expose key classes for a cleaner import:

# from .lexer import Lexer, Token

# from .parser import Parser, ASTNode, Number, BinaryOp, UnaryOp, Assign, Variable, FunctionCall

# from .evaluator import Evaluator, Environment
```

`calculator/lexer.py`

```python
"""
Lexer (Tokenizer) module.
Converts source code strings into a list of Tokens.
"""
# Implementation provided in Section 5.
```

calculator/parser.py

```python
"""
Parser module.
Builds an Abstract Syntax Tree (AST) from a list of Tokens.
"""
# Implementation provided in Section 6.
```

calculator/evaluator.py

```python
"""
Evaluator module.
Walks the AST to compute a numeric result, using an Environment for variables.
"""
# Implementation provided in Section 7.
```

calculator/builtins.py

```python
"""
Built-in function definitions.
"""
import math

# Dictionary mapping function names to callable Python functions.
# These will be looked up by the Evaluator when evaluating a FunctionCall node.
BUILTINS = {
    "sin": math.sin,
    "cos": math.cos,
    "sqrt": math.sqrt,
    "abs": abs,
    # Add more as needed: "tan", "log", "exp", etc.
}
```

tests/test_lexer.py

```python
"""
Unit tests for the Lexer.
"""
import pytest
from calculator.lexer import Lexer, Token

def test_tokenize_single_number():
    lexer = Lexer()
    tokens = lexer.tokenize("42")
    assert tokens == [Token(type='NUMBER', lexeme='42', position=0)]

def test_tokenize_addition():
    lexer = Lexer()
    tokens = lexer.tokenize("2 + 3")
    expected = [
        Token(type='NUMBER', lexeme='2', position=0),
        Token(type='PLUS', lexeme='+', position=2),
        Token(type='NUMBER', lexeme='3', position=4),
    ]
    assert tokens == expected

# Add more tests for other operators, parentheses, identifiers, etc.
```

main.py

```python
#!/usr/bin/env python3
"""
Command-line interface for the calculator.
"""
import sys
from calculator.lexer import Lexer
from calculator.parser import Parser
from calculator.evaluator import Evaluator, Environment

def main():
    print("Calculator (type 'quit' to exit)")
    env = Environment()
    while True:
        try:
            text = input(">> ").strip()
        except EOFError:
            break
        if text.lower() in ("quit", "exit"):
            break
        if not text:
            continue

        try:
            # 1. Lex
            lexer = Lexer()
            tokens = lexer.tokenize(text)
            # 2. Parse
            parser = Parser(tokens)
            ast = parser.parse()
            # 3. Evaluate
            evaluator = Evaluator()
            result = evaluator.evaluate(ast, env)
            print(result)
        except Exception as e:
            print(f"Error: {e}", file=sys.stderr)

if __name__ == "__main__":
    main()
```

C. Infrastructure Starter Code The `main.py` and `builtins.py` files above are complete, ready-to-use starter code. The test file provides a pattern for writing unit tests. The next sections (5, 6, 7) will provide skeleton code for the core `Lexer`, `Parser`, and `Evaluator` classes.

D. Core Logic Skeleton Code The detailed skeletons for `Lexer`, `Parser`, and `Evaluator` are provided in their respective component design sections (5, 6, 7). The structure outlined here ensures you have the correct files and imports set up to receive that code.

E. Language-Specific Hints (Python)

- **`dataclasses`:** Use `from dataclasses import dataclass` for `Token` and AST nodes. They reduce boilerplate code for classes that primarily store data.
- **Type Hints:** Add type hints (e.g., `def tokenize(self, input: str) -> list[Token]:`) to improve code clarity and enable static type checking with tools like `mypy`.
- **`math` module:** Use `import math` for functions like `sin`, `cos`, `sqrt`. Note that `math.sqrt` raises `ValueError` for negative inputs, which you may want to catch and convert to a custom calculator error.
- **Error Handling:** Define custom exception classes (e.g., `ParseError`, `RuntimeError`) that inherit from `Exception` for more precise error handling.

F. Milestone Checkpoint (After Setting Up Structure)

1. **Action:** Create the directory and file structure as shown above.
2. **Verification:** From the project root (`calculator_project/`), run `python -m pytest tests/ -v`. You should see that tests are discovered but likely fail because the modules are empty (or contain skeletons). The important thing is that no `ModuleNotFoundError` occurs.
3. **Manual Test:** Run `python main.py`. You should see the prompt `Calculator (type 'quit' to exit)`. The program will crash if you type anything (because the core components aren't implemented yet), but it should start.
4. **Success Criteria:** The project structure is created, imports work, and the CLI starts without immediate import errors.

---

4. Data Model

> **Milestone(s):** All Milestones (1-3) — The data models defined in this section form the foundation for all three milestones. Tokens are needed for Milestone 1's basic arithmetic, AST nodes expand for Milestone 2's unary and power operations, and the Environment becomes essential for Milestone 3's variables and functions.

The data model defines the fundamental building blocks our calculator uses to transform a raw input string into a computed result. These structures act as the **intermediate representations (IR)** that enable clean **separation of concerns** between the lexer, parser, and evaluator components. Think of them as three distinct "languages" the system speaks: the *lexer* translates characters into a stream of meaningful words (**Tokens**), the *parser* organizes those words into a structured sentence diagram (**AST Nodes**), and the *evaluator* interprets that diagram using a dictionary of known values (**Environment**) to arrive at the final meaning (the numeric result).

Token Types

Mental Model: The Calculator's Vocabulary Cards Imagine you're teaching a child to read a math problem. You'd first break the sentence "2 + 3 * 4" into individual concept cards: one card for the number "2", another for the "+" symbol, and so on. Each card has two key pieces of information: *what kind of concept it is* (a number, a plus sign) and *the exact text* you read from the page. The `Token` data structure serves exactly this purpose: it's a standardized "vocabulary card" the lexer produces, categorizing every meaningful piece of the input string for the parser to understand easily.

A `Token` is a simple bundle of three pieces of information, as defined in the naming conventions:

Field Name	Type	Description
`type`	`str`	The category of the token. This uses the predefined constants (e.g., `PLUS`, `NUMBER`). It tells the parser *what* this piece of the input represents.
`lexeme`	`str`	The exact sequence of characters from the input string that formed this token. For the input "12.34", the lexeme would be `"12.34"`. This is crucial for accurate error reporting and for converting string literals to actual numeric values later.
`position`	`int`	The starting index (0-based) of this token's lexeme within the original input string. This allows the system to point directly to the location of a syntax error (e.g., "Unexpected character at position 5").

The lexer's job is to scan the input and produce a list of these `Token` objects. The complete set of token `type` constants our calculator must recognize is defined below. These constants act as the agreed-upon vocabulary between the lexer and parser.

Token Type Constant	Example Lexeme(s)	Description
`NUMBER`	`"5"`, `"-2.718"`, `"3.14"`	Any numeric literal, including integers and floating-point numbers. The lexeme may include a leading minus sign if it's part of the literal (e.g., `-5` as a single token), not a unary operator.
`IDENTIFIER`	`"x"`, `"sin"`, `"myVar"`	A sequence of letters and possibly digits (starting with a letter) that names a variable or a built-in function.
`PLUS`	`"+"`	The addition operator.
`MINUS`	`"-"`	The subtraction or unary negation operator. Its specific role (binary vs. unary) is determined by the parser based on context.
`STAR`	`"*"`	The multiplication operator.
`SLASH`	`"/"`	The division operator.
`CARET`	`"^"`	The exponentiation (power) operator.
`EQUAL`	`"="`	The assignment operator (for variable assignment, not equality comparison).
`LPAREN`	`"("`	Left parenthesis, used for grouping and function calls.
`RPAREN`	`")"`	Right parenthesis.
`EOF`	N/A	A special token type indicating the **End Of File** or end of the input stream. The lexer generates this after the last real token. It is crucial for the parser to know when to stop.

> **Design Insight:** The `MINUS` token is deliberately ambiguous at the lexical level. The lexer doesn't need to decide if a `-` is for subtraction (binary) or negation (unary); it simply emits a `MINUS` token. This simplifies the lexer. The parser, with its understanding of grammatical context (e.g., a `MINUS` after another operand is binary, a `MINUS` at the start of an expression or after `(` is unary), resolves this ambiguity.

ADR: Lexer Token Granularity

> **Decision: Single `MINUS` Token Type**
>
> - **Context**: The input character `-` can represent either the binary subtraction operator (as in `5 - 3`) or the unary negation operator (as in `-5` or `- (3 + 2)`). We must decide at which stage—lexer or parser—to distinguish between these two meanings.
> - **Options Considered**:
>   1. **Single `MINUS` token**: The lexer emits a generic `MINUS` token for all `-` characters. The parser determines its role based on surrounding grammatical context.
>   2. **Separate `SUBTRACT` and `UNARY_MINUS` tokens**: The lexer analyzes the context (e.g., looks at the preceding token) and emits different token types.
> - **Decision**: Use a single `MINUS` token type.
> - **Rationale**: This adheres to the principle of keeping the lexer simple and context-free. The lexer's job is to perform a straightforward, linear scan of characters and identify basic patterns. Determining operator role requires understanding the expression's grammatical structure, which is the parser's responsibility. This clean separation makes both components easier to debug and maintain. The parser, with its recursive descent logic, is naturally equipped to handle this contextual disambiguation.
> - **Consequences**: The parser's grammar rules must explicitly handle the unary vs. binary distinction. This adds minor complexity to the parser but results in a more modular and conceptually clean design. It also correctly handles edge cases like `--5` (two consecutive unary minuses), which the parser can interpret as `-(-5)`.

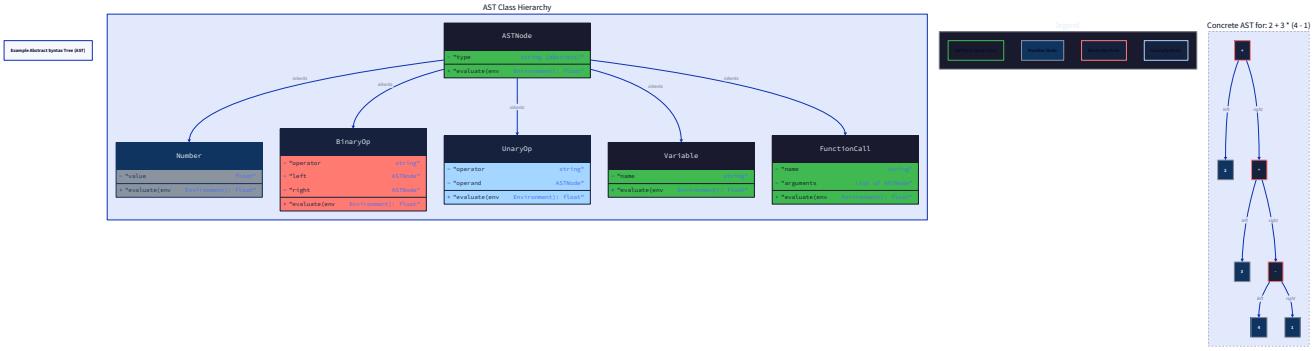
Option	Pros	Cons	Chosen?
Single `MINUS` token	Lexer is simpler, context-free, and faster. Aligns with standard compiler design principles.	Parser must handle context, adding a small amount of logic to its expression parsing.	**Yes**
Separate `SUBTRACT` / `UNARY_MINUS`	Parser grammar is slightly simpler, as the token type indicates the role.	Lexer becomes context-sensitive and more complex. It must "look ahead" or "look behind" to decide, breaking the clean linear scan model. Harder to handle consecutive minuses (`--`).	No

Abstract Syntax Tree (AST) Nodes

Mental Model: The Expression's Family Tree If you were to diagram the calculation "2 + 3 * 4" showing order of operations, you might draw a tree. The root is the `+` operation, its left child is the number `2`, and its right child is another operation `*`, which in turn has children `3` and `4`. An **Abstract Syntax Tree (AST)** is this

diagram formalized into data structures. Each node in the tree represents a single syntactic element of the expression: a literal value, an operator with its operands, a variable name, or a function call. The tree structure *abstracts away* the original string format and parentheses, leaving only the essential hierarchical relationships dictated by **precedence** and **associativity**.

All AST nodes share a common base. In our design, this is the (conceptual) `ASTNode` abstract class. In practice, for a dynamically-typed language like Python, we might simply ensure all node types have a common interface, but the mental model of a hierarchy is essential.



The following table describes each concrete node type, its purpose, and its fields. Remember, `ASTNode` in the `left` / `right` / `operand` / `value` fields means "any other AST node," allowing trees to nest arbitrarily deep.

Node Type	Fields	Description
`Number`	`value` (`float`)	Represents a numeric constant literal (e.g., `5`, `-2.5`). It is a leaf node in the AST (has no children). The `value` field stores the actual numeric value parsed from the token's lexeme.
`BinaryOp`	`operator` (`str`), `left` (`ASTNode`), `right` (`ASTNode`)	Represents a **binary operator** application like addition, subtraction, multiplication, division, or exponentiation. The `operator` field contains the token type string (e.g., `"PLUS"`). The `left` and `right` fields are AST nodes representing the operands.
`UnaryOp`	`operator` (`str`), `operand` (`ASTNode`)	Represents a **unary operator** application. Currently, this is only unary negation (e.g., `-5`). The `operator` is `"MINUS"`. The `operand` field is the AST node for the expression being negated.
`Variable`	`name` (`str`)	Represents a variable reference (e.g., `x` in the expression `x + 1`). It is a leaf node. The `name` field stores the variable's identifier as a string. During evaluation, the evaluator will look up this name in the `Environment`.
`Assign`	`name` (`str`), `value` (`ASTNode`)	Represents a variable assignment expression (e.g., `x = 5 + 3`). This node is unique because it has a side effect: it modifies the `Environment`. The `name` field is the variable identifier, and the `value` field is an AST node for the expression whose result will be stored.
`FunctionCall`	`name` (`str`), `argument` (`ASTNode`)	Represents a call to a built-in function like `sin(angle)` or `sqrt(9)`. The `name` field is the function identifier (e.g., `"sin"`). The `argument` field is an AST node for the expression being passed as the single argument.

> **Design Insight:** The AST is a pure data structure that represents *what* to compute, not *how* to compute it yet. It deliberately contains no evaluation logic. This separation allows the same AST to be evaluated multiple times (e.g., in a loop with changing variable values) or even transformed (e.g., for optimization) without modifying the parser.

ADR: AST Node Representation for Assignment

> **Decision: `Assign` as an Expression Node**
>
> - **Context**: In many programming languages, assignment (`=`) is a statement that does not produce a value. In our calculator, we want to support expressions like `x = 5` and also `y = (x = 5) + 2` . This requires assignment to behave as an expression that both modifies the environment *and* yields a value (the value being assigned).
> - **Options Considered**:
>   1. **`Assign` as an expression node**: As described above, `Assign` is an `ASTNode` subtype that has a `value` child and evaluates to that value.
>   2. **Separate statement hierarchy**: Distinguish between *expressions* (which produce values) and *statements* (which perform actions). Assignment would be a statement, and the grammar would not allow nesting it inside expressions.
> - **Decision**: Model `Assign` as an expression AST node.
> - **Rationale**: This choice keeps the syntactic and evaluation model simple and consistent for a beginner-level calculator. It mimics the behavior of assignment in languages like C or JavaScript where assignment is an expression. It allows users to write intuitive chains like `a = b = 5` . The implementation complexity is minimal—the evaluator simply updates the environment and returns the value.
> - **Consequences**: The parser must allow `Assign` nodes wherever an expression is expected. The evaluator must handle the side effect of updating the environment when evaluating an `Assign` node. This approach is less pure from a language design perspective but offers greater flexibility for interactive use.

Environment (Symbol Table)

Mental Model: The Calculator's Memory Notebook Imagine a small notebook where you write down variable names and their current values. When the calculator evaluates an expression containing `x` , it flips through this notebook to find the current value associated with `x` . The `Environment` is this notebook, implemented as a **symbol table**—a mapping from names (strings) to numeric values (floats). It is a mutable, stateful component that persists across evaluations within a session, allowing variables to be defined and later referenced.

The `Environment` is fundamentally a wrapper around a dictionary with a simple interface, primarily supporting variable assignment and lookup. Its structure is minimal:

Field Name	Type	Description
`variables`	`dict[str, float]`	The core storage, mapping variable names (keys) to their corresponding numeric values (values). It is initialized as an empty dictionary.

While the field is the primary data, we conceptualize the `Environment` as having two key operations, even if they are implemented as direct dictionary accesses in the evaluator:

1. **Lookup (`get`)**: Given a variable name, retrieve its stored `float` value. If the name is not found, it indicates an undefined variable error.
2. **Assignment (`set`)**: Given a variable name and a `float` value, update (or create) the mapping in the dictionary.

For this project, we implement a single, global environment. In a more advanced interpreter, environments might be nested to support scopes, but that is a **non-goal** for our calculator.

> **Design Insight:** The `Environment` is passed to the `Evaluator.evaluate()` method. This makes the evaluation process explicitly dependent on external state. This design allows the same AST (like a formula with variables) to be evaluated multiple times with different environments, enabling use cases like "What is `x + 1` if `x = 5` ? What if `x = 10` ?"

ADR: Environment as a Separate Object vs. Simple Dictionary

> **Decision: Separate `Environment` Class**
>
> - **Context**: The symbol table functionality could be implemented as a plain dictionary passed around, or encapsulated in a dedicated class.
> - **Options Considered**:
>   1. **Plain dictionary**: Use a Python `dict` directly in the evaluator for variable storage.
>   2. **Dedicated `Environment` class**: Wrap the dictionary in a class, potentially adding methods for lookup and assignment, and validation (e.g., checking for undefined variables).
> - **Decision**: Use a dedicated `Environment` class.
> - **Rationale**: Encapsulation improves code clarity and maintainability. It provides a clear place to centralize error handling for undefined variables (raising a clear `NameError` with the variable name). It also makes the system easier to extend later—if we wanted to add constants (like `pi`) or nested scopes, we could modify the `Environment` class without changing the evaluator's core logic. For a beginner project, it teaches good software design practice by modeling a distinct concept with its own data structure.
> - **Consequences**: Slightly more code than a raw dictionary, but the benefits in organization and future extensibility outweigh this minor cost.

Common Pitfalls

⚠️ Pitfall: Confusing Token Lexeme with Processed Value

- **Description**: Storing the raw string `"12.34"` from the `Token.lexeme` directly into an `Number` node's `value` field, instead of converting it to a `float` first.
- **Why it's wrong**: The `Number.value` field must be a numeric type (`float`) to participate in arithmetic operations. Keeping it as a string will cause type errors during evaluation or require repeated, inefficient conversions.
- **Fix**: The parser, when creating a `Number` node from a `NUMBER` token, must convert the token's `lexeme` string to a `float` using the language's standard conversion function (e.g., `float()` in Python) and store that numeric result.

⚠️ Pitfall: Incomplete Token List Missing EOF

- **Description**: The lexer produces a list of tokens for the input characters but forgets to append a final `EOF` token.
- **Why it's wrong**: The parser uses the `EOF` token as a critical sentinel to know when the entire input has been consumed. Without it, the parser may either attempt to read past the end of the token list (causing an index error) or mistakenly think the expression is incomplete.
- **Fix**: After the lexer's main loop finishes scanning characters, explicitly append a new `Token(type=EOF, lexeme='', position=end_of_input)` to the token list before returning it.

⚠️ Pitfall: AST Nodes Sharing Mutable References

- **Description**: Accidentally reusing the same mutable object (like a list or another AST node) as a child in multiple places in the AST, when independent copies are intended.
- **Why it's wrong**: If two different `BinaryOp` nodes inadvertently share the same `left` child node, modifying one (e.g., during a hypothetical AST optimization pass) would incorrectly affect the other, leading to unpredictable evaluation results.
- **Fix**: When the parser creates a new AST node, ensure its child nodes are distinct objects. In practice, since our parser builds the AST from the ground up, each node instantiation (e.g., `Number(value=5)`) creates a new object, so this pitfall is less common. However, one must be cautious if caching or reusing node objects.

⚠️ Pitfall: Environment Persistence Leading to Cross-Expression Contamination

- **Description**: Using a single, globally shared `Environment` object across multiple unrelated expression evaluations without clearing it, causing variable values from a previous calculation to affect a new one unintentionally.
- **Why it's wrong**: While persistence is desired within a calculator session (so `x=5` sets `x` for later use), if the system is used as a library to evaluate isolated expressions, leftover variable bindings from one call could leak into another, causing confusing bugs (e.g., an expression containing `y` might unexpectedly get a value from a previous, unrelated `y = 10`).
- **Fix**: Clearly define the lifetime of the `Environment`. For an interactive calculator REPL (Read-Eval-Print Loop), a single persistent environment is correct. For a library function that evaluates a single expression, a new, empty environment should be created for each call unless explicitly designed otherwise. The design should document this behavior.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Data Structures	Python `dataclasses` for Tokens and AST Nodes. Simple class for `Environment`.	Use `enum.Enum` for token types for stricter type checking. Use `abc.ABC` for an abstract `ASTNode` base class.

B. Recommended File/Module Structure Place the data model definitions in their own module(s) to keep the code organized. Here's a suggested structure:

```
calculator_project/
├── lexer.py          # Lexer class implementation
├── parser.py         # Parser class and AST node classes
├── evaluator.py      # Evaluator class and Environment class
├── tokens.py         # Token dataclass and token type constants (optional)
└── main.py          # Main application or REPL
```

For simplicity, we'll define the `Token` and all AST node classes in `parser.py`, and the `Environment` in `evaluator.py`. However, you could separate `tokens.py` for clarity.

C. Infrastructure Starter Code (COMPLETE) Here is a complete, ready-to-use definition for the `Token` dataclass and token type constants. You can place this in a `tokens.py` file or at the top of `lexer.py`.

```python
# tokens.py (or lexer.py)
"""
Token definitions for the calculator lexer.
"""

# Token type constants - these are just string values for easy comparison
PLUS = 'PLUS'
MINUS = 'MINUS'
STAR = 'STAR'
SLASH = 'SLASH'
CARET = 'CARET'
EQUAL = 'EQUAL'
LPAREN = 'LPAREN'
RPAREN = 'RPAREN'
NUMBER = 'NUMBER'
IDENTIFIER = 'IDENTIFIER'
EOF = 'EOF'

class Token:
    """Represents a single lexical token from the input string."""
    def __init__(self, type_: str, lexeme: str, position: int):
        """
        Args:
            type_: One of the token type constants (e.g., PLUS, NUMBER).
            lexeme: The actual text from the input that formed this token.
            position: The starting index of this token in the input string.
        """
        self.type = type_
        self.lexeme = lexeme
        self.position = position

    def __repr__(self):
        """Helpful representation for debugging."""
        return f"Token({self.type}, '{self.lexeme}', pos={self.position})"
```

PYTHON

D. Core Logic Skeleton Code (TODOs) Here are the skeleton definitions for the AST node classes and the `Environment` class. These should go in `parser.py` and `evaluator.py` respectively.

PYTHON

```python
# parser.py (AST Node definitions section)

"""
AST (Abstract Syntax Tree) node definitions for the calculator parser.
"""

class ASTNode:
    """Base class for all AST nodes. (Mostly a conceptual marker in this implementation.)"""
    pass

class Number(ASTNode):
    """AST node representing a numeric literal."""
    def __init__(self, value: float):
        # TODO: Store the provided float value in self.value
        pass

    def __repr__(self):
        # TODO: Return a string like "Number(3.14)" for debugging
        pass

class BinaryOp(ASTNode):
    """AST node representing a binary operation (e.g., 2 + 3)."""
    def __init__(self, operator: str, left: ASTNode, right: ASTNode):
        # TODO: Store operator (e.g., PLUS), left child node, and right child node
        pass

    def __repr__(self):
        # TODO: Return a string like "BinaryOp(PLUS, Number(2.0), Number(3.0))"
        pass

class UnaryOp(ASTNode):
    """AST node representing a unary operation (e.g., -5)."""
    def __init__(self, operator: str, operand: ASTNode):
        # TODO: Store operator (should be MINUS) and the operand child node
        pass

    def __repr__(self):
        # TODO: Return a string like "UnaryOp(MINUS, Number(5.0))"
        pass

class Variable(ASTNode):
    """AST node representing a variable reference (e.g., x)."""
    def __init__(self, name: str):
        # TODO: Store the variable name as a string
```

```
        pass


    def __repr__(self):
        # TODO: Return a string like "Variable('x')"
        pass

class Assign(ASTNode):
    """AST node representing a variable assignment (e.g., x = 5)."""
    def __init__(self, name: str, value: ASTNode):
        # TODO: Store the variable name and the AST node for the value expression
        pass


    def __repr__(self):
        # TODO: Return a string like "Assign('x', Number(5.0))"
        pass

class FunctionCall(ASTNode):
    """AST node representing a function call (e.g., sin(angle))."""
    def __init__(self, name: str, argument: ASTNode):
        # TODO: Store the function name and the AST node for the argument expression
        pass


    def __repr__(self):
        # TODO: Return a string like "FunctionCall('sin', Number(1.57))"
        pass
```

PYTHON

```python
# evaluator.py (Environment class)

"""
Environment (symbol table) for storing variable values.
"""

class Environment:
    """Maps variable names to their numeric values."""
    def __init__(self):
        # TODO: Initialize an empty dictionary for self.variables
        # Example: self.variables = {}
        pass

    def get(self, name: str) -> float:
        """
        Retrieve the value of a variable.

        Args:
            name: The variable name to look up.

        Returns:
            The numeric value associated with the name.

        Raises:
            NameError: If the variable name is not found in the environment.
        """
        # TODO 1: Check if 'name' exists in self.variables
        # TODO 2: If it exists, return the corresponding value
        # TODO 3: If it does NOT exist, raise a NameError with a helpful message
        #         Example: raise NameError(f"Undefined variable '{name}'")
        pass

    def set(self, name: str, value: float) -> None:
        """
        Set or update the value of a variable.

        Args:
            name: The variable name.
            value: The numeric value to assign.
        """
        # TODO: Store the value in self.variables with 'name' as the key
        # Example: self.variables[name] = value
```

```
        pass


    def __repr__(self):
        # Optional: Helpful representation for debugging
        return f"Environment({self.variables})"
```

E. Language-Specific Hints

- **Python `dataclasses`**: You can decorate the `Token` class with `@dataclass` to automatically generate `__init__` and `__repr__`. However, the skeleton above uses a plain class for clarity.
- **Type Hints**: Adding type hints (as shown in the skeletons) is highly recommended. They serve as documentation and help catch errors with tools like `mypy`.
- **Float Conversion**: Use `float(token.lexeme)` to convert a `NUMBER` token's lexeme. Be aware that this will raise a `ValueError` if the lexeme is malformed (e.g., `"12.34.56"`). The lexer should ensure only valid numeric strings are emitted as `NUMBER` tokens.
- **`__repr__` Methods**: Implementing `__repr__` for all AST nodes is invaluable for debugging. It allows you to print the parsed tree and see its structure clearly.

5. Component Design: Lexer (Tokenizer)

> **Milestone(s):** Milestone 1 (Basic Arithmetic), Milestone 2 (Unary and Power), Milestone 3 (Variables and Functions) — The lexer is the first critical component in the pipeline, responsible for converting raw input characters into a structured token stream. All three milestones rely on its ability to correctly identify numbers, operators, parentheses, identifiers, and the assignment operator.

The **lexer** (also called a tokenizer or scanner) acts as the front-line interpreter of raw user input. Its job is to transform a continuous string of characters — like `"2 + 3 * (4 - 1)"` — into a structured sequence of discrete, meaningful symbols called **tokens**. This transformation is the essential first step that allows the parser to understand the grammatical structure of the expression without getting bogged down in character-level details such as whitespace or multi-digit numbers.

Mental Model: The Text Scanner

Imagine you are reading a handwritten recipe. Your eyes scan the page, grouping sequences of letters into words (like "flour"), recognizing individual punctuation marks (like ":"), and ignoring irrelevant spaces and line breaks. You don't interpret the recipe's meaning yet; you simply identify the basic lexical units. The lexer does exactly this for mathematical expressions.

- **Characters → Words:** The lexer reads the input string character by character. When it sees `'2'`, `'3'`, and `'*'` in sequence, it doesn't treat them as three separate things. It recognizes `"23"` as a single number and `"*"` as a single operator.
- **Ignoring Whitespace:** Just as you skip over spaces between words, the lexer consumes and discards any whitespace characters (spaces, tabs, newlines) without producing tokens for them. This makes the expression `"2+3"` equivalent to `"2 + 3"` from the parser's perspective.
- **Identifying Boundaries:** It knows when one token ends and another begins. For example, in `"sin(45)"`, it correctly identifies `"sin"` as an identifier and `"("` as a separate token, not as part of the identifier.

This scanning process produces a clean, unambiguous list of tokens that the parser can process sequentially, much like a person reading a list of ingredients before following the cooking instructions.

Interface

The lexer's interface is deliberately simple and stateless. It takes a raw string as input and produces a list of tokens. Its only responsibility is accurate identification, not interpretation.

Method Name	Parameters	Returns	Description
`tokenize`	`input: str`	`list[Token]`	The sole public method. Scans the entire input string from start to finish, returning a sequence of `Token` objects representing all meaningful lexical units found. The list ends with a special `EOF` (End-Of-File) token.

The `Token` data structure is the lexer's output and is defined in the Data Model (Section 4). For convenience, its structure is reiterated here:

Field Name	Type	Description
`type`	`str`	The category of the token. Must be one of the defined constants (e.g., `NUMBER`, `PLUS`, `IDENTIFIER`).
`lexeme`	`str`	The exact substring of the input that this token represents (e.g., `"42"`, `"+"`, `"x"`).
`position`	`int`	The index (usually starting from 0) in the original input string where this token's lexeme begins. Crucial for generating helpful error messages.

Internal Algorithm

The lexer implements a **single-pass, look-ahead scanning algorithm**. It maintains a pointer (`current_pos`) into the input string and examines one character at a time, deciding when to start a new token, what type it is, and when to finish it.

The core procedure for `tokenize(input)` is as follows:

1. **Initialization:** Create an empty list `tokens` . Set `current_pos = 0` . Determine the total length `n` of the input string.
2. **Main Scanning Loop:** While `current_pos` is less than `n` :
    1. **Skip Whitespace:** Read the character at `current_pos` . If it is a space (`' '`), tab (`'\t'`), or newline (`'\n'`), increment `current_pos` and continue to the next loop iteration. This consumes all consecutive whitespace without creating tokens.
    2. **Start Token:** Record the `start_pos = current_pos` . This becomes the token's `position` field.
    3. **Classify and Consume:** Based on the character at `current_pos` (the "lookahead"), decide the token type and consume all characters belonging to it.
        - **Single-Character Token:** If the lookahead is `'+'` , `'-'` , `'*'` , `'/'` , `'^'` , `'='` , `'('` , or `')'` , create a token with the corresponding type (e.g., `PLUS` , `LPAREN`). The `lexeme` is that single character. Advance `current_pos` by 1.
        - **Multi-Character Number:** If the lookahead is a digit (`'0'` - `'9'`) or a dot (`.` for a decimal starting with zero), enter **number-scanning mode**.
            1. Consume consecutive digits.
            2. If a dot (`.`) is encountered and the following character is a digit, consume the dot and then consume consecutive digits after it.
            3. The resulting lexeme (e.g., `"123"` , `"3.14"` , `".5"`) is converted to a float value later by the parser/evaluator. For now, create a `NUMBER` token with the lexeme.
        - **Identifier (Variable/Function Name):** If the lookahead is an alphabetic character (`'a'` - `'z'` , `'A'` - `'Z'`) or an underscore (`_`), enter **identifier-scanning mode**.
            1. Consume consecutive alphanumeric characters (letters, digits, underscores).
            2. Create an `IDENTIFIER` token with the consumed lexeme (e.g., `"x"` , `"sin"` , `"my_var"`).
        - **Unknown Character:** If the lookahead does not match any of the above patterns, raise a **LexicalError** indicating an unexpected character at position `current_pos` .
    4. **Append Token:** Add the newly created token to the `tokens` list.
3. **Termination:** After the loop ends (the entire input has been scanned), append a final `EOF` token to the `tokens` list. This special token signals to the parser that there are no more tokens to consume. Return the `tokens` list.

> **Key Insight:** The lexer is *greedy* — it always tries to form the longest possible valid token. For example, when seeing `"23"` , it reads both digits as a single `NUMBER` token, not as two separate tokens `"2"` and `"3"` .

ADR: Lexer Token Granularity

> **Decision: Single `MINUS` Token Type**
>
> - **Context:** The minus symbol (`-`) can serve two grammatical roles in our calculator language: as a **binary operator** for subtraction (e.g., `5 - 3`) and as a **unary operator** for negation (e.g., `-5` or `- (2 + 3)`). We must decide at which stage of processing this distinction is made.
> - **Options Considered:**
>   1. **Distinguish at Lexer:** Emit different token types (e.g., `UNARY_MINUS` and `BINARY_MINUS`) based on immediate context during scanning.
>   2. **Single Token at Lexer:** Emit only a `MINUS` token. Let the parser determine its role based on the grammatical context where it appears.
> - **Decision:** We will emit a single `MINUS` token type from the lexer.
> - **Rationale:** The grammatical role of `-` is a **syntactic** or **parsing** concern, not a **lexical** one. The lexeme `"-"` is identical in both cases. Determining if a `-` is unary or binary requires understanding the surrounding token sequence (e.g., a `-` after an open parenthesis `(` or another operator is likely unary). This contextual analysis is the parser's job. Keeping the lexer simple and context-free improves its clarity, testability, and adherence to the **separation of concerns** principle.
> - **Consequences:** The parser's grammar rules must be designed to handle the `MINUS` token in both prefix (unary) and infix (binary) positions. This adds a small amount of complexity to the parser but results in a cleaner, more modular design.

Option	Pros	Cons	Chosen?
Distinguish at Lexer	Parser grammar is simpler.	Lexer must peek at previous/next token, breaking its context-free nature. Harder to read and maintain. Error-prone for edge cases (e.g., `--5`).	No
Single Token at Lexer	Lexer is simple, fast, and purely context-free. Clear separation of concerns.	Parser must implement rules for both unary and binary minus, adding minor complexity.	**Yes**

Common Pitfalls

⚠️ Pitfall: Mishandling Floating-Point Numbers

- **Description:** Incorrectly scanning numbers like `.5`, `123.`, or `12.34.56`. A common mistake is to consume a dot only if preceded by a digit, which would miss valid numbers like `.5`.
- **Why it's Wrong:** The calculator should accept standard decimal notation. Rejecting `.5` or incorrectly parsing `123.` as the number `123` followed by a dot operator will cause syntax errors or wrong calculations.
- **How to Fix:** Implement a robust number-scanning state machine. Consume an optional integer part (digits), then if a dot is seen, consume it and require at least one digit in the fractional part. Alternatively, follow the common convention: allow numbers starting with a dot (`.5`) but not ending with a dot (`123.` is invalid unless it's `123.0`). For this project, accepting `123.` as `123.0` is a reasonable simplification.

⚠️ Pitfall: Forgetting to Skip Whitespace

- **Description:** The lexer produces tokens for spaces or, more subtly, fails to advance the `current_pos` pointer when encountering whitespace, leading to an infinite loop.
- **Why it's Wrong:** Whitespace is not meaningful in arithmetic expressions (except as a separator). Including it in the token stream forces the parser to handle it, unnecessarily complicating the grammar. An infinite loop will hang the program.
- **How to Fix:** At the start of each loop iteration, before attempting to classify a character, repeatedly consume and discard any whitespace characters until a non-whitespace character is found.

⚠️ Pitfall: Incorrectly Detecting End-of-Input

- **Description:** The scanning loop terminates but forgets to append the final `EOF` token, or incorrectly uses an empty string or `None` to signal the end.
- **Why it's Wrong:** The parser consumes tokens sequentially and needs a clear, universal signal that no more tokens are available. Without an `EOF` token, the parser might attempt to access tokens beyond the list's bounds, causing an `IndexError`.
- **How to Fix:** Always append a token with `type=EOF` (and a sentinel `lexeme` like `""`) at the end of the `tokenize` method. The parser should check for this token type explicitly.

⚠️ Pitfall: Conflating the Minus Sign with Hyphen in Identifiers

- **Description:** Allowing the minus character `-` within identifiers (e.g., `my-var`), or conversely, mistakenly starting an identifier scan when seeing a `-`.
- **Why it's Wrong:** In our grammar, `-` is exclusively an operator. Allowing it in identifiers would make parsing ambiguous (is `a-b` subtraction or a variable name?). The lexer must correctly classify the character based on its position.
- **How to Fix:** The classification step must be exclusive and ordered. Check for single-character operators *before* checking for identifier-starting characters. A `-` is always an operator token, never the start of an identifier.

Implementation Guidance (Python)

This section provides the complete starter code for the lexer component. The `Token` dataclass and constant definitions are provided as they are foundational. The `Lexer` class contains the skeleton of the `tokenize` method with detailed TODO comments that map directly to the algorithm steps described above.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Token Representation	`dataclass` with `type`, `lexeme`, `position`	`Enum` for token types, `NamedTuple` for performance
Character Classification	Python's `str.isdigit()`, `str.isalpha()`, `str.isspace()`	Pre-computed lookup tables or regex for speed (premature optimization for this project)
Error Reporting	Raise a custom `LexerError` exception with position info	Collect multiple errors (not required for a calculator)

Recommended File/Module Structure:

```
calculator_project/
├── lexer.py          # Contains the Lexer class and Token dataclass
├── parser.py         # (For later milestones)
├── evaluator.py      # (For later milestones)
├── ast_nodes.py      # (For later milestones) AST node classes
├── environment.py    # (For Milestone 3) Environment class
└── main.py           # Entry point for the application
```

Core Logic Skeleton Code:

PYTHON

```python
# lexer.py


# Import for type hints (optional but recommended for clarity)

from dataclasses import dataclass

from typing import List


# --- Token Types (Constants) ---

# These string constants are used as the `type` field in Token objects.

# Using constants prevents typos and makes code more readable.

PLUS = 'PLUS'

MINUS = 'MINUS'

STAR = 'STAR'

SLASH = 'SLASH'

CARET = 'CARET'

EQUAL = 'EQUAL'

LPAREN = 'LPAREN'

RPAREN = 'RPAREN'

NUMBER = 'NUMBER'

IDENTIFIER = 'IDENTIFIER'

EOF = 'EOF'


@dataclass

class Token:

    """Represents a single meaningful unit scanned from the input string."""

    type: str   # One of the token type constants defined above

    lexeme: str # The exact substring from the input

    position: int # Index in the input where this token starts


class LexerError(Exception):

    """Exception raised when the lexer encounters an invalid character."""

    def __init__(self, message: str, position: int):

        super().__init__(f"{message} at position {position}")

        self.position = position


class Lexer:

    """Converts a source string into a list of tokens."""



    def tokenize(self, input_str: str) -> List[Token]:

        """

        The main lexer method. Scans the entire input string and returns a

        list of tokens. The final token is always of type EOF.



        Args:
```

```
    input_str: The raw expression string (e.g., "2 + 3 * (4 - 1)")

Returns:
    A list of Token objects in the order they appear.

Raises:
    LexerError: If an unrecognized character is encountered.
"""
tokens = []
current_pos = 0
n = len(input_str)

# TODO 1: Start main scanning loop. Continue while current_pos < n.
while current_pos < n:
    # TODO 2: Skip whitespace. Use str.isspace().
    #   While current_pos < n and input_str[current_pos].isspace():
    #       current_pos += 1
    #   If this loop moved us to the end of input, break.
    while current_pos < n and input_str[current_pos].isspace():
        current_pos += 1
    if current_pos >= n:
        break

    start_pos = current_pos
    current_char = input_str[current_pos]

    # TODO 3: Classify the current character and build the appropriate token.

    # TODO 3a: Single-character tokens.
    #   Check if current_char is one of: '+', '-', '*', '/', '^', '=', '(', ')'.
    #   If yes, create a token with the corresponding type constant.
    #   Example: if current_char == '+': token = Token(PLUS, '+', start_pos)
    #   Then, current_pos += 1 and append token to tokens list.
    if current_char in '+-*/^=()':
        if current_char == '+':
            token_type = PLUS
        elif current_char == '-':
            token_type = MINUS
        elif current_char == '*':
            token_type = STAR
```

```
        elif current_char == '/':
            token_type = SLASH
        elif current_char == '^':
            token_type = CARET
        elif current_char == '=':
            token_type = EQUAL
        elif current_char == '(':
            token_type = LPAREN
        else:  # current_char == ')'
            token_type = RPAREN
        tokens.append(Token(token_type, current_char, start_pos))
        current_pos += 1
        continue  # Move to next loop iteration


    # TODO 3b: Multi-character numbers.
    #   Check if current_char.isdigit() or current_char == '.'.
    #   If yes, we have a number. Initialize an empty string `lexeme`.
    #   While current_pos < n and (input_str[current_pos].isdigit() or input_str[current_pos] == '.'):
    #       Append the character to lexeme, current_pos += 1.
    #   Important: This simple rule allows multiple dots. You may want to add validation
    #   (e.g., ensure at most one dot) or handle it later in the parser.
    #   Create a NUMBER token with the lexeme.
    if current_char.isdigit() or current_char == '.':
        lexeme = ''
        while current_pos < n and (input_str[current_pos].isdigit() or input_str[current_pos] == '.'):
            lexeme += input_str[current_pos]
            current_pos += 1
        tokens.append(Token(NUMBER, lexeme, start_pos))
        continue


    # TODO 3c: Identifiers (for variables and functions, Milestone 3).
    #   Check if current_char.isalpha() or current_char == '_'.
    #   If yes, we have an identifier. Initialize an empty string `lexeme`.
    #   While current_pos < n and (input_str[current_pos].isalnum() or input_str[current_pos] == '_'):
    #       Append the character to lexeme, current_pos += 1.
    #   Create an IDENTIFIER token with the lexeme.
    if current_char.isalpha() or current_char == '_':
        lexeme = ''
        while current_pos < n and (input_str[current_pos].isalnum() or input_str[current_pos] == '_'):
            lexeme += input_str[current_pos]
```

```
                current_pos += 1
            tokens.append(Token(IDENTIFIER, lexeme, start_pos))
            continue


        # TODO 4: If none of the above matched, it's an invalid character.
        #   Raise a LexerError with a descriptive message and the current position.
        #   Example: raise LexerError(f"Unexpected character '{current_char}'", start_pos)
        raise LexerError(f"Unexpected character '{current_char}'", start_pos)


    # TODO 5: After the loop, append an EOF token to signify the end of the token stream.
    #   The position for EOF can be set to `n` (the length of the input).
    tokens.append(Token(EOF, '', n))


    return tokens
```

Language-Specific Hints:

- Using `dataclass` for `Token` automatically generates useful `__init__`, `__repr__`, and `__eq__` methods, making debugging easier.
- The `str` methods `.isdigit()`, `.isalpha()`, `.isalnum()`, and `.isspace()` are reliable for character classification in this context and handle Unicode correctly (though our calculator only needs ASCII).
- Raising a custom `LexerError` exception allows the main program to catch it and present a user-friendly error message, possibly highlighting the erroneous position in the input.

Milestone Checkpoint (End of Lexer Implementation):

After implementing the lexer, you should be able to test it in isolation. Create a simple test script or run the following in an interactive Python session:

PYTHON

```
from lexer import Lexer, PLUS, MINUS, STAR, SLASH, NUMBER, LPAREN, RPAREN, EOF


lexer = Lexer()

tokens = lexer.tokenize("2 + 3 * (4 - 1)")

for token in tokens:

    print(f"Token(type={token.type}, lexeme='{token.lexeme}', position={token.position})")
```

Expected Output:

```
Token(type=NUMBER, lexeme='2', position=0)
Token(type=PLUS, lexeme='+', position=2)
Token(type=NUMBER, lexeme='3', position=4)
Token(type=STAR, lexeme='*', position=6)
Token(type=LPAREN, lexeme='(', position=8)
Token(type=NUMBER, lexeme='4', position=9)
Token(type=MINUS, lexeme='-', position=11)
Token(type=NUMBER, lexeme='1', position=13)
Token(type=RPAREN, lexeme=')', position=14)
Token(type=EOF, lexeme='', position=15)
```

Verify:

1. Whitespace is ignored (note the positions jump from 0 to 2, etc.).
2. Multi-digit numbers and identifiers (if you've added that part) are scanned as single tokens.
3. The final `EOF` token is present.
4. An expression like `"sin(45)"` produces tokens: `IDENTIFIER('sin')`, `LPAREN`, `NUMBER('45')`, `RPAREN`, `EOF`.

If you get errors or the output doesn't match, use the **Debugging Tips** below.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Infinite loop or program hangs	The `current_pos` pointer is not advancing, often due to missing `continue` statements or incorrect loop condition.	Add a print statement at the top of the loop: `print(f"Loop: pos={current_pos}, char='{input_str[current_pos] if current_pos < n else 'EOF'}")`.	Ensure each classification branch (`if/elif`) ends with `current_pos += 1` and a `continue`, or that the pointer is updated within the while loops for numbers/identifiers.
`IndexError` when accessing `input_str[current_pos]`	The loop condition `current_pos < n` is false, but code still tries to read the character. This often happens after the whitespace-skipping loop.	Check the logic after the whitespace loop. Did you break correctly if `current_pos >= n`?	After the whitespace loop, add a check: `if current_pos >= n: break`.
Numbers like `.5` are not recognized	The condition for starting a number only checks `.isdigit()`, missing the leading dot case.	Test with input `".5"`. The lexer will likely raise an `LexerError` for an unexpected dot.	Update the condition to `current_char.isdigit() or current_char == '.'`.
`--5` causes a LexerError on the second `-`	The identifier-scanning logic mistakenly treats `-` as a valid identifier character.	Ensure the single-character token check (for `-`) comes **before** the identifier check in your `if/elif` chain.	Reorder the classification branches: single chars first, then numbers, then identifiers.

6. Component Design: Parser

> **Milestone(s):** Milestone 1 (Basic Arithmetic), Milestone 2 (Unary and Power), Milestone 3 (Variables and Functions) — The parser is the central component that transforms the linear sequence of tokens produced by the lexer into a structured Abstract Syntax Tree (AST), correctly encoding operator precedence, associativity, and grouping for all three milestones.

Mental Model: The Grammar Rule Enforcer

Imagine you are a traffic controller at a complex intersection where different types of vehicles (cars, trucks, emergency vehicles) arrive in a single-file line. Your job is to direct them into a multi-lane highway where the lane they enter determines the order they will exit. Emergency vehicles (parentheses) get immediate priority, regardless of what's around them. Trucks (multiplication/division) go into a faster lane that merges before the car lane, and cars (addition/subtraction) go into the standard lane. Furthermore, when multiple trucks arrive in sequence, they must merge in the order they arrived (left-to-right), but when a convoy of specialty vehicles (exponentiation) arrives, they must link up from the last to the first (right-to-left). The parser is this traffic controller. It examines the stream of tokens (vehicles) and, by applying a set of predefined grammar rules (traffic laws), builds a structured roadmap (the AST) that dictates the exact order of operations (exit sequence). This mental model helps visualize why a naive left-to-right evaluation fails and how precedence and associativity rules must be enforced structurally.

Interface

The parser's primary responsibility is to consume the flat list of `Token` objects and produce a single, hierarchical `ASTNode` representing the entire expression. Its interface is minimal and focused.

Method	Parameters	Returns	Description
`parse`	`tokens: list[Token]`	`ASTNode` (root)	The main entry point. Transforms a token sequence (ending with `EOF`) into an Abstract Syntax Tree. It validates the overall structure (e.g., no stray tokens) and returns the root node of the tree. Raises a `SyntaxError` for malformed expressions.

Preconditions:

- The input `tokens` list is non-empty and its last element is of type `EOF`.
- The token list is the direct output of `Lexer.tokenize()`.

Postconditions:

- If successful, returns an `ASTNode` that is the root of a valid AST for the expression.
- The entire token sequence (except `EOF`) is consumed.
- If a syntax error is encountered, raises a `SyntaxError` with a descriptive message and the position of the error.

Side Effects:

- The parser maintains internal state (a current position index) as it traverses the token list, but this does not persist beyond the call.

Internal Algorithm (Recursive Descent)

The chosen algorithm is **recursive descent parsing with precedence climbing**. This method structures the parser as a set of mutually recursive functions, each corresponding to a level of precedence in the grammar. The flow mirrors the "traffic controller" mental model by having higher-priority operators handled by functions that are called deeper in the recursion, ensuring they bind tighter in the resulting tree.

The grammar for our calculator (covering all three milestones) can be summarized as follows, from lowest to highest precedence:

```
expression  → assignment
assignment  → identifier "=" expression | logical_or
logical_or  → logical_and ( ("||") logical_and )*
logical_and → equality ( ("&&") equality )*
equality    → comparison ( ("==" | "!=") comparison )*
comparison  → term ( (">" | ">=" | "<" | "<=") term )*
term        → factor ( ("+" | "-") factor )*
factor      → unary ( ("*" | "/") unary )*
unary       → ("-" | "+") unary | power
power       → primary ("^" unary)*
primary     → NUMBER | IDENTIFIER | "(" expression ")" | IDENTIFIER "(" expression ")"
```

> **Note:** For our calculator project, we do not implement logical, equality, or comparison operators (they are non-goals). However, the grammar structure is shown for completeness and to illustrate how the pattern extends. Our implementation will focus on `assignment`, `term`, `factor`, `unary`, `power`, and `primary`.

The parsing algorithm is implemented as a series of methods in the `Parser` class. The parser keeps track of the current token using an index into the token list. The following numbered steps describe the algorithm for the core expression types relevant to our milestones.

1. **Initialization:** The parser receives the token list and initializes a current token index to `0`. The method at the top of the precedence hierarchy (`parse_assignment` or `parse_expression`) is called.
2. **Parsing Assignment (Milestone 3):** To parse an `assignment` rule (`IDENTIFIER "=" expression`), the parser must look ahead.
    1. It first attempts to parse a `logical_or` (which, for us, is effectively `term`). This yields a potential left-hand side node.
    2. It then checks if the current token is `EQUAL` (`=`) and if the left-hand side node is of type `Variable`. If both conditions are true, it consumes the `=` token and recursively parses the right-hand side (`expression`). It then returns an `Assign` node.
    3. If the `=` is present but the left-hand side is not a `Variable`, a syntax error is raised (e.g., `"5 = 3"` is invalid).
    4. If no `=` is found, it simply returns the node parsed in step 2.1.
3. **Parsing Terms (Addition/Subtraction - Milestone 1):** The `parse_term` method handles the lowest-precedence binary operators (`+`, `-`). It follows a **left-associative** pattern.
    1. It first calls `parse_factor` (the next higher precedence level) to get the left operand.
    2. It then enters a loop while the current token is `PLUS` or `MINUS`.
    3. Inside the loop, it saves the operator, consumes the token, and then calls `parse_factor` again to get the right operand.
    4. It combines the left operand, operator, and right operand into a new `BinaryOp` node. This new node becomes the left operand for the next iteration of the loop.
    5. This loop constructs a left-associative tree: `a - b - c` becomes `((a - b) - c)`.
4. **Parsing Factors (Multiplication/Division - Milestone 1):** The `parse_factor` method is identical in structure to `parse_term` but calls `parse_unary` and loops on `STAR` (`*`) and `SLASH` (`/`) operators. It also enforces left associativity.
5. **Parsing Unary Operators (Milestone 2):** The `parse_unary` method handles prefix operators (`+`, `-`).
    1. It checks if the current token is `PLUS` or `MINUS`.
    2. If it is, it consumes the token and recursively calls `parse_unary` (not `parse_power`) to get its single operand. This allows multiple unary operators to stack (e.g., `--5`).
    3. It returns a `UnaryOp` node.
    4. If the current token is not a unary operator, it proceeds to call `parse_power`.
6. **Parsing Power (Exponentiation - Milestone 2):** The `parse_power` method handles the `^` operator, which is **right-associative** and has higher precedence than unary.
    1. It first calls `parse_primary` to get the left operand (base).
    2. It then checks if the current token is `CARET` (`^`).
    3. If it is, it consumes the `^` token and then **crucially calls `parse_unary`** (or `parse_power` itself for right associativity) to get the right operand (exponent). Calling `parse_unary` ensures `-2^2` is parsed as `-(2^2)` because the `-` is a unary operator with lower precedence than `^`. To achieve right associativity (`2^3^2` = `2^(3^2)`), the recursive call should be to `parse_power` itself.

4. It returns a `BinaryOp` node. The right-associative tree is built naturally by the recursion: parsing `2 ^ 3 ^ 2` results in a call chain that effectively builds `BinaryOp("^", left=2, right=BinaryOp("^", left=3, right=2))`.

7. **Parsing Primary Expressions (All Milestones):** The `parse_primary` method handles the basic building blocks: numbers, identifiers, parentheses, and function calls.
    1. If the token is `NUMBER`, it consumes it and returns a `Number` node.
    2. If the token is `IDENTIFIER`, it consumes it and then looks ahead.
        - If the next token is `LPAREN`, it's a function call. It consumes the `LPAREN`, calls `parse_expression` to parse the argument, and then consumes the expected `RPAREN`. It returns a `FunctionCall` node.
        - Otherwise, it's a variable reference, and it returns a `Variable` node.
    3. If the token is `LPAREN`, it consumes it, calls the top-level `parse_expression` to parse the grouped expression, and then consumes the expected `RPAREN`. It returns the parsed sub-expression node.
    4. If none of the above match, it raises a syntax error (e.g., "Unexpected token ')'").

8. **Consumption and Synchronization:** Each parsing method uses a helper `consume` method to match and advance past an expected token type, raising a syntax error if the current token doesn't match. This provides clear error messages. The parser does not implement sophisticated error recovery; it stops at the first error.

The recursive descent flow for the expression `-2 + 3 * (4 - 1)` is depicted in the following diagram, showing how control moves between methods to respect precedence:

ADR: Parser Strategy Choice

Decision: Use Recursive Descent Parsing over Pratt Parsing

- **Context**: We need a parsing algorithm for the calculator that correctly handles operator precedence, associativity, unary operators, parentheses, and function calls. The algorithm must be understandable by beginners and easy to implement incrementally across three milestones.
- **Options Considered**:
  1. **Recursive Descent Parsing**: Implement a set of recursive functions that directly mirror the grammar rules, using a technique sometimes called "precedence climbing" for binary operators.
  2. **Pratt Parsing (Operator-Precedence Parsing)**: Use a single `parse_expression` function that accepts a precedence parameter and uses lookup tables to determine how to parse based on the current token's binding power.
- **Decision**: Recursive Descent Parsing is selected as the primary implementation strategy.
- **Rationale**:
  - **Conceptual Clarity for Beginners**: Recursive descent maps directly to the grammar rules (e.g., `parse_term`, `parse_factor`). Each function has a clear, single responsibility, making the control flow easier to trace and debug for someone new to parsing.
  - **Incremental Implementation**: The structure naturally supports the project milestones. Milestone 1 implements `parse_term` and `parse_factor`. Milestone 2 adds `parse_unary` and `parse_power`. Milestone 3 adds `parse_primary` extensions for functions and `parse_assignment`. Each addition is localized.
  - **Explicit Control Flow**: The handling of associativity (loops for left, recursion for right) is visually explicit in the code, whereas Pratt parsing encapsulates this logic in a table, which can be more abstract.
  - **Adequate for Grammar Complexity**: The calculator's grammar is simple and not excessively recursive. Recursive descent performs well and is straightforward to implement without the added complexity of precedence/led/nud tables required by Pratt parsing.
- **Consequences**:
  - **Positive**: The parser code will be longer but more readable and pedagogically valuable. Error reporting can be more contextual because each parsing function knows what grammatical construct it's trying to parse.
  - **Negative**: The code may have more duplication (the left-associative loop pattern is repeated in `parse_term` and `parse_factor`). Adding a new binary operator with a new precedence level requires creating a new parsing method, which is more work than adding an entry to a Pratt table. However, this is acceptable given the fixed operator set.

Option	Pros	Cons	Chosen?
Recursive Descent	Intuitive mapping to grammar; Easy to debug; Excellent for incremental development; Clear error context.	Some code duplication for similar operator levels; Slightly more code for many operators.	**Yes**
Pratt Parsing	Very concise for many operators; Elegant handling of precedence via binding power; Easier to extend with new operators.	More abstract; Table-driven logic can be harder for beginners to grasp; Error messages can be less specific.	No

Common Pitfalls

⚠️ Pitfall: Infinite Recursion in Left-Associative Parsing

- **Description**: When implementing `parse_term` or `parse_factor`, mistakenly calling `parse_term` again inside the loop instead of calling the next higher-precedence function (`parse_factor` or `parse_unary`). For example, in `parse_term`, the right operand must be fetched via `parse_factor`, not `parse_term`. Calling `parse_term` again creates an infinite loop for an expression like `1 + 2`.
- **Why it's wrong**: The parser never advances past the current precedence level, causing a stack overflow.
- **How to fix**: Ensure the recursive call in the loop of a binary operator parsing method always goes to the method for the **next higher** precedence level.

⚠️ Pitfall: Incorrect Associativity for Exponentiation

- **Description**: Implementing the power operator as left-associative (using a loop like `parse_term`) or failing to make the recursive call correctly. This causes `2^3^2` to evaluate as `(2^3)^2 = 64` instead of the correct `2^(3^2) = 512`.
- **Why it's wrong**: Mathematically, exponentiation is right-associative. A left-associative parse tree yields an incorrect result.
- **How to fix**: In `parse_power`, after seeing a `^` token, recursively call `parse_power` (or `parse_unary` if you want to enforce precedence relative to unary) to parse the exponent. This nested recursion builds a right-heavy tree.

⚠️ Pitfall: Misplacing Unary Operator Precedence

- **Description**: Placing the `parse_unary` method at a precedence level below `parse_power` (e.g., calling `parse_unary` from `parse_factor`). This would cause `-2^2` to be parsed as `(-2)^2 = 4`.
- **Why it's wrong**: Unary operators (like negation) have lower precedence than exponentiation. The expression `-2^2` should be `-(2^2) = -4`.
- **How to fix**: Ensure the grammar order is `unary → power → factor → term`. In code, `parse_unary` should call `parse_power`, and `parse_power` should call `parse_primary`.

⚠️ Pitfall: Failing to Handle the "Lookahead" in Primary

- **Description**: In `parse_primary`, when an `IDENTIFIER` is seen, immediately returning a `Variable` node without checking if the next token is a left parenthesis. This makes function calls like `sin(30)` impossible to parse.
- **Why it's wrong**: The grammar distinguishes between variables and function calls based on the presence of `(` after the identifier.
- **How to fix**: Use a "lookahead" technique. Store the identifier name, consume the `IDENTIFIER` token, then inspect (but do not consume) the *next* token (`peek`). If it's `LPAREN`, parse it as a function call. Otherwise, return a `Variable` node.

⚠️ Pitfall: Inadequate Syntax Error Reporting

- **Description**: Raising a generic error like "Syntax error" without indicating where in the input the problem occurred or what was expected.
- **Why it's wrong**: It leaves the user guessing, especially for longer expressions.
- **How to fix**: Always include the token's position (`token.position`) and its lexeme in the error message. When expecting a specific token (e.g., `RPAREN`), state what was expected and what was actually found. Example: `f"Syntax error at position {token.position}: Expected ')' after expression, but found '{token.lexeme}'"`.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Parser Algorithm	Recursive Descent (hand-coded)	Pratt Parser (table-driven)
AST Representation	Python classes with an `evaluate` method	Visitor Pattern for separate evaluation/traversal
Error Handling	Raise `SyntaxError` with message	Custom `ParseError` class with line/col info

B. Recommended File/Module Structure:

```
calculator_project/
├── lexer.py          # From previous section
├── tokens.py         # Token class and constants
├── ast_nodes.py      # AST node class definitions
├── parser.py         # Parser class (this section)
├── evaluator.py      # Evaluator class (next section)
├── environment.py    # Environment class
└── main.py           # Command-line interface
```

C. Infrastructure Starter Code: The `Token` class and constants from `tokens.py` are prerequisites. Ensure they are available.

```python
# tokens.py (Complete - from Lexer section)

from dataclasses import dataclass

from typing import final


# Token types

@final

class TokenType:

    PLUS = 'PLUS'

    MINUS = 'MINUS'

    STAR = 'STAR'

    SLASH = 'SLASH'

    CARET = 'CARET'

    EQUAL = 'EQUAL'

    LPAREN = 'LPAREN'

    RPAREN = 'RPAREN'

    NUMBER = 'NUMBER'

    IDENTIFIER = 'IDENTIFIER'

    EOF = 'EOF'


@dataclass

class Token:

    type: str

    lexeme: str

    position: int


    def __repr__(self):

        return f"Token({self.type}, '{self.lexeme}', pos={self.position})"
```

D. Core Logic Skeleton Code:

PYTHON

```python
# ast_nodes.py

from abc import ABC, abstractmethod

from dataclasses import dataclass

from typing import Optional


class ASTNode(ABC):

    """Abstract base class for all nodes in the Abstract Syntax Tree."""

    @abstractmethod

    def __repr__(self) -> str:

        pass


@dataclass

class Number(ASTNode):

    value: float

    def __repr__(self):

        return f"Number({self.value})"


@dataclass

class BinaryOp(ASTNode):

    operator: str

    left: ASTNode

    right: ASTNode

    def __repr__(self):

        return f"BinaryOp('{self.operator}', {self.left}, {self.right})"


@dataclass

class UnaryOp(ASTNode):

    operator: str

    operand: ASTNode

    def __repr__(self):

        return f"UnaryOp('{self.operator}', {self.operand})"


@dataclass

class Variable(ASTNode):

    name: str

    def __repr__(self):

        return f"Variable('{self.name}')"


@dataclass

class Assign(ASTNode):

    name: str

    value: ASTNode

    def __repr__(self):
```

```
        return f"Assign('{self.name}', {self.value})"

@dataclass
class FunctionCall(ASTNode):
    name: str
    argument: ASTNode
    def __repr__(self):
        return f"FunctionCall('{self.name}', {self.argument})"
```

PYTHON

```python
# parser.py

from typing import List

from tokens import Token, TokenType

from ast_nodes import (

    ASTNode, Number, BinaryOp, UnaryOp, Variable, Assign, FunctionCall

)


class Parser:

    def __init__(self, tokens: List[Token]):

        self.tokens = tokens

        self.current = 0  # Index of the next token to consume


    def parse(self) -> ASTNode:

        """Top-level parse method. Parses an expression or assignment."""

        # TODO 1: Call the method that starts parsing the highest-level grammar rule (e.g., parse_assignment or parse_expression).

        # TODO 2: After parsing, ensure all tokens have been consumed (except EOF). If not, raise a SyntaxError with the unexpected token's details.

        # TODO 3: Return the root AST node.

        pass


    # ------------------------------------------------------------------

    # Grammar rule methods (implement in order of increasing precedence)

    # ------------------------------------------------------------------

    def parse_assignment(self) -> ASTNode:

        """assignment → IDENTIFIER "=" expression | logical_or"""

        # TODO 1: First, parse a logical_or expression (for us, this will be parse_term). This becomes the potential left-hand side.

        # TODO 2: Check if the current token is EQUAL ('=') AND if the left-hand side node is a Variable.

        # TODO 3: If both conditions are true, consume the EQUAL token.

        # TODO 4: Recursively parse the right-hand side by calling parse_assignment (not parse_term, to allow chained assignments like a = b = 5).

        # TODO 5: Return an Assign node with the variable name and the right-hand side expression.

        # TODO 6: If the token is EQUAL but the left-hand side is not a Variable, raise a SyntaxError (e.g., "Cannot assign to non-variable").

        # TODO 7: If the token is not EQUAL, simply return the left-hand side node parsed in step 1.

        pass


    def parse_expression(self) -> ASTNode:

        """For simplicity, we can alias parse_assignment as the entry point, or have a separate expression rule."""

        return self.parse_term()  # In our grammar, the top-level expression is a term (after handling assignment).


    def parse_term(self) -> ASTNode:

        """term → factor ( ( "+" | "-" ) factor )*"""

        # TODO 1: Parse the first factor by calling parse_factor. This is the left operand.

        # TODO 2: Enter a while loop while the current token is PLUS or MINUS.
```

```
        # TODO 3: Inside the loop, save the operator (PLUS or MINUS), consume the token.
        # TODO 4: Parse the next factor by calling parse_factor. This is the right operand.
        # TODO 5: Create a new BinaryOp node with the saved operator, the left operand, and the right operand.
        # TODO 6: Set this new BinaryOp node as the left operand for the next iteration of the loop (to achieve left associativity).
        # TODO 7: After the loop ends, return the final left operand.
        pass

    def parse_factor(self) -> ASTNode:
        """factor → unary ( ( "*" | "/" ) unary )*"""
        # TODO 1: Parse the first unary by calling parse_unary. This is the left operand.
        # TODO 2: Enter a while loop while the current token is STAR or SLASH.
        # TODO 3: Inside the loop, save the operator (STAR or SLASH), consume the token.
        # TODO 4: Parse the next unary by calling parse_unary. This is the right operand.
        # TODO 5: Create a new BinaryOp node with the saved operator, the left operand, and the right operand.
        # TODO 6: Set this new BinaryOp node as the left operand for the next iteration.
        # TODO 7: After the loop ends, return the final left operand.
        pass

    def parse_unary(self) -> ASTNode:
        """unary → ( "-" | "+" ) unary | power"""
        # TODO 1: Check if the current token is PLUS or MINUS.
        # TODO 2: If it is, save the operator, consume the token.
        # TODO 3: Recursively call parse_unary (not parse_power) to parse the operand. This allows stacking unary operators (e.g., --5).
        # TODO 4: Return a UnaryOp node with the saved operator and the operand.
        # TODO 5: If the current token is not PLUS or MINUS, proceed to parse_power by calling parse_power.
        pass

    def parse_power(self) -> ASTNode:
        """power → primary ( "^" unary )*   # Right-associative variant"""
        # TODO 1: Parse the first primary by calling parse_primary. This is the base.
        # TODO 2: Check if the current token is CARET ('^').
        # TODO 3: If it is, consume the CARET token.
        # TODO 4: For RIGHT associativity, recursively call parse_unary (or parse_power) to parse the exponent.
        #         - Calling parse_unary ensures correct precedence relative to unary ops.
        #         - For strict right associativity among multiple '^', you could call parse_power here.
        # TODO 5: Create a BinaryOp node with operator '^', the base as left, and the exponent as right.
        # TODO 6: Return the BinaryOp node.
        # TODO 7: If there is no '^', simply return the primary node from step 1.
        pass

    def parse_primary(self) -> ASTNode:
        """primary → NUMBER | IDENTIFIER | "(" expression ")" | IDENTIFIER "(" expression ")" """
```

```
        token = self.current_token()
        # TODO 1: If token.type is NUMBER, consume it and return a Number node with float(token.lexeme).
        # TODO 2: If token.type is IDENTIFIER, consume it and store the name.
        # TODO 3: Look ahead (peek) at the next token. If it is LPAREN, then it's a function call.
        #         - Consume the LPAREN.
        #         - Parse the argument expression by calling parse_expression.
        #         - Consume the expected RPAREN (raise error if not found).
        #         - Return a FunctionCall node with the stored name and the argument.
        # TODO 4: If the next token is not LPAREN, return a Variable node with the stored name.
        # TODO 5: If token.type is LPAREN, consume it.
        #         - Parse the inner expression by calling parse_expression.
        #         - Consume the expected RPAREN (raise error if not found).
        #         - Return the inner expression node.
        # TODO 6: If none of the above, raise a SyntaxError like f"Unexpected token '{token.lexeme}' at position {token.position}"
        pass

    # ------------------------------------------------------------------
    # Helper methods for token consumption and lookahead
    # ------------------------------------------------------------------
    def current_token(self) -> Token:
        """Returns the token at the current position."""
        return self.tokens[self.current]

    def peek(self) -> Token:
        """Returns the next token without consuming it, or EOF if at end."""
        if self.current + 1 < len(self.tokens):
            return self.tokens[self.current + 1]
        return self.tokens[-1]  # EOF token

    def consume(self, expected_type: str, error_msg: str = None) -> Token:
        """Consumes the current token if it matches the expected type, otherwise raises SyntaxError."""
        token = self.current_token()
        if token.type == expected_type:
            self.current += 1
            return token
        # Provide a helpful error message
        if error_msg is None:
            error_msg = f"Expected token of type '{expected_type}', but got '{token.type}' ('{token.lexeme}') at position
{token.position}"
        raise SyntaxError(error_msg)

    def match(self, *types: str) -> bool:
        """Checks if the current token matches any of the given types. If yes, consumes it and returns True."""
```

```
        token = self.current_token()
        for t in types:
            if token.type == t:
                self.consume(t)
                return True
        return False

    def is_at_end(self) -> bool:
        """Checks if the current token is EOF."""
        return self.current_token().type == TokenType.EOF
```

E. Language-Specific Hints:

- **Python's `float` conversion**: Use `float(token.lexeme)` to convert number lexemes. Be aware that this will accept strings like `"3.14"` and `"-5"` but will raise a `ValueError` on malformed numbers. Since the lexer already validated the number format, this is safe.
- **Error Types**: Use Python's built-in `SyntaxError` exception for parse errors. It's recognizable and can be caught specifically.
- **Lookahead**: The `peek()` method is crucial for distinguishing between a variable and a function call without consuming tokens prematurely.
- **Recursion Limit**: Python has a recursion limit (default ~1000). Our expressions are unlikely to hit this, but deeply nested parentheses could. This is acceptable for the learning scope.

F. Milestone Checkpoint (Parser):

- **After Milestone 1**: Test with `parser.parse()` on tokenized input `"2 + 3 * (4 - 1)"`. Use a debugger or print the resulting AST. It should resemble a tree where `*` is higher than `+` and `-` is inside parentheses. A visual representation might look like `BinaryOp('+', Number(2), BinaryOp('*', Number(3), BinaryOp('-', Number(4), Number(1))))`.
- **After Milestone 2**: Test `"-2^2"` → Should yield `UnaryOp('-', BinaryOp('^', Number(2), Number(2)))`. Test `"2^3^2"` → Should yield a right-associative tree: `BinaryOp('^', Number(2), BinaryOp('^', Number(3), Number(2)))`.
- **After Milestone 3**: Test `"x = 5"` → `Assign('x', Number(5))`. Test `"sin(3.14/2)"` → `FunctionCall('sin', BinaryOp('/', Number(3.14), Number(2)))`.

---

7. Component Design: Evaluator

> **Milestone(s):** Milestone 1 (Basic Arithmetic), Milestone 2 (Unary and Power), Milestone 3 (Variables and Functions) — The evaluator is the final execution engine that brings the entire system together. While Milestone 1 establishes the basic recursive evaluation pattern for arithmetic, Milestone 2 extends it to handle unary operations and right-associative exponentiation, and Milestone 3 introduces stateful operations through variable assignment and function calls.

Mental Model: The Tree Walker

Imagine you're a warehouse worker tasked with assembling a complex shipment based on a **nested packing slip**. This slip isn't a simple list but a hierarchical tree of instructions: "Box A contains 3 units of Product X and Box B," where Box B's slip in turn says "contains 2 units of Product Y." To determine the total contents, you can't just read the slip from top to bottom. Instead, you must **open each box (node)** when you encounter it, recursively unpack its contents, perform any local calculations (like summing quantities within a box), and return the result back up to the enclosing box.

This is exactly how the **Evaluator** works. It receives an **Abstract Syntax Tree (AST)**—our hierarchical packing slip—where each node is either a terminal value (a number, a variable name) or an operation to perform (add, multiply, assign, call a function). The evaluator's job is to **walk this tree from the leaves upward**, resolving each operation by first resolving its sub-operations. This recursive "tree walk" transforms the structural representation into a single numeric result, exactly as the warehouse worker transforms nested packing instructions into a total item count.

Interface

The evaluator exposes a single primary method that serves as the entry point for computing results from ASTs. It requires access to an `Environment`—the symbol table that maintains variable state across evaluations (crucial for Milestone 3).

Method	Parameters	Returns	Description
`evaluate`	`node: ASTNode` (the root of the expression tree to evaluate) `env: Environment` (the symbol table for variable storage and lookup)	`float` (the computed numeric result)	The main public interface. Recursively traverses the AST, executing the operations represented by each node. For variable assignment nodes, it also updates the environment as a side effect. Raises appropriate runtime errors (e.g., `ZeroDivisionError`, `NameError`).

> **Design Insight:** The separation between the `Parser` (which builds the AST) and the `Evaluator` (which walks it) is a classic example of **separation of concerns**. This design allows us to parse an expression once and then evaluate it multiple times with different environments, or potentially transform, optimize, or serialize the AST independently of execution logic.

Internal Algorithm

The evaluator implements a **recursive, visitor-like procedure**. For each type of AST node, it follows a specific evaluation recipe. The algorithm is intrinsically recursive because the structure of the AST is recursive—a `BinaryOp` node has left and right children that are themselves `ASTNode`s, which must be evaluated before the binary operation can be performed.

The evaluation proceeds via a top-down, depth-first traversal:

1. **Base Case – `Number` Node:**
   - The node contains a literal numeric value (e.g., `5`, `3.14`).
   - **Action:** Simply return the `value` field as a Python `float`. This terminates the recursion for that branch.
2. **Base Case – `Variable` Node:**
   - The node contains an identifier name (e.g., `x`, `result`).
   - **Action:** Call `env.get(name)` to look up the current value associated with that name in the symbol table.
   - **Error Handling:** If `env.get()` raises a `NameError` (because the name is not defined), propagate it with a clear message (e.g., `"Undefined variable 'x'"`).
3. **Unary Operation – `UnaryOp` Node:**
   - The node has an `operator` (only `'-'` for unary minus in our calculator) and a single `operand` child node.
   - **Action:**
     1. Recursively evaluate the `operand` node by calling `evaluate(operand, env)`. This yields a numeric value.
     2. Apply the unary operator: For `'-'`, return the negative of the evaluated operand (`-value`).
   - **Note:** This correctly handles nested unary operators like `--5`. The parser produces a tree like `UnaryOp('-', UnaryOp('-', Number(5)))`. The evaluator will recursively evaluate the inner `UnaryOp`, then apply the outer negation.
4. **Binary Operation – `BinaryOp` Node:**
   - The node has an `operator` (`'+'`, `'-'`, `'*'`, `'/'`, `'^'`), a `left` child, and a `right` child.
   - **Action:**
     1. Recursively evaluate the `left` child node: `left_val = evaluate(left, env)`.
     2. Recursively evaluate the `right` child node: `right_val = evaluate(right, env)`.
     3. Apply the binary operator to `left_val` and `right_val`:
        - `'+'`: Return `left_val + right_val`.
        - `'-'`: Return `left_val - right_val`.
        - `'*'`: Return `left_val * right_val`.
        - `'/'`: Return `left_val / right_val`. **Critical:** Check if `right_val == 0` and raise a `ZeroDivisionError` with a descriptive message.
        - `'^'` (exponentiation): Return `left_val ** right_val` (using Python's `**` operator which handles both integer and float exponents).
   - **Key Point:** The order of evaluation (left subtree first, then right) and the operator implementation **enforce the precedence and associativity rules** that were encoded into the tree structure by the parser. The evaluator simply follows the tree's hierarchy.
5. **Assignment – `Assign` Node:**
   - The node has a `name` (the variable identifier) and a `value` child node (the expression to evaluate and assign).
   - **Action:**
     1. Recursively evaluate the `value` node: `val = evaluate(value, env)`.
     2. Call `env.set(name, val)` to store the computed value in the environment, associating it with the given name.
     3. **Return the assigned value.** This makes assignment an expression that evaluates to the value assigned (e.g., `x = 5` evaluates to `5`), allowing chained assignments like `y = x = 5`.

- **Design Note:** Assignment is a side effect—it modifies the environment. The evaluator returns the value to allow the assignment to be used within a larger expression, which is a common language design choice.

6. **Function Call – `FunctionCall` Node:**
   - The node has a `name` (the function identifier, e.g., `sqrt`) and an `argument` child node (the expression to pass as the function's argument).
   - **Action:**
     1. Recursively evaluate the `argument` node: `arg_val = evaluate(argument, env)`.
     2. Look up the function implementation by name in a predefined **built-in function table**. For our calculator, this table maps `'sin'` to `math.sin`, `'cos'` to `math.cos`, `'sqrt'` to `math.sqrt`, and `'abs'` to `abs`.
     3. Call the built-in function with `arg_val` as the argument. For `sqrt`, ensure `arg_val >= 0` and raise a `ValueError` if negative.
     4. Return the function's result as a float.
   - **Error Handling:** If the function name is not in the built-in table, raise a `NameError` (e.g., `"Unknown function 'foo'"`).

The algorithm's control flow is elegantly captured by a series of `if isinstance(node, ...)` checks (or a visitor pattern) that dispatches to the appropriate logic for each node type. The recursion naturally follows the tree's depth-first structure.

ADR: Evaluation Strategy

> **Decision: Recursive Tree Walk Interpreter**
>
> - **Context:** We need to execute the parsed expression tree to produce a numeric result. The AST is a relatively simple, in-memory tree structure representing the expression. The primary audience is learners, and the primary goals are clarity, educational value, and direct mapping to the parsing concepts.
> - **Options Considered:**
>   1. **Recursive Tree Walk (Interpreter):** Implement a recursive function that traverses the AST node-by-node, evaluating subexpressions and applying operations directly.
>   2. **Bytecode Compilation & Virtual Machine:** Translate the AST into a sequence of simple instructions (bytecode) for a stack-based or register-based virtual machine, then execute that bytecode.
>   3. **Direct Translation to Host Language:** Transform the AST into an equivalent expression in the host language (e.g., Python `eval` of a constructed string or code object) and execute it.
> - **Decision:** We chose Option 1, the recursive tree walk interpreter.
> - **Rationale:**
>   - **Clarity and Pedagogical Value:** The recursive tree walk directly mirrors the recursive structure of the AST and the grammar. Each step in evaluation corresponds transparently to a node in the tree, making it easy for learners to trace execution and understand the connection between parsing and evaluation.
>   - **Simplicity:** Implementation is straightforward—essentially a large match statement on node types with recursive calls. It requires no intermediate representation (bytecode) or complex code generation.
>   - **Sufficient Performance:** For a calculator evaluating single expressions, the overhead of tree walking is negligible. The simplicity benefit far outweighs any micro-optimization.
>   - **Safety and Control:** Unlike direct translation to host language execution (Option 3), the tree walk gives us full control over evaluation. We can implement custom error handling, precisely control side effects (variable assignment), and ensure operations behave exactly as defined (e.g., our own exponentiation semantics).
> - **Consequences:**
>   - **Positive:** The implementation is easy to write, read, debug, and extend. Adding a new node type (e.g., for a modulo operator) requires only adding a new branch to the evaluator.
>   - **Negative:** For extremely deep expression trees (thousands of nodes), Python's recursion limit could be hit, though this is improbable for calculator expressions. Performance is slightly slower than bytecode execution for repeated evaluations of the same expression, as the tree must be traversed each time.

Option	Pros	Cons	Why Not Chosen?
Recursive Tree Walk	Intuitive, maps directly to AST structure, easy to implement and debug, full control over semantics	Slightly slower for repeated evaluation, potential recursion depth limits for huge expressions	**CHOSEN** — Best for learning and adequate for the problem scope.
Bytecode Compilation	Faster for repeated evaluation (compile once, run many times), more realistic model for production languages	Significant additional complexity (design bytecode, compiler, VM), obscures the direct link between parsing and execution	Overkill for a beginner-focused calculator; adds conceptual overhead without proportional educational benefit.
Direct Host Execution	Very fast, leverages host language's optimized evaluator	Dangerous (security risk if input is untrusted), loses control over semantics (e.g., Python's `**` has right associativity, but what if we wanted different?), hard to integrate with custom environment for variables	Security and control are paramount; we want to define our own exact evaluation rules, not delegate to Python.

Common Pitfalls

⚠️ Pitfall: Incorrect Order of Child Evaluation in Binary Operators

- **Description:** Evaluating the right child before the left child, or evaluating one child multiple times. While for pure arithmetic the order might not affect the result due to commutativity of `+` and `*`, it becomes critical for:
  1. **Side effects:** In `x = 5 + (y = 3)`, the assignment to `y` must happen before the addition.
  2. **Function calls with side effects** (if extended).
  3. **Correct error reporting** (which subexpression failed first?).
- **Why It's Wrong:** Violates the expected left-to-right evaluation order common in most languages and can lead to incorrect results if operations have side effects or errors.
- **Fix:** Always evaluate `left` completely, then evaluate `right`. Store their results in local variables before applying the operator.

⚠️ Pitfall: Not Handling Division by Zero at Evaluation Time

- **Description:** Simply performing `left_val / right_val` in Python will raise a `ZeroDivisionError`, but the error message will be generic and not contextualized to the calculator expression.
- **Why It's Wrong:** While Python's exception is okay, a better user experience is to catch this exception and re-raise with a clearer message (e.g., `"Division by zero in expression '5 / 0'"`). More importantly, we must ensure the evaluator doesn't crash in an uncontrolled way.
- **Fix:** Explicitly check if `right_val == 0` (using a tolerance for floating-point if desired) before division and raise a custom `ZeroDivisionError` with a helpful message.

⚠️ Pitfall: Confusing Variable Assignment with Variable Lookup in Expression Evaluation

- **Description:** When evaluating an `Assign` node, incorrectly returning `None` or not updating the environment, or when evaluating a `Variable` node, accidentally assigning to it instead of looking it up.
- **Why It's Wrong:** Assignment must modify the environment state and return the assigned value. Lookup must only read, not write. Mixing these breaks variable semantics.
- **Fix:** Clearly separate the logic: `Assign.evaluate` calls `env.set()` and returns the value; `Variable.evaluate` calls `env.get()` and returns the retrieved value.

⚠️ Pitfall: Floating-Precision Issues in Exponentiation

- **Description:** Using the built-in `pow()` or `**` for exponentiation can produce tiny floating-point errors (e.g., `4 ** 0.5` might yield `1.9999999999999998` instead of `2`). Also, raising negative numbers to fractional powers (e.g., `(-2) ** 0.5`) results in complex numbers, which are outside our calculator's scope.
- **Why It's Wrong:** Learners might be confused by imprecise results. Complex results would require a number type we don't support.
- **Fix:** For integer exponents, consider special-casing to use multiplication loops for perfect precision with integer bases. For fractional exponents, accept the floating-point approximation but document it. Consider validating that for `sqrt(arg)`, `arg >= 0`, and for `a ** b` with fractional `b`, require `a >= 0`.

⚠️ Pitfall: Forgetting to Propagate Environment Through Recursive Calls

- **Description:** The evaluator's `evaluate` method takes an `env` parameter, but when making recursive calls for child nodes, accidentally passing a different environment or `None`.
- **Why It's Wrong:** Child nodes need access to the same symbol table to resolve variables and update assignments. Passing a different environment would create separate scopes, breaking variable sharing.
- **Fix:** Always pass the same `env` reference in every recursive call to `evaluate`.

⚠️ Pitfall: Infinite Recursion on Cyclic AST (Theoretical)

- **Description:** While our parser cannot produce a cyclic AST, a bug could hypothetically create one (e.g., a `Variable` node that refers to itself via the environment in some convoluted way). The evaluator would then recurse indefinitely.

- **Why It's Wrong:** Program hangs or crashes due to recursion depth overflow.
- **Fix:** Not a primary concern for this project, but in a more advanced setting, one could implement cycle detection by tracking visited nodes during evaluation.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Evaluation Engine	Recursive tree walk with `isinstance` checks	Visitor pattern using Python's `singledispatch` or a NodeVisitor base class
Math Functions	Python's built-in `math` module (`math.sin` , `math.sqrt` , etc.)	Custom implementations (e.g., Taylor series) for educational purposes
Error Handling	Python's built-in exceptions (`ZeroDivisionError` , `ValueError` , `NameError`)	Custom exception hierarchy with detailed error context

B. Recommended File/Module Structure

Add the evaluator to the existing project structure:

```
calculator/
├── __init__.py
├── lexer.py          # From Milestone 1
├── parser.py         # From Milestone 1 & 2
├── ast_nodes.py      # From Milestone 1 (contains AST node classes)
├── evaluator.py      # NEW: This file contains the Evaluator class and built-in function map
├── environment.py    # NEW: Contains the Environment class for variable storage
└── main.py           # Command-line interface or REPL
```

C. Infrastructure Starter Code (COMPLETE)

File: `environment.py` – A complete, ready-to-use symbol table.

PYTHON

```
"""
Environment: Symbol table for variable storage and lookup.
"""
class Environment:
    """Maps variable names to numeric values."""

    def __init__(self, parent=None):
        """
        Initialize a new environment.

        Args:
            parent: Optional parent environment for nested scopes (not used in
                    base milestones but useful for extension).
        """
        self.variables = {}
        self.parent = parent

    def get(self, name: str) -> float:
        """
        Retrieve the value of a variable.

        Args:
            name: The variable identifier.

        Returns:
            The numeric value associated with the name.

        Raises:
            NameError: If the variable is not defined in this environment or any parent.
        """
        if name in self.variables:
            return self.variables[name]
        elif self.parent is not None:
            return self.parent.get(name)
        else:
            raise NameError(f"Undefined variable '{name}'")

    def set(self, name: str, value: float) -> None:
        """
        Assign a value to a variable, creating it if it doesn't exist.
```

```
        Args:
            name: The variable identifier.
            value: The numeric value to assign.
        """
        self.variables[name] = value
```

D. Core Logic Skeleton Code (signature + TODOs only)

File: `evaluator.py` – Skeleton for the core evaluator logic.

PYTHON

```python
"""
Evaluator: Recursively evaluates an AST to produce a numeric result.
"""
import math
from typing import Any

# Import AST node classes (adjust import path as needed)
from .ast_nodes import (
    ASTNode, Number, BinaryOp, UnaryOp, Assign, Variable, FunctionCall
)
from .environment import Environment

class Evaluator:
    """Evaluates AST nodes recursively."""

    # Built-in function table: maps names to Python callables
    BUILTINS = {
        'sin': math.sin,
        'cos': math.cos,
        'tan': math.tan,
        'sqrt': math.sqrt,
        'abs': abs,
        'log': math.log,   # Optional extension
        'exp': math.exp,   # Optional extension
    }

    @classmethod
    def evaluate(cls, node: ASTNode, env: Environment) -> float:
        """
        Public entry point: evaluate any AST node.

        Args:
            node: The root of the AST to evaluate.
            env: The environment for variable lookup and assignment.

        Returns:
            The numeric result of evaluating the expression.

        Raises:
            ZeroDivisionError: For division by zero.
            NameError: For undefined variables or functions.
```

```
            ValueError: For invalid arguments to functions (e.g., sqrt(-1)).
        """
        # TODO 1: If node is an instance of Number, return its value field.
        # TODO 2: If node is an instance of Variable, call env.get(node.name) and return the result.
        # TODO 3: If node is an instance of UnaryOp:
        #   a) Recursively evaluate node.operand -> operand_value.
        #   b) If node.operator == '-', return -operand_value.
        #   (Optionally handle unary '+' if your parser produces it.)
        # TODO 4: If node is an instance of BinaryOp:
        #   a) Recursively evaluate node.left -> left_val.
        #   b) Recursively evaluate node.right -> right_val.
        #   c) Based on node.operator:
        #      - '+' : return left_val + right_val
        #      - '-' : return left_val - right_val
        #      - '*' : return left_val * right_val
        #      - '/' :
        #           i) Check if right_val == 0 (or abs(right_val) < 1e-12 for float safety).
        #           ii) If zero, raise ZeroDivisionError with message.
        #           iii) Else return left_val / right_val
        #      - '^' : return left_val ** right_val
        # TODO 5: If node is an instance of Assign:
        #   a) Recursively evaluate node.value -> value_result.
        #   b) Call env.set(node.name, value_result) to store the value.
        #   c) Return value_result (so assignment expression evaluates to the assigned value).
        # TODO 6: If node is an instance of FunctionCall:
        #   a) Recursively evaluate node.argument -> arg_value.
        #   b) Look up node.name in cls.BUILTINS.
        #   c) If not found, raise NameError(f"Unknown function '{node.name}'").
        #   d) Call the function with arg_value as argument.
        #   e) For 'sqrt', check if arg_value < 0 and raise ValueError if so.
        #   f) Return the result as float.
        # TODO 7: If node type is not recognized, raise TypeError(f"Unknown node type: {type(node)}").
        pass  # Remove this line after implementing
```

E. Language-Specific Hints

- **Float Comparison:** When checking for division by zero, consider using `abs(right_val) < 1e-12` instead of `right_val == 0` to handle floating-point rounding errors from previous calculations.
- **Math Functions:** The `math` module expects angles in radians for `sin`, `cos`. If you want to support degrees, you'd need to convert: `math.sin(math.radians(degrees))`.
- **Exponentiation with `**`:** Python's `**` operator correctly handles right associativity and fractional exponents. For integer exponents with integer bases, you could use `pow(base, exp)` for integer results, but `**` is fine for our general case.
- **Error Messages:** Use f-strings to include the problematic variable/function name in error messages (e.g., `NameError(f"Undefined variable '{name}'")`).

F. Milestone Checkpoint

After implementing the evaluator for each milestone, verify with these expressions:

Milestone 1 (Basic Arithmetic):

- Run `evaluate(parse(tokenize("2 + 3 * 4")), Environment())` → Should return `14.0`, not `20.0`.
- Run `evaluate(parse(tokenize("(2 + 3) * 4")), Environment())` → Should return `20.0`.
- Run `evaluate(parse(tokenize("10 / 2")), Environment())` → Should return `5.0`.
- Run `evaluate(parse(tokenize("10 / 0")), Environment())` → Should raise `ZeroDivisionError`.

Milestone 2 (Unary and Power):

- Run `evaluate(parse(tokenize("-5 + 3")), Environment())` → Should return `-2.0`.
- Run `evaluate(parse(tokenize("--5")), Environment())` → Should return `5.0`.
- Run `evaluate(parse(tokenize("2 ^ 3 ^ 2")), Environment())` → Should return `512.0` (right-associative: 2^(3^2)).
- Run `evaluate(parse(tokenize("-2 ^ 2")), Environment())` → Should return `-4.0` (unary binds tighter: -(2^2)).

Milestone 3 (Variables and Functions):

```python
env = Environment()

# Assignment returns value

result = evaluate(parse(tokenize("x = 5")), env)  # Should return 5.0

print(result, env.get('x'))  # Both should be 5.0

# Use variable in expression

result = evaluate(parse(tokenize("x + 2")), env)  # Should return 7.0

# Function call

result = evaluate(parse(tokenize("sqrt(16)")), env)  # Should return 4.0

# Undefined variable

try:

    evaluate(parse(tokenize("y + 1")), env)

except NameError as e:

    print(e)  # Should print "Undefined variable 'y'"
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`2+3*4` evaluates to `20.0` (not `14.0`)	Precedence not enforced in AST construction (parser issue) or evaluator processes operators left-to-right	Print the AST. It should show `BinaryOp('+', Number(2), BinaryOp('*', Number(3), Number(4)))`. If it's flattened, parser is wrong.	Debug parser precedence levels (`parse_term`, `parse_factor`).
`--5` raises error or evaluates incorrectly	Unary operator parsing or evaluation chain broken	Print AST for `--5`. Should be `UnaryOp('-', UnaryOp('-', Number(5)))`. If not, parser's `parse_unary` is faulty. If AST is correct, evaluator's `UnaryOp` case isn't recursive.	Ensure `evaluate` recursively calls itself for the operand in `UnaryOp`.
Variable assignment works but subsequent lookup fails	Environment not being passed correctly to recursive evaluation, or `env.set` not being called	Add debug prints in `evaluate` for `Assign` and `Variable` nodes to show environment state.	Ensure the same `env` object is passed to all recursive `evaluate` calls.
`sqrt(-1)` doesn't raise an error	Missing argument validation in function call evaluation	Check the `FunctionCall` evaluation step: it should validate `arg_value >= 0` for `sqrt`.	Add explicit check: `if node.name == 'sqrt' and arg_value < 0: raise ValueError("sqrt of negative number")`.
`2 ^ 3 ^ 2` evaluates to `64.0` (not `512.0`)	Right associativity not implemented in parser for power operator	AST should be right-associative: `BinaryOp('^', Number(2), BinaryOp('^', Number(3), Number(2)))`. If it's left-associative, parser's `parse_power` is wrong.	In `parse_power`, after parsing left primary, if next token is `'^'`, recursively call `parse_power` for the right operand (not `parse_primary`).
Division by zero crashes with generic Python error	No explicit check in evaluator	The error trace will point to the line with `left_val / right_val`.	Add explicit zero check before division and raise a descriptive `ZeroDivisionError`.

8. Interactions and Data Flow

> **Milestone(s):** All Milestones (1-3) — This section traces the complete data transformation pipeline from raw input string to final computed result, demonstrating how all previously described components work together. The walkthrough specifically showcases variable assignment (Milestone 3), operator precedence (Milestone 1), and multiplication before addition (Milestone 1), providing a holistic view of the system's operation.

Understanding how data flows through a system is like tracing a package through a shipping network: the package (your mathematical expression) passes through multiple sorting facilities (components), each performing specialized transformations, until it reaches its final destination (the computed result). This end-to-end perspective reveals how the **separation of concerns** between lexing, parsing, and evaluation creates a clean, maintainable architecture where each component focuses on one specific transformation.

End-to-End Walkthrough: "x = 2 + 3 * 4"

Let's trace the complete journey of the expression `"x = 2 + 3 * 4"` through our three-stage pipeline. This example intentionally includes multiple concepts: **variable assignment**, **operator precedence** (multiplication before addition), and **left-associative** binary operators. We'll examine the state of the data at each transformation point, showing exactly how the string `"x = 2 + 3 * 4"` ultimately yields the numeric result `14.0` and stores the value `14.0` in variable `x`.

Phase 1: Lexical Analysis — From Characters to Tokens

Mental Model: Imagine a librarian scanning a book's table of contents and creating an index card for each chapter title and page number. The lexer performs a similar task: it scans the raw character sequence, identifies meaningful "words" and "punctuation" (tokens), discards irrelevant whitespace, and produces an organized index (token list) for the next stage.

The lexer receives the input string `"x = 2 + 3 * 4"` and processes it left-to-right using **greedy scanning** (consuming as many characters as possible for each token). Here is the complete tokenization process:

Input Character	Action Taken	Token Produced	Token Details
`'x'`	Start identifier, continue	-	Building `lexeme = "x"`
`' '` (space)	End identifier (whitespace)	`Token(type=IDENTIFIER, lexeme="x", position=0)`	Variable name `x`
`' '`	Skip whitespace	-	No token produced
`'='`	Single-character operator	`Token(type=EQUAL, lexeme="=", position=2)`	Assignment operator
`' '`	Skip whitespace	-	No token produced
`'2'`	Start number, continue	-	Building `lexeme = "2"`
`' '` (space)	End number (whitespace)	`Token(type=NUMBER, lexeme="2", position=4)`	Numeric value `2`
`' '`	Skip whitespace	-	No token produced
`'+'`	Single-character operator	`Token(type=PLUS, lexeme="+", position=6)`	Addition operator
`' '`	Skip whitespace	-	No token produced
`'3'`	Start number, continue	-	Building `lexeme = "3"`
`' '` (space)	End number (whitespace)	`Token(type=NUMBER, lexeme="3", position=8)`	Numeric value `3`
`' '`	Skip whitespace	-	No token produced
`'*'`	Single-character operator	`Token(type=STAR, lexeme="*", position=10)`	Multiplication operator
`' '`	Skip whitespace	-	No token produced
`'4'`	Start number, continue	-	Building `lexeme = "4"`
End of string	End number (EOF)	`Token(type=NUMBER, lexeme="4", position=12)`	Numeric value `4`
End of string	Produce EOF marker	`Token(type=EOF, lexeme="", position=13)`	Signals end of input

The lexer outputs the following token sequence (shown as a Python list for clarity):

```python
[
    Token(type='IDENTIFIER', lexeme='x', position=0),
    Token(type='EQUAL', lexeme='=', position=2),
    Token(type='NUMBER', lexeme='2', position=4),
    Token(type='PLUS', lexeme='+', position=6),
    Token(type='NUMBER', lexeme='3', position=8),
    Token(type='STAR', lexeme='*', position=10),
    Token(type='NUMBER', lexeme='4', position=12),
    Token(type='EOF', lexeme='', position=13)
]
```

Notice that all whitespace has been eliminated—the tokens contain only the meaningful lexical elements. The `position` field in each token preserves the original character index, which will be invaluable for producing meaningful error messages if parsing fails.

Phase 2: Syntactic Analysis — From Tokens to Abstract Syntax Tree

Mental Model: Picture a traffic controller at a complex intersection who directs vehicles into different lanes based on their type and priority (emergency vehicles first, then buses, then cars). The parser acts as this controller: it examines the stream of tokens and directs them into the appropriate hierarchical structure (the AST) based on **operator precedence** and **associativity** rules.

The parser receives the token list and begins its recursive descent parsing. The top-level method `Parser.parse()` calls `Parser.parse_assignment()`, which handles variable assignments. Let's trace the call sequence:

1. **`parse_assignment()`** — Checks if the first token is an `IDENTIFIER` followed by `EQUAL`.
    - Current token: `IDENTIFIER 'x'` → matches identifier pattern
    - `consume(IDENTIFIER)` consumes `'x'`, stores name `'x'`

- `match(EQUAL)` returns `True` (next token is `EQUAL`) → this is an assignment
- `consume(EQUAL)` consumes `'='` token
- Calls `parse_expression()` to parse the right-hand side `2 + 3 * 4`

2. **`parse_expression()`** — Parses addition and subtraction (lowest precedence).
   - Calls `parse_term()` to parse `2 + 3 * 4`
3. **`parse_term()`** — Parses multiplication and division (medium precedence).
   - Calls `parse_factor()` to parse `2`
     - `parse_factor()` calls `parse_unary()` which calls `parse_primary()`
     - `parse_primary()` sees `NUMBER '2'` → creates `Number(value=2.0)`
   - Result: `left = Number(value=2.0)`
   - While next token is `PLUS`, `MINUS`, `STAR`, or `SLASH`:
     - Next token is `PLUS` (addition) → store operator `'+'`
     - `consume(PLUS)` consumes `'+'` token
     - Calls `parse_factor()` to parse `3 * 4`:
       - `parse_factor()` calls `parse_unary()` which calls `parse_primary()`
       - `parse_primary()` sees `NUMBER '3'` → creates `Number(value=3.0)`
       - Back in `parse_term()`: while next token is `STAR`:
         - Operator `'*'` stored, `consume(STAR)`
         - Calls `parse_factor()` for `4` → `Number(value=4.0)`
         - Creates `BinaryOp(operator='*', left=Number(3.0), right=Number(4.0))`
     - Creates `BinaryOp(operator='+', left=Number(2.0), right=BinaryOp('*', Number(3.0), Number(4.0)))`
   - Returns this binary operation tree
4. **Back in `parse_assignment()`** — Now has:
   - `name = 'x'`
   - `value = BinaryOp('+', Number(2.0), BinaryOp('*', Number(3.0), Number(4.0)))`
   - Creates `Assign(name='x', value=the_binary_op_tree)`

The resulting **Abstract Syntax Tree (AST)** is a hierarchical structure that perfectly encodes the precedence: multiplication (`3 * 4`) is nested deeper than addition (`2 + ...`), ensuring it will be evaluated first. The assignment node sits at the root:

```
Assign(
    name='x',
    value=BinaryOp(
        operator='+',
        left=Number(value=2.0),
        right=BinaryOp(
            operator='*',
            left=Number(value=3.0),
            right=Number(value=4.0)
        )
    )
)
```

This tree structure is the **intermediate representation (IR)** that decouples parsing from evaluation. The parser's job is complete: it has transformed a linear token sequence into a structured tree that explicitly represents operator precedence through nesting.

Phase 3: Evaluation — From AST to Numeric Result

Mental Model: Imagine a warehouse worker following a nested packing slip. The outer box says "Ship to Customer X," and inside is a list of items to pack: "Item A plus the contents of Box B." Box B itself contains instructions: "Multiply Item C by Item D." The worker must open each box, follow its instructions, and combine results according to the outer box's operation. The evaluator performs exactly this **tree walk**, recursively opening each AST node to compute its value.

The evaluator receives the root `Assign` node and an initially empty `Environment` (symbol table). The `Evaluator.evaluate()` method begins its recursive traversal:

1. **Evaluate `Assign` node**:
   - Calls `evaluate()` on its `value` child (the `BinaryOp('+', ...)` node)
2. **Evaluate `BinaryOp('+', ...)` node**:
   - First evaluates its `left` child: `Number(2.0)` → returns `2.0`
   - Then evaluates its `right` child: `BinaryOp('*', ...)` node

3. **Evaluate `BinaryOp('*', ...)` node**:
    - Evaluates `left` child: `Number(3.0)` → returns `3.0`
    - Evaluates `right` child: `Number(4.0)` → returns `4.0`
    - Applies operation: `3.0 * 4.0` = `12.0`
    - Returns `12.0` to parent
4. **Back in `BinaryOp('+', ...)` evaluation**:
    - Now has `left_result = 2.0`, `right_result = 12.0`
    - Applies operation: `2.0 + 12.0` = `14.0`
    - Returns `14.0` to parent
5. **Back in `Assign` evaluation**:
    - Has `name = 'x'` and `computed_value = 14.0`
    - Calls `Environment.set('x', 14.0)` to store the value
    - **Side effect**: The environment's `variables` dictionary now contains `{'x': 14.0}`
    - Returns `14.0` as the result of the entire expression

The evaluation sequence demonstrates the power of the AST: the tree structure naturally enforces the correct order of operations. The evaluator doesn't need to know about operator precedence rules—it simply follows the tree's nested structure, which the parser already arranged according to those rules.

The sequence diagram above illustrates the recursive evaluation process for a binary operation. Notice how the evaluator calls itself recursively for child nodes before combining their results—this depth-first traversal matches the nested structure of the AST.

Final State Summary

After processing the expression `"x = 2 + 3 * 4"`, the system reaches the following final state:

Component	Final State	Description
Input String	`"x = 2 + 3 * 4"`	Original expression (unchanged)
Token Sequence	8 tokens (including EOF)	Lexical units produced by lexer
Abstract Syntax Tree	`Assign('x', BinaryOp('+', Number(2), BinaryOp('*', Number(3), Number(4))))`	Hierarchical representation of expression structure
Environment	`{'x': 14.0}`	Symbol table with one variable binding
Final Result	`14.0`	Numeric value returned by evaluator

This walkthrough illustrates several key architectural principles:

> **Design Insight:** The pipeline's **unidirectional data flow** (string → tokens → AST → number) creates a clean separation of concerns. Each component transforms the data into a form that makes the next component's job simpler: the lexer eliminates whitespace and categorizes characters, the parser resolves precedence ambiguities through tree structure, and the evaluator simply follows the tree without worrying about parsing rules.

> **Design Insight:** The **AST serves as a contract** between the parser and evaluator. As long as both components agree on the AST node types and their meanings, they can evolve independently. For example, you could add a new operator by modifying only the parser and evaluator—the lexer might not need changes if the operator uses an existing token type.

Data Flow for Subsequent Expressions

Now consider what happens when a subsequent expression references the variable we just assigned. If the user inputs `"x + 1"`:

1. **Lexer** produces: `[IDENTIFIER('x'), PLUS('+'), NUMBER('1'), EOF]`
2. **Parser** produces: `BinaryOp('+', Variable('x'), Number(1.0))`
3. **Evaluator**:
   - Evaluates `Variable('x')` → calls `Environment.get('x')` → returns `14.0`
   - Evaluates `Number(1.0)` → returns `1.0`
   - Applies addition: `14.0 + 1.0` = `15.0`
4. **Result**: `15.0` (environment remains `{'x': 14.0}` unchanged)

This demonstrates the **persistent state** of the `Environment` across evaluations. The environment acts as a shared memory space that outlives any single expression evaluation, enabling variables to store values for later use.

Common Pitfalls in Data Flow Integration

⚠️ Pitfall: Forgetting to Propagate Position Information

- **Description:** When a parser or evaluator encounters an error, it reports a generic message without pointing to the problematic location in the original input.
- **Why It's Wrong:** Users cannot easily locate and fix syntax errors in longer expressions.
- **Fix:** Always propagate token `position` fields through to AST nodes (add a `position` field to each AST node class). When reporting errors, include the position: `"Syntax error at character 8: unexpected token '*'"`.

⚠️ Pitfall: Evaluating Assignment as Expression Without Side Effect

- **Description:** The evaluator returns the assigned value for an `Assign` node but forgets to call `Environment.set()` to actually store the value.
- **Why It's Wrong:** Subsequent references to the variable will fail with "undefined variable" errors, even though the assignment appeared to succeed.
- **Fix:** Ensure `Assign` evaluation has two effects: (1) modify environment, (2) return the assigned value. The modification must happen before returning.

⚠️ Pitfall: Ignoring Evaluation Order Within Binary Operations

- **Description:** When evaluating `BinaryOp`, the evaluator computes `left` and `right` operands in an undefined order or in the wrong order for side effects.
- **Why It's Wrong:** While mathematically pure arithmetic has no side effects, once functions or assignments are introduced (like `f(x) + g(x)` where functions modify global state), evaluation order matters. Standard mathematical convention is left-to-right.
- **Fix:** Always evaluate `left` operand completely before evaluating `right` operand. Document this as a guarantee in the evaluator's specification.

⚠️ Pitfall: Not Handling EOF in Parser Lookahead

- **Description:** The parser's `peek()` method tries to access `tokens[current_position + 1]` without checking if it's within bounds.
- **Why It's Wrong:** When parsing incomplete expressions (like `"2 + "`), the parser may crash with an `IndexError` rather than reporting a proper syntax error.
- **Fix:** Implement `peek()` to return an `EOF` token when at the end of the token list. Ensure `is_at_end()` checks `current_token().type == EOF`.

The data flow walkthrough reveals the elegance of the three-stage pipeline: each component has a single responsibility, communicates through well-defined data structures, and together they transform a human-readable expression into a computed result while correctly handling operator precedence, variable storage, and mathematical evaluation.

9. Error Handling and Edge Cases

> **Milestone(s):** All Milestones (1-3) — Proper error handling becomes increasingly important across milestones. Milestone 1 requires detecting syntax errors in basic expressions. Milestone 2 adds unary operator edge cases. Milestone 3 introduces semantic errors with undefined variables and function misuse. This section provides a comprehensive framework for making the calculator robust and user-friendly.

Mental Model: The Calculator as a Proofreader

Imagine you're a proofreader reviewing a mathematical manuscript. Your job isn't just to calculate results but to ensure the text follows proper mathematical grammar. When you encounter an issue, you don't simply say "there's an error"—you provide specific feedback: *"On line 3, the closing parenthesis is missing after 'sin(2+3'."* This section transforms our calculator from a brittle calculator that crashes on bad input to a helpful assistant that clearly explains what went wrong and where.

Our calculator faces four distinct classes of errors, each caught at different stages of processing. Understanding this layered defense helps us build a system that fails gracefully and provides actionable feedback.

9.1 Error Categories and Detection

Errors in the calculator pipeline fall into four distinct categories, each detected by a different component. The table below summarizes this defense-in-depth approach:

Error Category	Detection Component	Typical Causes	Examples
Lexical Errors	`Lexer` during tokenization	Invalid characters in input	`2 @ 3`, `7.8.9`, `$var`
Syntactic Errors	`Parser` during tree construction	Malformed expression structure	`2 + * 3`, `(2 + 3`, `sin 2)`
Semantic Errors	`Evaluator` during tree traversal	Meaning violations, undefined symbols	`x + 2` (x undefined), `sqrt(-1)`
Runtime Errors	`Evaluator` during computation	Numeric domain/range violations	`1 / 0`, `2 ^ 1000000` (overflow)

Lexical Errors: Invalid Building Blocks

The `Lexer`'s job is to break the input string into valid tokens. When it encounters a character or sequence that doesn't match any known token pattern, it must raise a `LexerError`. These are the earliest detectable errors—if the input contains characters that don't belong to the calculator's alphabet, we know immediately something is wrong.

Common Lexical Error Scenarios:

1. **Unknown Characters:** Mathematical symbols not in our operator set (`@`, `&`, `#`)
2. **Malformed Numbers:** Multiple decimal points (`12.34.56`), trailing decimal points without digits (`42.`)
3. **Invalid Identifier Characters:** Identifiers starting with numbers (`2var`) or containing special characters (`var-name`)

> **Key Insight:** The lexer uses **greedy scanning**—it consumes as many valid characters as possible for the current token type. This means `12.34.56` is initially scanned as a valid `NUMBER` token `"12.34"`, then the second `.` triggers a lexical error because a decimal point can't follow a completed number without an intervening operator.

The lexer should track the current position (character index) in the input string so it can report *where* the error occurred. For example, for input `"2 @ 3"`, the error message should indicate position 2 (0-based) or character 3 (1-based), not just say "invalid character."

Syntactic Errors: Grammar Violations

Once we have valid tokens, the `Parser` must arrange them according to the grammar rules. A **syntax error** occurs when the token sequence doesn't conform to any valid expression structure. The parser's recursive descent algorithm naturally detects these errors when it expects a particular token type but encounters something else.

Common Syntactic Error Patterns:

1. **Missing Operands:** `2 + * 3` (binary operator without right operand)
2. **Missing Operators:** `2 3` (two numbers adjacent without operator)
3. **Mismatched Parentheses:** `(2 + 3` or `2 + 3)`
4. **Incomplete Function Calls:** `sin(` or `sin 2)` (missing parenthesis)
5. **Malformed Assignment:** `= 5` (missing left-hand identifier) or `x =` (missing expression)

The parser's `consume(expected_type, error_msg)` method is the primary syntax error detection mechanism. When it expects, say, an `RPAREN` after parsing a parenthesized expression but finds `PLUS` instead, it raises a `SyntaxError` with a descriptive message.

Architecture Decision: Immediate Failure vs. Error Recovery

Context: When the parser encounters a syntax error, it could either stop immediately or attempt to recover and continue parsing to find additional errors. This calculator is a learning tool with simple expressions, not a production compiler.

Options Considered:

1. **Immediate Failure:** Raise exception at first error, stop parsing.
2. **Panic-Mode Recovery:** Skip tokens until a known synchronization point (like semicolon in other languages), continue to find more errors.
3. **Error Productions:** Add grammar rules that match erroneous constructs to provide better error messages.

Decision: Immediate failure on first syntax error.

Rationale:

- Calculator expressions are short (rarely more than one line), so finding one error at a time is sufficient.
- Error recovery adds significant complexity to the parser for minimal benefit in this context.
- Beginners benefit from clear, focused error messages about one problem at a time.

Consequences:

- Simpler parser implementation (no error recovery state tracking).
- Users must fix one error at a time rather than getting a comprehensive list.
- Acceptable trade-off given the calculator's educational purpose and typical expression length.

Option	Pros	Cons	Chosen?
Immediate Failure	Simple implementation, clear error focus	Only one error reported per run	**Yes**
Panic-Mode Recovery	Can report multiple errors	Complex synchronization logic, may skip valid code	No
Error Productions	Better error messages for specific patterns	Grammar complexity, many special cases	No

Semantic Errors: Meaningful But Invalid

Semantic errors occur when an expression is syntactically valid (follows the grammar) but doesn't make sense mathematically. These are caught during evaluation, not parsing. The distinction is crucial: `x + 2` is perfectly valid syntax—it's only invalid if `x` isn't defined in the current environment.

Common Semantic Error Types:

1. **Undefined Variables:** Referencing a variable that hasn't been assigned a value.
2. **Function Argument Issues:** Wrong number of arguments (though our calculator only has single-argument functions), or arguments outside the function's domain (e.g., negative for `sqrt`).
3. **Type Mismatches:** While our calculator only deals with numbers, more advanced versions might distinguish types.

The `Environment.get(name)` method is responsible for checking if a variable exists before returning its value. If the name isn't found, it raises a `NameError` with the variable name. Similarly, built-in functions should validate their arguments before computation.

Key Insight: Semantic errors represent a **separation of concerns** between syntax and meaning. The parser's job is to ensure the expression is well-formed; the evaluator's job is to ensure it's meaningful. This allows us to parse expressions with undefined variables (useful for some applications) and only complain when we actually try to evaluate them.

Runtime Errors: Computation Failures

Even with valid syntax and semantics, computations can fail during execution. These **runtime errors** are fundamentally different from the others because they depend on the actual values involved, not just the expression structure.

Primary Runtime Error Scenarios:

1. **Division by Zero:** `1 / 0`, `x / 0` (where x is any number)
2. **Domain Errors:** `sqrt(-1)` (real-valued square root of negative)
3. **Overflow/Underflow:** Extremely large or small results (`2 ^ 1000000`)
4. **Function-Specific Issues:** `tan(π/2)` (approaching infinity)

These errors are caught by the `Evaluator` when performing actual arithmetic operations or function calls. Unlike syntax errors, some runtime errors might be acceptable in certain mathematical contexts (complex numbers for `sqrt(-1)`), but for our real-valued calculator, they're errors.

9.2 Recovery and User Reporting

The Principle of Helpful Error Messages

A good error message answers three questions for the user:

1. **What** went wrong? (Error type and description)
2. **Where** did it happen? (Position in input)
3. **Why** is it wrong? (Context and expected pattern)

For each error category, we can provide progressively better context:

Lexical Errors: The lexer knows the exact character position. Report: `"Lexical error at position 5: Invalid character '@' in input '2 + @ 3'"`

Syntactic Errors: The parser knows which token caused the issue and what it expected. Report: `"Syntax error at token 'PLUS': Expected RPAREN after expression in '(2 + 3 +'"`

Semantic Errors: The evaluator knows which operation failed and why. Report: `"Semantic error: Variable 'radius' not defined in expression '2 * π * radius'"`

Runtime Errors: The evaluator knows the operation and operands. Report: `"Runtime error: Division by zero in expression '1 / (x - 2)' where x = 2"`

Error Reporting Strategy Table

Error Type	Detection Point	Information Available	Recommended Message Format
Lexical	`Lexer.tokenize()`	Character position, invalid lexeme	`"Lexical error at position {pos}: Invalid character '{char}'"`
Syntactic	`Parser.consume()`	Current token, expected token type, parser state	`"Syntax error at token '{token.lexeme}': Expected {expected_type} in {context}"`
Semantic	`Environment.get()` `Evaluator.evaluate()`	Variable/function name, expression context	`"Semantic error: {variable} not defined"` or `"Domain error: sqrt argument must be non-negative"`
Runtime	Arithmetic operations	Operation, operands, failure reason	`"Runtime error: {reason} in {operation} with values {values}"`

Position Tracking Implementation

To provide accurate position information, we need to track positions at multiple levels:

1. **Token Positions:** Each `Token` includes a `position` field (starting index in original input).
2. **Parser Context:** The parser maintains which tokens it's processing and can report token positions.
3. **Evaluation Context:** The evaluator can track which AST node is being evaluated for runtime errors.

For lexical errors, the position is the character index where scanning failed. For syntactic errors, it's typically the position of the unexpected token. For semantic and runtime errors, we can propagate position information through the AST—each node could optionally store a reference to the token(s) that created it, though for simplicity in this project, we might just report the error without position for these later stages.

Should We Attempt Error Recovery?

> **Architecture Decision: No Error Recovery**
>
> **Context:** In production compilers and interpreters, error recovery allows the system to continue parsing after an error to find multiple issues in one pass. Our calculator processes single expressions, not multi-line programs.
>
> **Options Considered:**
>
> 1. **No Recovery:** Stop at first error, report it clearly.
> 2. **Best-Effort Recovery:** Try to skip to a reasonable point and continue.
> 3. **Interactive Recovery:** In a REPL environment, suggest corrections.
>
> **Decision:** No recovery beyond reporting the error clearly.
>
> **Rationale:**
>
> - Calculator expressions are short (rarely more than 20 tokens).
> - Multiple errors in one expression usually indicate fundamental misunderstanding; fixing one often reveals the next.
> - Error recovery logic significantly increases parser complexity for marginal benefit.
> - Clear, single error messages are more helpful to learners than multiple potentially confusing ones.
>
> **Consequences:**
>
> - Simpler implementation focused on educational clarity.
> - Users correct one error at a time, which aligns with learning progression.
> - Acceptable for the target use case of evaluating individual mathematical expressions.

Option	Pros	Cons	Chosen?
No Recovery	Simple, clear error focus	Only one error per evaluation	**Yes**
Best-Effort Recovery	Can report multiple issues	Complex, may produce confusing secondary errors	No
Interactive Recovery	User-friendly, educational	Very complex, requires AI/ML or heuristics	No

Common Pitfalls in Error Handling

⚠️ Pitfall: Generic Error Messages

- **Description:** Reporting only "Error" or "Invalid input" without specifics.
- **Why It's Wrong:** Users can't fix what they don't understand. A message like "Error" forces guessing.
- **Fix:** Always include the error type, location (when known), and what was expected versus found.

⚠️ Pitfall: Losing Position Information

- **Description:** Lexer tracks positions but parser/evaluator doesn't propagate them to error reports.
- **Why It's Wrong:** "Syntax error" without position means scanning the entire expression to find the issue.
- **Fix:** Store position in tokens, reference in parser errors. Consider adding source position to AST nodes.

⚠️ Pitfall: Overly Technical Error Messages

- **Description:** Reporting parser internals like "Expected parse_primary to return non-None."
- **Why It's Wrong:** Users don't know (or care) about parser implementation details.
- **Fix:** Translate internal states to user-facing concepts: "Expected number, variable, or '(' here."

⚠️ Pitfall: Silent Failures on Runtime Errors

- **Description:** Returning `NaN` or `inf` without explanation for division by zero.
- **Why It's Wrong:** Users might not realize the result is invalid and continue calculations.
- **Fix:** Always raise an exception for runtime errors with a clear description.

⚠️ Pitfall: Inconsistent Error Handling Across Milestones

- **Description:** Adding new features (variables, functions) without updating error handling.
- **Why It's Wrong:** New capabilities introduce new failure modes that go unreported.
- **Fix:** For each milestone, identify new error cases and ensure they're properly detected and reported.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Reporting	Python's built-in exceptions with custom messages	Structured error objects with error codes, severity levels
Position Tracking	Integer character offsets	Line/column tracking with source line snippets
User Interface	Print error messages to console	Color-coded output with underline pointing to error

Recommended File/Module Structure

Extend the project structure to include error types:

```
calculator/
├── __init__.py
├── lexer.py           # Lexer class, LexerError exception
├── parser.py          # Parser class, syntax error handling
├── ast_nodes.py       # AST node classes
├── evaluator.py       # Evaluator class, semantic/runtime errors
├── environment.py     # Environment class, NameError
├── errors.py          # Custom exception hierarchy (optional)
└── main.py            # Command-line interface with error catching
```

Infrastructure Starter Code: Custom Exception Hierarchy

While Python's built-in exceptions work, a custom hierarchy clarifies error origins. Create `errors.py`:

```python
"""
Custom exception hierarchy for the calculator parser.
"""

class CalculatorError(Exception):
    """Base class for all calculator-specific errors."""
    def __init__(self, message: str, position: int = -1):
        super().__init__(message)
        self.message = message
        self.position = position  # -1 means position unknown/not applicable

    def __str__(self) -> str:
        if self.position >= 0:
            return f"[Position {self.position}] {self.message}"
        return self.message

class LexerError(CalculatorError):
    """Raised by the lexer when encountering invalid characters."""
    pass

class SyntaxError(CalculatorError):
    """Raised by the parser for malformed expression syntax."""
    pass

class SemanticError(CalculatorError):
    """Raised for semantic issues like undefined variables."""
    pass

class RuntimeError(CalculatorError):
    """Raised for computation failures like division by zero."""
    pass
```

Core Logic Skeleton: Enhanced Error Reporting

In `lexer.py` - Enhanced tokenize method with position tracking:

```python
def tokenize(self, input_str: str) -> List[Token]:
    """Convert input string to token list with position tracking.

    Raises:
        LexerError: If an invalid character is encountered.
    """
    tokens = []
    current_pos = 0

    while current_pos < len(input_str):
        char = input_str[current_pos]

        # Skip whitespace
        if char.isspace():
            current_pos += 1
            continue

        # TODO 1: Check for single-character tokens (+, -, *, /, ^, =, (, ))
        #   If match, create token with current position, advance position by 1

        # TODO 2: Check for multi-character tokens (numbers)
        #   Use isdigit() and '.' to collect full number
        #   Create NUMBER token with float(value) conversion
        #   Handle edge case: multiple '.' should raise LexerError

        # TODO 3: Check for identifiers (variables/functions)
        #   Start with letter or underscore, then alphanumeric/underscore
        #   Create IDENTIFIER token

        # TODO 4: If no pattern matches, raise LexerError
        #   Include character and position in error message

    tokens.append(Token(EOF, "", current_pos))
    return tokens
```

In `parser.py` - Enhanced consume method with better error messages:

```python
def consume(self, expected_type: str, error_msg: str = None) -> Token:
    """Consume current token if it matches expected type, else raise error.

    Args:
        expected_type: The token type expected.
        error_msg: Custom error message. If None, generates default.

    Returns:
        The consumed token.

    Raises:
        SyntaxError: If current token doesn't match expected_type.
    """
    if self.is_at_end():
        # TODO 1: Raise SyntaxError with message about unexpected end of input
        #   Include what was expected (e.g., "Expected ')' but reached end of expression")

    if self.current_token().type == expected_type:
        token = self.current_token()
        self.pos += 1
        return token

    # TODO 2: Generate helpful error message
    #   Default: f"Expected {expected_type} but found {current_token.type}"
    #   If custom error_msg provided, use it
    #   Include token position from current_token().position

    # TODO 3: Raise SyntaxError with the generated message and token position
```

In `evaluator.py` - Enhanced evaluate method with runtime checks:

PYTHON

```python
@classmethod
def evaluate(cls, node: ASTNode, env: Environment) -> float:
    """Evaluate AST node to numeric result with error handling."""
    try:
        if isinstance(node, Number):
            # TODO 1: Return node.value (safe, no runtime errors)

        elif isinstance(node, BinaryOp):
            # TODO 2: Recursively evaluate left and right operands
            # TODO 3: For division, check if right == 0, raise RuntimeError
            # TODO 4: Perform operation based on node.operator

        elif isinstance(node, UnaryOp):
            # TODO 5: Recursively evaluate operand
            # TODO 6: Apply unary minus (negation)

        elif isinstance(node, Variable):
            # TODO 7: Call env.get(node.name)
            #   This will raise NameError if undefined

        elif isinstance(node, Assign):
            # TODO 8: Evaluate value, call env.set(name, value)

        elif isinstance(node, FunctionCall):
            # TODO 9: Evaluate argument
            # TODO 10: Look up function in BUILTINS
            # TODO 11: Check domain constraints (e.g., sqrt(negative))
            # TODO 12: Call function with argument

    except ZeroDivisionError as e:
        # TODO 13: Convert to RuntimeError with calculator-specific message
        #   Include operation context (e.g., "Division by zero in 'x / y'")

    except ValueError as e:
        # TODO 14: Handle math domain errors (e.g., sqrt(-1))
        #   Convert to RuntimeError with helpful message

    except NameError as e:
        # TODO 15: Convert to SemanticError with variable name
```

Language-Specific Hints (Python)

- **Exception Chaining:** Use `raise NewError(...) from original_error` to preserve the original traceback while providing a user-friendly message.
- **Math Module Errors:** `math.sqrt(-1)` raises `ValueError`, which you can catch and convert to a `RuntimeError` with a clearer message.
- **Position Reporting:** For command-line interfaces, you can print the input string with a caret (`^`) under the error position:

```
Input: 2 + * 3
         ^
Syntax error: Expected expression after '+'
```

Debugging Tips for Error Handling

Symptom	Likely Cause	How to Diagnose	Fix
Error message says position -1	Not setting position when raising error	Check error instantiation in lexer/parser	Pass `position=token.position` or `position=current_pos`
"Syntax error" without context	Generic error message in `consume()`	Examine `Parser.consume()` implementation	Include expected vs found tokens and position
Division by zero not caught	Missing check before division	Add debug print of right operand value	Add `if right == 0: raise RuntimeError(...)` before division
Undefined variable error crashes REPL	Uncaught exception at top level	Wrap evaluation in try-except in main loop	Catch `SemanticError` and print message, continue REPL
Error position off by one	0-based vs 1-based indexing confusion	Test with simple inputs like `"2 @ 3"`	Document whether positions are 0 or 1-based, be consistent

10. Testing Strategy

> **Milestone(s):** All Milestones (1-3) — A comprehensive testing strategy is essential for verifying that each component works correctly in isolation and that they integrate properly to form a working calculator system. This section provides a structured approach to testing, from unit tests for individual components to integration tests for the complete expression evaluation pipeline, with specific checkpoints for each project milestone.

Testing Approach

Mental Model: The Manufacturing Quality Control Line

Imagine building the calculator as an assembly line in a factory that produces complex mechanical watches. Each component of the calculator represents a different station on the assembly line:

1. **Lexer Testing (Raw Material Inspection):** The first station receives raw metal sheets (input strings) and must cut them into precisely shaped gears and springs (tokens). We test this station by feeding it various metal sheets and verifying that it produces the correct shapes with the exact dimensions and types specified in the blueprints. If it produces a malformed gear or fails to recognize the material, we catch it here before it moves down the line.
2. **Parser Testing (Assembly Verification):** The second station takes the individual gears and springs and assembles them into movement subassemblies (AST nodes). We test this station by providing pre-cut components and verifying that they are assembled in the correct hierarchical structure, with the right gear ratios (operator precedence) and directional connections (associativity). If gears are assembled in the wrong order or with incorrect connections, the watch won't keep time properly.
3. **Evaluator Testing (Functional Testing):** The third station winds the assembled watch and checks if it keeps accurate time (produces correct numeric results). We test this station by providing fully assembled watches and verifying they compute the correct time for different settings. We also test edge cases like overwinding (division by zero) or missing components (undefined variables).
4. **Integration Testing (End-to-End Quality Control):** Finally, we test the entire assembly line by feeding raw metal sheets at the beginning and verifying that finished, accurate watches emerge at the end. This catches any subtle integration issues where components work individually but fail when connected.

This staged testing approach provides several critical benefits:

- **Early Bug Detection:** Problems are caught at the earliest possible stage, reducing debugging time.
- **Component Isolation:** When a test fails, we immediately know which component is responsible.
- **Regression Protection:** As we add new features in later milestones, existing tests ensure we don't break previously working functionality.
- **Design Validation:** Testing forces us to think about the interfaces between components, leading to cleaner separation of concerns.

Unit Testing Strategy for Individual Components

We recommend implementing unit tests for each of the three core components before writing integration tests. Each component should be tested in complete isolation, with carefully crafted inputs that exercise all possible code paths and edge cases.

Lexer Unit Tests focus on verifying that the tokenizer correctly converts character sequences into tokens:

Test Category	Description	Example Test Cases	Verification Points
Number Recognition	Tests that numeric literals are correctly tokenized	`"42"`, `"-3.14"`, `"0.5"`, `".7"`, `"1e-3"`	Token type is `NUMBER`, lexeme matches input, value is correctly parsed
Operator Recognition	Tests that arithmetic operators are correctly identified	`"+"`, `"-"`, `"*"`, `"/"`, `"^"`, `"="`	Correct token types (`PLUS`, `MINUS`, etc.) with proper lexemes
Parenthesis Recognition	Tests that grouping symbols are correctly tokenized	`"("`, `")"`	Token types `LPAREN` and `RPAREN`
Identifier Recognition	Tests that variable and function names are correctly tokenized	`"x"`, `"result"`, `"sin"`, `"sqrt"`	Token type `IDENTIFIER` with correct lexeme
Whitespace Handling	Tests that whitespace doesn't interfere with token recognition	`"2 + 3"`, `"\t\nx = 5\r"`	Tokens are produced in correct order, whitespace positions ignored
Position Tracking	Tests that token positions are accurately recorded for error reporting	`"2 + x"` with error at character 4	Each token's position field matches its starting index in input
Lexical Errors	Tests that invalid characters produce appropriate errors	`"2 @ 3"`, `"$var"`, `"3.14.15"`	`LexerError` raised with correct message and position

Parser Unit Tests focus on verifying that token sequences are correctly structured into valid ASTs:

Test Category	Description	Example Test Cases	Verification Points
Primary Expressions	Tests parsing of atomic expressions	`"42"`, `"x"`, `"(2+3)"`	Produces correct `Number` or `Variable` or nested expression
Unary Operations	Tests parsing of unary minus operations	`"-5"`, `"-x"`, `"--3"`	Produces `UnaryOp` nodes with correct operator and operand
Binary Operations	Tests parsing of binary operations with correct precedence	`"2+3"`, `"2*3+4"`, `"2+3*4"`	AST hierarchy reflects operator precedence
Associativity	Tests that operators with same precedence associate correctly	`"10-4-2"`, `"2^3^2"`	Left-associative operators chain left, right-associative chain right
Parentheses Grouping	Tests that parentheses override default precedence	`"(2+3)*4"`, `"2*(3+4)"`	AST structure shows parenthesized expressions as single units
Function Calls	Tests parsing of function applications	`"sin(0)"`, `"sqrt(4+5)"`	Produces `FunctionCall` nodes with correct function name and argument
Assignment Expressions	Tests parsing of variable assignments	`"x=5"`, `"y=2+3*4"`	Produces `Assign` nodes with identifier name and value expression
Syntax Errors	Tests detection of malformed expressions	`"2+"`, `"(2+3"`, `"sin 2)"`	`SyntaxError` raised with helpful message and position

Evaluator Unit Tests focus on verifying that ASTs are correctly evaluated to produce numeric results:

Test Category	Description	Example Test Cases	Verification Points
Number Evaluation	Tests evaluation of numeric leaf nodes	AST for `Number(42)`	Returns `42.0`
Binary Operation Evaluation	Tests computation of arithmetic operations	AST for `2+3`, `4*5`, `10/2`, `2^3`	Returns correct results for each operator
Unary Operation Evaluation	Tests evaluation of negation operations	AST for `-5`, `-(-3)`	Returns correct negated values
Variable Operations	Tests variable assignment and lookup	Sequence: `x=5` then evaluate `x`	Assignment returns value, lookup retrieves it
Function Application	Tests evaluation of built-in functions	AST for `sin(0)`, `sqrt(16)`, `abs(-5)`	Returns correct mathematical results
Environment Isolation	Tests that environments don't leak state between tests	Multiple tests using same evaluator instance	Each test starts with clean environment
Error Conditions	Tests detection of semantic and runtime errors	Division by zero, undefined variable	Appropriate error raised with helpful message
Floating-Point Precision	Tests handling of floating-point calculations	`0.1 + 0.2`, `sqrt(2)`	Uses appropriate tolerance for comparisons

Integration Testing Strategy for the Complete Pipeline

After unit tests pass for all components, we implement integration tests that exercise the complete expression evaluation pipeline from raw input string to final numeric result. These tests verify that the components work together correctly and that data flows properly between them.

Integration Test Structure:

1. **Setup:** Create fresh instances of all components (lexer, parser, evaluator, environment)
2. **Execution:** Feed an input string through the complete pipeline:
   - Lexer tokenizes the string
   - Parser builds AST from tokens
   - Evaluator computes result from AST with environment
3. **Verification:** Compare the computed result with the expected value (within floating-point tolerance for Milestone 2+)
4. **Cleanup:** Reset environment for next test

Key Integration Test Categories:

Category	Purpose	Example Expressions	Expected Results
Basic Arithmetic	Verify core calculator functionality	`"2+3*4"`, `"(2+3)*4"`, `"10-4-2"`	`14.0`, `20.0`, `4.0`
Unary Operations	Test negation in context	`"-2^2"`, `"(-2)^2"`, `"--5"`	`-4.0`, `4.0`, `5.0`
Variable Workflows	Test variable lifecycle	`"x=5; x+2"`, `"y=2*3; y^2"`	`7.0`, `36.0`
Function Applications	Test functions in expressions	`"sqrt(16)+2"`, `"sin(0)*cos(0)"`	`6.0`, `0.0`
Complex Expressions	Test multiple features combined	`"x=2; y=-x^2+3*sqrt(9)"`	`5.0` (when x=2)
Error Propagation	Verify errors flow through pipeline	`"2/0"`, `"unknown_var"`	Appropriate error with context

Test Organization Recommendations:

> **Key Insight:** Organize tests in parallel with your source code structure. This makes it easy to locate tests for specific components and ensures tests stay in sync with the code they test.

We recommend the following test file structure:

```
calculator/
├── src/
│   ├── __init__.py
│   ├── lexer.py          # Lexer implementation
│   ├── parser.py         # Parser implementation
│   ├── evaluator.py      # Evaluator implementation
│   ├── ast_nodes.py      # AST node classes
│   ├── environment.py    # Environment class
│   └── errors.py         # Custom exception classes
└── tests/
    ├── __init__.py
    ├── test_lexer.py     # Lexer unit tests
    ├── test_parser.py    # Parser unit tests
    ├── test_evaluator.py # Evaluator unit tests
    ├── test_integration.py # Full pipeline tests
    └── test_milestones.py # Milestone verification tests
```

Test Automation and Continuous Verification

For this project, we recommend using Python's built-in `unittest` framework due to its simplicity and availability. However, for more advanced testing features (better assertion messages, fixtures, parameterized tests), `pytest` is an excellent alternative.

Test Execution Strategy:

Test Type	Execution Command	When to Run	Purpose

Unit Tests	`python -m unittest tests.test_lexer`
	`python -m unittest tests.test_parser`
	`python -m unittest tests.test_evaluator`
		After implementing each component	Verify individual component correctness

All Tests	`python -m unittest discover tests`	Before committing changes	Ensure no regressions

Milestone Tests	`python -m unittest tests.test_milestones.TestMilestone1`
	`python -m unittest tests.test_milestones.TestMilestone2`
	`python -m unittest tests.test_milestones.TestMilestone3`
		After completing each milestone	Verify milestone acceptance criteria

Test Quality Metrics to Track:

Metric	Target	Why It Matters
Line Coverage	>90% for each component	Ensures all code paths are exercised
Branch Coverage	>85% for each component	Ensures both true and false branches of conditionals are tested
Mutation Score	>80% (if using mutation testing)	Measures test effectiveness at catching bugs
Test Readability	Clear test names and assertions	Makes tests maintainable and understandable

Milestone Verification Checkpoints

Each milestone in the project has specific acceptance criteria that must be verified through testing. The following tables provide concrete test cases for each milestone, organized by feature area. These tests serve as verification checkpoints—if all tests in a milestone's table pass, you can be confident that the milestone's requirements have been met.

Milestone 1: Basic Arithmetic Verification

Milestone 1 establishes the foundation with numbers, basic operators, and parentheses. The following tests verify all acceptance criteria:

Test ID	Expression	Expected Result	Test Category	Notes
M1-T1	`"42"`	`42.0`	Number Parsing	Simple integer
M1-T2	`"3.14"`	`3.14`	Number Parsing	Decimal number
M1-T3	`"-2.5"`	`-2.5`	Number Parsing	Negative decimal (lexer handles sign)
M1-T4	`".7"`	`0.7`	Number Parsing	Decimal without leading zero
M1-T5	`"2 + 3"`	`5.0`	Addition	Basic addition
M1-T6	`"7 - 5"`	`2.0`	Subtraction	Basic subtraction
M1-T7	`"3 * 4"`	`12.0`	Multiplication	Basic multiplication
M1-T8	`"12 / 4"`	`3.0`	Division	Basic division
M1-T9	`"2 + 3 + 4"`	`9.0`	Left Associativity	Addition chains left-to-right
M1-T10	`"10 - 4 - 2"`	`4.0`	Left Associativity	Subtraction chains left-to-right
M1-T11	`"8 / 4 / 2"`	`1.0`	Left Associativity	Division chains left-to-right
M1-T12	`"2 + 3 * 4"`	`14.0`	Precedence	Multiplication before addition
M1-T13	`"2 * 3 + 4"`	`10.0`	Precedence	Multiplication before addition (different order)
M1-T14	`"6 / 2 + 1"`	`4.0`	Precedence	Division before addition
M1-T15	`"2 + 6 / 3"`	`4.0`	Precedence	Division before addition
M1-T16	`"(2 + 3) * 4"`	`20.0`	Parentheses	Parentheses override precedence
M1-T17	`"2 * (3 + 4)"`	`14.0`	Parentheses	Parentheses create grouping
M1-T18	`"(2 + 3) * (4 - 1)"`	`15.0`	Parentheses	Multiple parenthesized groups
M1-T19	`"2 + 3 * (4 - 1)"`	`11.0`	Mixed	Combined precedence and parentheses
M1-T20	`" 2 + 3 "`	`5.0`	Whitespace	Ignoring whitespace
M1-E1	`"2 +"`	`SyntaxError`	Error Handling	Missing right operand
M1-E2	`"(2 + 3"`	`SyntaxError`	Error Handling	Unclosed parenthesis
M1-E3	`"2 + ) 3"`	`SyntaxError`	Error Handling	Mismatched parenthesis
M1-E4	`"2 / 0"`	`RuntimeError`	Error Handling	Division by zero

Verification Procedure for Milestone 1:

1. Implement lexer that tokenizes numbers and operators
2. Implement parser with precedence levels for addition/subtraction and multiplication/division
3. Implement evaluator for basic arithmetic operations
4. Run all M1 tests above
5. All tests should pass; any failures indicate issues with precedence, associativity, or parentheses handling

Milestone 2: Unary and Power Verification

Milestone 2 adds unary negation and exponentiation with right associativity. These tests build upon Milestone 1 functionality:

Test ID	Expression	Expected Result	Test Category	Notes
M2-T1	`"-5"`	`-5.0`	Unary Negation	Simple unary minus
M2-T2	`"-x"` (where x=3)	`-3.0`	Unary Negation	Unary minus on variable
M2-T3	`"--5"`	`5.0`	Nested Unary	Double negative
M2-T4	`"---2"`	`-2.0`	Nested Unary	Triple negative
M2-T5	`"2 + -3"`	`-1.0`	Unary in Expression	Unary in binary context
M2-T6	`"2 * -3"`	`-6.0`	Unary in Expression	Unary with multiplication
M2-T7	`"(-2) * 3"`	`-6.0`	Parenthesized Unary	Unary in parentheses
M2-T8	`"2 ^ 3"`	`8.0`	Exponentiation	Basic power
M2-T9	`"4 ^ 0.5"`	`2.0`	Exponentiation	Fractional exponent (square root)
M2-T10	`"2 ^ 3 ^ 2"`	`512.0`	Right Associativity	Right-associative: 2^(3^2) = 2^9
M2-T11	`"(2 ^ 3) ^ 2"`	`64.0`	Parentheses Override	Left grouping with parentheses
M2-T12	`"-2 ^ 2"`	`-4.0`	Precedence	Power before unary: -(2^2)
M2-T13	`"(-2) ^ 2"`	`4.0`	Parentheses	Parentheses change grouping
M2-T14	`"2 ^ -1"`	`0.5`	Negative Exponent	Power with negative exponent
M2-T15	`"2 * 3 ^ 2"`	`18.0`	Precedence	Power before multiplication: 2*(3^2)
M2-T16	`"2 ^ 3 * 4"`	`32.0`	Precedence	Power before multiplication: (2^3)*4
M2-T17	`"2 + 3 ^ 2 * 4"`	`38.0`	Complex Precedence	Power > Multiplication > Addition
M2-T18	`"(2 + 3) ^ 2"`	`25.0`	Mixed	Parentheses with power
M2-T19	`"-2 ^ 2 ^ 3"`	`-256.0`	Complex	-(2^(2^3)) = -(2^8)
M2-T20	`"(-2) ^ 2 ^ 3"`	`256.0`	Complex	((-2)^(2^3)) = (-2)^8
M2-E1	`"2 ^"`	`SyntaxError`	Error Handling	Missing exponent
M2-E2	`"^ 3"`	`SyntaxError`	Error Handling	Missing base

Precedence Hierarchy Verification for Milestone 2: The complete operator precedence hierarchy should be (from lowest to highest):

1. **Addition/Subtraction** (`+` , `-`): Left-associative
2. **Multiplication/Division** (`*` , `/`): Left-associative
3. **Unary Negation** (`-`): Right-associative (prefix)
4. **Exponentiation** (`^`): Right-associative

Verification Procedure for Milestone 2:

1. Extend parser to handle unary operators at appropriate precedence level
2. Add exponentiation parsing with right associativity
3. Update evaluator to compute powers
4. Run all M1 tests (should still pass)
5. Run all M2 tests
6. All tests should pass; pay special attention to right associativity of power and precedence of unary vs power

Milestone 3: Variables and Functions Verification

Milestone 3 adds variables, assignment, and built-in mathematical functions. These tests verify the complete calculator functionality:

Test ID	Expression Sequence	Expected Result	Test Category	Notes
M3-T1	`"x = 5"` → `"x"`	`5.0` then `5.0`	Variable Assignment	Assignment returns value, lookup retrieves it
M3-T2	`"x = 2 + 3"` → `"x * 2"`	`5.0` then `10.0`	Expression Assignment	Assignment with expression
M3-T3	`"x = 5"` → `"x = x + 2"` → `"x"`	`5.0` then `7.0` then `7.0`	Reassignment	Update variable with its own value
M3-T4	`"a = 2"` → `"b = 3"` → `"a * b"`	`2.0`, `3.0`, `6.0`	Multiple Variables	Multiple independent variables
M3-T5	`"result = (2 + 3) * 4"` → `"result / 2"`	`20.0` then `10.0`	Complex Assignment	Assignment with parenthesized expression
M3-T6	`"sin(0)"`	`0.0`	Function Call	Sine of zero
M3-T7	`"cos(0)"`	`1.0`	Function Call	Cosine of zero
M3-T8	`"sqrt(16)"`	`4.0`	Function Call	Square root
M3-T9	`"abs(-5)"`	`5.0`	Function Call	Absolute value
M3-T10	`"sqrt(2 + 2)"`	`2.0`	Nested Function	Function with expression argument
M3-T11	`"x = 4"` → `"sqrt(x)"`	`4.0` then `2.0`	Function with Variable	Function applied to variable
M3-T12	`"sin(0) + cos(0)"`	`1.0`	Function in Expression	Functions in binary operation
M3-T13	`"2 * sin(3.14159)"`	`0.0` (approx)	Floating-Point Function	Approximate result for pi
M3-T14	`"sqrt(2 ^ 4)"`	`4.0`	Complex Expression	Function with power expression
M3-T15	`"x = 3"` → `"y = sqrt(x^2)"` → `"y"`	`3.0`, `3.0`, `3.0`	Variable Chain	Variables used in function arguments
M3-T16	`"x = 2"` → `"-x ^ 2"`	`2.0` then `-4.0`	Unary with Variable	Unary operator on variable
M3-T17	`"angle = 0"` → `"sin(angle)"`	`0.0` then `0.0`	Descriptive Names	Meaningful variable names
M3-T18	`"result = sin(0) * cos(0) + sqrt(9)"`	`3.0`	Complete Expression	All features combined
M3-E1	`"undefined_var"`	`SemanticError`	Error Handling	Undefined variable reference
M3-E2	`"x = 5"` → `"x = "`	`5.0` then `SyntaxError`	Error Handling	Incomplete assignment
M3-E3	`"5 = 3"`	`SyntaxError`	Error Handling	Assignment to non-identifier
M3-E4	`"sin(2, 3)"`	`SyntaxError`	Error Handling	Function with wrong number of arguments
M3-E5	`"unknown_func(5)"`	`SemanticError`	Error Handling	Undefined function
M3-E6	`"sqrt(-1)"`	`RuntimeError` or `ValueError`	Error Handling	Domain error for function

Environment State Verification Tests: These additional tests verify that the environment correctly maintains state across evaluations:

Test Scenario	Steps	Expected Environment State
Independent Environments	Create two environments, set x=5 in env1, check x in env2	env1: `{'x': 5.0}`, env2: `{}` or error
Variable Shadowing	Set x=5 in parent env, create child env, set x=10 in child	Parent: `{'x': 5.0}`, Child: `{'x': 10.0}`
Variable Persistence	Set x=5, evaluate x+2, evaluate x*3	x remains 5.0 throughout

Verification Procedure for Milestone 3:

1. Implement `Environment` class with variable storage and lookup

2. Extend parser to handle assignment expressions and function calls
3. Update evaluator to support variable operations and function applications
4. Run all M1 and M2 tests (should still pass)
5. Run all M3 tests
6. Verify environment isolation between tests
7. All tests should pass, demonstrating complete calculator functionality

Comprehensive Integration Test Suite

After all milestones are complete, run a comprehensive integration test suite that exercises the complete system with complex, realistic expressions:

Complexity Level	Example Expression	Expected Result	Notes
Simple	`"2 + 3 * 4"`	`14.0`	Basic precedence
Intermediate	`"x = 5; y = x^2 - 2*x + 1"`	`16.0`	Quadratic expression with variables
Advanced	`"sqrt(sin(3.14159/2)^2 + cos(3.14159/2)^2)"`	`1.0` (approx)	Trigonometric identity
Real-world	`"principal = 1000; rate = 0.05; years = 3; principal * (1 + rate) ^ years"`	`1157.625`	Compound interest calculation
Edge Case	`"a = 1; b = 0; c = -1; (-b + sqrt(b^2 - 4*a*c)) / (2*a)"`	`1.0`	Quadratic formula

Regression Test Strategy: As you implement each milestone, maintain a growing suite of regression tests. Before making any changes to the code:

1. Run all existing tests to ensure they pass
2. Make your changes
3. Run all tests again to verify no regressions
4. Add new tests for the new functionality

This disciplined approach ensures that the calculator remains reliable as it grows in complexity across milestones.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation for Beginners
Testing Framework	`unittest` (Python standard library)	`pytest` (feature-rich third-party)	Start with `unittest` for simplicity
Assertion Library	Built-in `assert` statements	`pytest` assertions with detailed diffs	Built-in assertions are sufficient
Test Discovery	`python -m unittest discover`	`pytest` automatic discovery	Use unittest's discovery
Coverage Analysis	Manual inspection	`coverage.py` with HTML reports	Add `coverage.py` after basics are mastered
Mocking	Manual test doubles	`unittest.mock` library	Use simple fakes initially, learn mocking later

B. Recommended Test File Structure

```
calculator/
├── src/
│   ├── __init__.py
│   ├── lexer.py
│   ├── parser.py
│   ├── evaluator.py
│   ├── ast_nodes.py
│   ├── environment.py
│   └── errors.py
└── tests/
    ├── __init__.py
    ├── conftest.py             # Optional: Shared fixtures (if using pytest)
    ├── test_lexer.py           # Lexer unit tests
    ├── test_parser.py          # Parser unit tests
    ├── test_evaluator.py       # Evaluator unit tests
    ├── test_environment.py     # Environment unit tests
    ├── test_integration.py     # Full pipeline tests
    ├── test_milestones.py      # Milestone verification tests
    └── utils.py                # Test utilities and helpers
```

C. Infrastructure Starter Code (Complete Test Utilities)

Create a `tests/utils.py` file with helper functions that multiple test files can use:

PYTHON

```python
"""Test utilities for the calculator project."""

import math

from typing import List, Any


from src.lexer import Lexer, Token

from src.parser import Parser

from src.evaluator import Evaluator

from src.environment import Environment

from src.errors import LexerError, SyntaxError, SemanticError, RuntimeError


def tokenize_string(input_str: str) -> List[Token]:

    """Helper to tokenize a string for testing."""

    lexer = Lexer()

    return lexer.tokenize(input_str)


def parse_string(input_str: str) -> Any:  # Returns ASTNode, but we avoid import circularity

    """Helper to parse a string directly to AST."""

    tokens = tokenize_string(input_str)

    parser = Parser(tokens)

    return parser.parse()


def evaluate_string(input_str: str, env: Environment = None) -> float:

    """Helper to evaluate a string directly to result."""

    if env is None:

        env = Environment()

    node = parse_string(input_str)

    return Evaluator.evaluate(node, env)


def assert_float_equal(actual: float, expected: float, tolerance: float = 1e-10) -> None:

    """Assert that two floats are equal within tolerance."""

    assert abs(actual - expected) < tolerance, \

        f"Expected {expected}, got {actual} (difference: {abs(actual - expected)})"


def create_test_environment(**variables: float) -> Environment:

    """Create an environment with pre-set variables for testing."""

    env = Environment()

    for name, value in variables.items():

        env.set(name, value)

    return env
```

D. Core Logic Skeleton Code for Tests

Here's a complete starter template for `test_milestones.py` with TODOs for each test case:

PYTHON

```python
"""Milestone verification tests for the calculator project."""

import unittest

import math


from tests.utils import (

    evaluate_string,

    assert_float_equal,

    create_test_environment,

)

from src.errors import LexerError, SyntaxError, SemanticError, RuntimeError



class TestMilestone1(unittest.TestCase):

    """Tests for Milestone 1: Basic Arithmetic."""



    def test_number_parsing_integer(self):

        """M1-T1: Parse simple integer."""

        result = evaluate_string("42")

        self.assertEqual(result, 42.0)



    def test_number_parsing_decimal(self):

        """M1-T2: Parse decimal number."""

        result = evaluate_string("3.14")

        self.assertEqual(result, 3.14)



    def test_number_parsing_negative(self):

        """M1-T3: Parse negative decimal."""

        result = evaluate_string("-2.5")

        self.assertEqual(result, -2.5)



    def test_number_parsing_no_leading_zero(self):

        """M1-T4: Parse decimal without leading zero."""

        result = evaluate_string(".7")

        self.assertEqual(result, 0.7)



    def test_addition_basic(self):

        """M1-T5: Basic addition."""

        result = evaluate_string("2 + 3")

        self.assertEqual(result, 5.0)



    def test_subtraction_basic(self):

        """M1-T6: Basic subtraction."""
```

```
        result = evaluate_string("7 - 5")
        self.assertEqual(result, 2.0)

    def test_multiplication_basic(self):
        """M1-T7: Basic multiplication."""
        result = evaluate_string("3 * 4")
        self.assertEqual(result, 12.0)

    def test_division_basic(self):
        """M1-T8: Basic division."""
        result = evaluate_string("12 / 4")
        self.assertEqual(result, 3.0)

    def test_addition_associativity(self):
        """M1-T9: Left associativity of addition."""
        result = evaluate_string("2 + 3 + 4")
        self.assertEqual(result, 9.0)

    def test_subtraction_associativity(self):
        """M1-T10: Left associativity of subtraction."""
        result = evaluate_string("10 - 4 - 2")
        self.assertEqual(result, 4.0)  # (10-4)-2 = 4, not 10-(4-2)=8

    def test_division_associativity(self):
        """M1-T11: Left associativity of division."""
        result = evaluate_string("8 / 4 / 2")
        self.assertEqual(result, 1.0)  # (8/4)/2 = 1, not 8/(4/2)=4

    def test_precedence_multiplication_before_addition(self):
        """M1-T12: Multiplication has higher precedence than addition."""
        result = evaluate_string("2 + 3 * 4")
        self.assertEqual(result, 14.0)  # 2 + (3*4) = 14, not (2+3)*4=20

    def test_precedence_multiplication_before_addition_reverse(self):
        """M1-T13: Multiplication before addition (different order)."""
        result = evaluate_string("2 * 3 + 4")
        self.assertEqual(result, 10.0)  # (2*3) + 4 = 10, not 2*(3+4)=14

    def test_precedence_division_before_addition(self):
        """M1-T14: Division before addition."""
```

```
        result = evaluate_string("6 / 2 + 1")
        self.assertEqual(result, 4.0)  # (6/2) + 1 = 4, not 6/(2+1)=2

    def test_precedence_division_before_addition_right(self):
        """M1-T15: Division before addition (on right)."""
        result = evaluate_string("2 + 6 / 3")
        self.assertEqual(result, 4.0)  # 2 + (6/3) = 4, not (2+6)/3≈2.67

    def test_parentheses_override_precedence(self):
        """M1-T16: Parentheses override default precedence."""
        result = evaluate_string("(2 + 3) * 4")
        self.assertEqual(result, 20.0)  # (2+3)*4 = 20, not 2+(3*4)=14

    def test_parentheses_create_grouping(self):
        """M1-T17: Parentheses create explicit grouping."""
        result = evaluate_string("2 * (3 + 4)")
        self.assertEqual(result, 14.0)  # 2*(3+4) = 14, not 2*3+4=10

    def test_parentheses_multiple_groups(self):
        """M1-T18: Multiple parenthesized groups."""
        result = evaluate_string("(2 + 3) * (4 - 1)")
        self.assertEqual(result, 15.0)  # 5 * 3 = 15

    def test_mixed_precedence_and_parentheses(self):
        """M1-T19: Combined precedence and parentheses."""
        result = evaluate_string("2 + 3 * (4 - 1)")
        self.assertEqual(result, 11.0)  # 2 + 3*3 = 11

    def test_whitespace_ignored(self):
        """M1-T20: Whitespace is ignored in parsing."""
        result = evaluate_string("  2   +   3   ")
        self.assertEqual(result, 5.0)

    def test_error_missing_right_operand(self):
        """M1-E1: Syntax error for missing right operand."""
        with self.assertRaises(SyntaxError):
            evaluate_string("2 +")

    def test_error_unclosed_parenthesis(self):
        """M1-E2: Syntax error for unclosed parenthesis."""
```

```
        with self.assertRaises(SyntaxError):
            evaluate_string("(2 + 3")

    def test_error_mismatched_parenthesis(self):
        """M1-E3: Syntax error for mismatched parenthesis."""
        with self.assertRaises(SyntaxError):
            evaluate_string("2 + ) 3")

    def test_error_division_by_zero(self):
        """M1-E4: Runtime error for division by zero."""
        with self.assertRaises(RuntimeError):
            evaluate_string("2 / 0")

class TestMilestone2(unittest.TestCase):
    """Tests for Milestone 2: Unary and Power."""

    def test_unary_minus_simple(self):
        """M2-T1: Simple unary minus."""
        result = evaluate_string("-5")
        self.assertEqual(result, -5.0)

    def test_unary_minus_on_variable(self):
        """M2-T2: Unary minus on variable."""
        env = create_test_environment(x=3.0)
        result = evaluate_string("-x", env)
        self.assertEqual(result, -3.0)

    def test_double_negative(self):
        """M2-T3: Double negative (--5 = 5)."""
        result = evaluate_string("--5")
        self.assertEqual(result, 5.0)

    def test_triple_negative(self):
        """M2-T4: Triple negative (---2 = -2)."""
        result = evaluate_string("---2")
        self.assertEqual(result, -2.0)

    def test_unary_in_binary_expression(self):
        """M2-T5: Unary minus in binary expression."""
        result = evaluate_string("2 + -3")
```

```
    self.assertEqual(result, -1.0)


def test_unary_with_multiplication(self):
    """M2-T6: Unary minus with multiplication."""
    result = evaluate_string("2 * -3")
    self.assertEqual(result, -6.0)


def test_parenthesized_unary(self):
    """M2-T7: Parenthesized unary minus."""
    result = evaluate_string("(-2) * 3")
    self.assertEqual(result, -6.0)


def test_exponentiation_basic(self):
    """M2-T8: Basic exponentiation."""
    result = evaluate_string("2 ^ 3")
    self.assertEqual(result, 8.0)


def test_exponentiation_fractional(self):
    """M2-T9: Exponentiation with fractional exponent."""
    result = evaluate_string("4 ^ 0.5")
    assert_float_equal(result, 2.0)


def test_exponentiation_right_associative(self):
    """M2-T10: Right associativity of exponentiation."""
    result = evaluate_string("2 ^ 3 ^ 2")
    # Right-associative: 2^(3^2) = 2^9 = 512
    # NOT left-associative: (2^3)^2 = 8^2 = 64
    self.assertEqual(result, 512.0)


def test_exponentiation_parentheses_override(self):
    """M2-T11: Parentheses override right associativity."""
    result = evaluate_string("(2 ^ 3) ^ 2")
    self.assertEqual(result, 64.0)  # (8)^2 = 64


def test_precedence_power_before_unary(self):
    """M2-T12: Power has higher precedence than unary minus."""
    result = evaluate_string("-2 ^ 2")
    # Power before unary: -(2^2) = -4
    # NOT (-2)^2 = 4
    self.assertEqual(result, -4.0)
```

```python
def test_parentheses_change_power_grouping(self):
    """M2-T13: Parentheses change grouping for power."""
    result = evaluate_string("(-2) ^ 2")
    self.assertEqual(result, 4.0)


def test_exponentiation_negative_exponent(self):
    """M2-T14: Exponentiation with negative exponent."""
    result = evaluate_string("2 ^ -1")
    assert_float_equal(result, 0.5)


def test_precedence_power_before_multiplication(self):
    """M2-T15: Power before multiplication."""
    result = evaluate_string("2 * 3 ^ 2")
    # 2 * (3^2) = 2*9 = 18
    # NOT (2*3)^2 = 6^2 = 36
    self.assertEqual(result, 18.0)


def test_precedence_power_before_multiplication_right(self):
    """M2-T16: Power before multiplication (power on left)."""
    result = evaluate_string("2 ^ 3 * 4")
    # (2^3) * 4 = 8*4 = 32
    # NOT 2^(3*4) = 2^12 = 4096
    self.assertEqual(result, 32.0)


def test_complex_precedence_hierarchy(self):
    """M2-T17: Complete precedence hierarchy test."""
    result = evaluate_string("2 + 3 ^ 2 * 4")
    # Power first: 3^2 = 9
    # Then multiplication: 9*4 = 36
    # Finally addition: 2 + 36 = 38
    self.assertEqual(result, 38.0)


def test_parentheses_with_power(self):
    """M2-T18: Parentheses with power operator."""
    result = evaluate_string("(2 + 3) ^ 2")
    self.assertEqual(result, 25.0)  # 5^2 = 25


def test_complex_unary_and_power(self):
    """M2-T19: Complex expression with unary and right-associative power."""
```

```
        result = evaluate_string("-2 ^ 2 ^ 3")
        # Right-associative: 2^(2^3) = 2^8 = 256
        # Then unary: -256
        self.assertEqual(result, -256.0)

    def test_complex_parenthesized_power_chain(self):
        """M2-T20: Complex parenthesized power chain."""
        result = evaluate_string("(-2) ^ 2 ^ 3")
        # (-2)^(2^3) = (-2)^8 = 256
        self.assertEqual(result, 256.0)

    def test_error_missing_exponent(self):
        """M2-E1: Syntax error for missing exponent."""
        with self.assertRaises(SyntaxError):
            evaluate_string("2 ^")

    def test_error_missing_base(self):
        """M2-E2: Syntax error for missing base."""
        with self.assertRaises(SyntaxError):
            evaluate_string("^ 3")

class TestMilestone3(unittest.TestCase):
    """Tests for Milestone 3: Variables and Functions."""

    def test_variable_assignment_and_lookup(self):
        """M3-T1: Variable assignment returns value, lookup retrieves it."""
        env = Environment()
        # Assignment should return the value
        result1 = evaluate_string("x = 5", env)
        self.assertEqual(result1, 5.0)
        # Lookup should retrieve the stored value
        result2 = evaluate_string("x", env)
        self.assertEqual(result2, 5.0)

    def test_assignment_with_expression(self):
        """M3-T2: Assignment with expression."""
        env = Environment()
        result1 = evaluate_string("x = 2 + 3", env)
        self.assertEqual(result1, 5.0)
        result2 = evaluate_string("x * 2", env)
```

```
        self.assertEqual(result2, 10.0)


    def test_variable_reassignment(self):
        """M3-T3: Variable reassignment using its own value."""
        env = Environment()
        evaluate_string("x = 5", env)  # x = 5
        evaluate_string("x = x + 2", env)  # x = 5 + 2 = 7
        result = evaluate_string("x", env)
        self.assertEqual(result, 7.0)


    def test_multiple_independent_variables(self):
        """M3-T4: Multiple independent variables."""
        env = Environment()
        result1 = evaluate_string("a = 2", env)
        result2 = evaluate_string("b = 3", env)
        result3 = evaluate_string("a * b", env)
        self.assertEqual(result1, 2.0)
        self.assertEqual(result2, 3.0)
        self.assertEqual(result3, 6.0)


    def test_complex_assignment(self):
        """M3-T5: Assignment with parenthesized expression."""
        env = Environment()
        result1 = evaluate_string("result = (2 + 3) * 4", env)
        self.assertEqual(result1, 20.0)
        result2 = evaluate_string("result / 2", env)
        self.assertEqual(result2, 10.0)


    def test_function_sine(self):
        """M3-T6: Sine function."""
        result = evaluate_string("sin(0)")
        self.assertEqual(result, 0.0)


    def test_function_cosine(self):
        """M3-T7: Cosine function."""
        result = evaluate_string("cos(0)")
        self.assertEqual(result, 1.0)


    def test_function_square_root(self):
        """M3-T8: Square root function."""
```

```
        result = evaluate_string("sqrt(16)")
        self.assertEqual(result, 4.0)

    def test_function_absolute_value(self):
        """M3-T9: Absolute value function."""
        result = evaluate_string("abs(-5)")
        self.assertEqual(result, 5.0)

    def test_function_with_expression_argument(self):
        """M3-T10: Function with expression argument."""
        result = evaluate_string("sqrt(2 + 2)")
        self.assertEqual(result, 2.0)

    def test_function_applied_to_variable(self):
        """M3-T11: Function applied to variable."""
        env = create_test_environment(x=4.0)
        result = evaluate_string("sqrt(x)", env)
        self.assertEqual(result, 2.0)

    def test_functions_in_binary_operation(self):
        """M3-T12: Functions in binary operation."""
        result = evaluate_string("sin(0) + cos(0)")
        self.assertEqual(result, 1.0)

    def test_floating_point_function_approximation(self):
        """M3-T13: Floating-point approximation for sin(pi)."""
        result = evaluate_string("2 * sin(3.14159)")
        # sin(π) ≈ 0, so 2*0 ≈ 0
        assert_float_equal(result, 0.0, tolerance=1e-5)

    def test_function_with_power_expression(self):
        """M3-T14: Function with power expression argument."""
        result = evaluate_string("sqrt(2 ^ 4)")
        self.assertEqual(result, 4.0)  # sqrt(16) = 4

    def test_variable_chain_in_functions(self):
        """M3-T15: Variables used in function arguments."""
        env = Environment()
        evaluate_string("x = 3", env)
        evaluate_string("y = sqrt(x^2)", env)
```

```
        result = evaluate_string("y", env)
        self.assertEqual(result, 3.0)

    def test_unary_operator_on_variable(self):
        """M3-T16: Unary operator on variable."""
        env = create_test_environment(x=2.0)
        result = evaluate_string("-x ^ 2", env)
        self.assertEqual(result, -4.0)  # -(2^2) = -4

    def test_descriptive_variable_names(self):
        """M3-T17: Meaningful variable names work correctly."""
        env = Environment()
        evaluate_string("angle = 0", env)
        result = evaluate_string("sin(angle)", env)
        self.assertEqual(result, 0.0)

    def test_complete_expression_all_features(self):
        """M3-T18: Expression using all features."""
        result = evaluate_string("result = sin(0) * cos(0) + sqrt(9)")
        self.assertEqual(result, 3.0)  # 0*1 + 3 = 3

    def test_error_undefined_variable(self):
        """M3-E1: Semantic error for undefined variable."""
        with self.assertRaises(SemanticError):
            evaluate_string("undefined_var")

    def test_error_incomplete_assignment(self):
        """M3-E2: Syntax error for incomplete assignment."""
        env = create_test_environment(x=5.0)
        with self.assertRaises(SyntaxError):
            evaluate_string("x = ", env)

    def test_error_assignment_to_non_identifier(self):
        """M3-E3: Syntax error for assignment to non-identifier."""
        with self.assertRaises(SyntaxError):
            evaluate_string("5 = 3")

    def test_error_function_wrong_argument_count(self):
        """M3-E4: Syntax error for function with wrong number of arguments."""
        with self.assertRaises(SyntaxError):
```

```
            evaluate_string("sin(2, 3)")


    def test_error_undefined_function(self):
        """M3-E5: Semantic error for undefined function."""
        with self.assertRaises(SemanticError):
            evaluate_string("unknown_func(5)")


    def test_error_function_domain_error(self):
        """M3-E6: Runtime error for function domain error."""
        with self.assertRaises(RuntimeError):
            evaluate_string("sqrt(-1)")


if __name__ == '__main__':
    unittest.main()
```

E. Language-Specific Hints for Python Testing

1. **Use `unittest.TestCase`** as the base class for all test classes. It provides useful assertion methods like `assertEqual`, `assertAlmostEqual`, `assertRaises`, etc.
2. **For floating-point comparisons**, use `assertAlmostEqual` or our custom `assert_float_equal` helper to avoid precision issues:

```python
# Instead of:
self.assertEqual(0.1 + 0.2, 0.3)  # Might fail due to floating-point imprecision


# Use:
self.assertAlmostEqual(0.1 + 0.2, 0.3)  # Default tolerance 7 decimal places
# Or our helper with explicit tolerance:
assert_float_equal(0.1 + 0.2, 0.3, tolerance=1e-10)
```

3. **Test error conditions** using `assertRaises` as a context manager:

```python
with self.assertRaises(SyntaxError) as context:
    evaluate_string("2 +")
# You can also check the error message:
self.assertIn("unexpected end of input", str(context.exception))
```

4. **Set up test fixtures** in `setUp` method for common initialization:

```python
class TestEvaluator(unittest.TestCase):
    def setUp(self):
        """Create fresh environment for each test."""
        self.env = Environment()
```

5. **Name tests descriptively**. Test method names should indicate what they're testing:
   - Good: `test_addition_left_associative`
   - Bad: `test1` or `test_addition`

F. Milestone Checkpoint Verification

After completing Milestone 1:

1. Run: `python -m unittest tests.test_milestones.TestMilestone1`
2. Expected: All 24 tests pass (20 positive tests + 4 error tests)
3. If tests fail, check:
   - Are operator precedence rules correctly implemented? (Multiplication before addition)
   - Does subtraction/division associate left-to-right? (`10-4-2` should be 4, not 8)
   - Do parentheses correctly override precedence?

After completing Milestone 2:

1. Run: `python -m unittest tests.test_milestones.TestMilestone1` (should still pass)
2. Run: `python -m unittest tests.test_milestones.TestMilestone2`
3. Expected: All 22 tests pass (20 positive + 2 error tests)
4. Critical checks:
   - Power is right-associative: `2^3^2` = 512, not 64
   - Power has higher precedence than unary: `-2^2` = -4, not 4
   - Unary has higher precedence than multiplication: `2*-3` works correctly

After completing Milestone 3:

1. Run all previous milestone tests (should still pass)
2. Run: `python -m unittest tests.test_milestones.TestMilestone3`
3. Expected: All 24 tests pass (18 positive + 6 error tests)
4. Critical checks:
   - Variable assignment returns the assigned value
   - Variable lookup retrieves the stored value
   - Functions work with variables as arguments
   - Errors are raised for undefined variables/functions

Final comprehensive test:

1. Run: `python -m unittest discover tests`
2. Expected: All tests across all test files pass
3. This verifies the complete calculator implementation

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Basic arithmetic gives wrong result (e.g., `2+3*4=20`)	Incorrect operator precedence	Print the AST structure to see if multiplication is nested under addition	Review parser precedence levels; ensure multiplication/division parse at higher level than addition/subtraction
Subtraction/division gives unexpected result (e.g., `10-4-2=8`)	Wrong associativity	Check AST structure: should be left-associative `((10-4)-2)` not `(10-(4-2))`	Ensure parser builds left-associative trees for left-associative operators
Power operator gives wrong result (e.g., `2^3^2=64`)	Wrong associativity (left instead of right)	Check AST: should be right-associative `2^(3^2)`	Implement right associativity in `parse_power()` using while loop with precedence check
Unary minus on power gives wrong sign (e.g., `-2^2=4`)	Incorrect precedence (unary before power)	Check AST: should be `-(2^2)` not `(-2)^2`	Ensure unary operators parse at lower precedence than power in parser hierarchy
Variable assignment works but lookup fails	Environment not being passed or updated correctly	Add debug prints to see if environment is modified during assignment	Ensure assignment updates environment and lookup retrieves from same environment
Function call parsed as variable	Parser not detecting parentheses after identifier	Check token stream: identifier should be followed by `LPAREN`	In `parse_primary()`, check if identifier is followed by `LPAREN` to distinguish variable from function
Multiple tests interfere with each other	Test isolation failure (shared environment)	Check if tests use separate Environment instances	Use `setUp()` to create fresh environment for each test, or pass new Environment to each evaluation
Floating-point tests fail intermittently	Precision issues in comparisons	Print actual vs expected values with high precision	Use `assertAlmostEqual` or custom `assert_float_equal` with tolerance instead of exact equality

Debugging Technique: Visualize the AST Add a debug method to print AST structure:

```python
def print_ast(node, indent=0):
    """Print AST structure for debugging."""
    indent_str = "  " * indent
    if isinstance(node, Number):
        print(f"{indent_str}Number({node.value})")
    elif isinstance(node, UnaryOp):
        print(f"{indent_str}UnaryOp({node.operator})")
        print_ast(node.operand, indent + 1)
    elif isinstance(node, BinaryOp):
        print(f"{indent_str}BinaryOp({node.operator})")
        print(f"{indent_str}  left:")
        print_ast(node.left, indent + 2)
        print(f"{indent_str}  right:")
        print_ast(node.right, indent + 2)
    # ... handle other node types
```

Use this to verify that the parser is building the correct tree structure before debugging evaluation logic.

11. Debugging Guide

> **Milestone(s):** All Milestones (1-3) — Debugging skills are essential throughout development. This section provides practical tools and diagnostic tables to identify and fix common bugs in your calculator implementation.

When building a parser and evaluator, bugs often manifest as incorrect calculation results or cryptic error messages. The hierarchical nature of the system—where a small mistake in the lexer or parser can produce wrong results much later in evaluation—makes debugging particularly challenging. This guide provides two complementary approaches: a **symptom-based lookup table** for common issues and **systematic inspection techniques** to examine internal data structures.

Common Bugs: Symptom → Cause → Fix

Think of debugging your calculator like diagnosing a car that won't start. The symptom (engine doesn't turn over) could have many causes (dead battery, faulty starter, empty fuel tank). You need systematic checks to isolate the problem. The table below maps observable symptoms to their likely root causes, along with specific diagnostic steps and fixes.

Symptom	Likely Cause	How to Diagnose	Fix
`2+3*4` evaluates to `20` instead of `14`	Operator **precedence** incorrectly implemented—addition and multiplication have equal precedence or are evaluated strictly left-to-right.	1. Print the AST for `2+3*4`. 2. If the AST shows `(2+3)*4` instead of `2+(3*4)`, the parser is not respecting multiplication's higher precedence.	Ensure your recursive descent parser has separate `parse_term()` (for `+/-`) and `parse_factor()` (for `*/`) methods, with `parse_term()` calling `parse_factor()` for its operands. Multiplication should be parsed at a deeper recursion level than addition.
`--5` causes a syntax error or evaluates incorrectly	The lexer or parser doesn't handle **unary minus** correctly, either treating the second `-` as a binary operator or failing to recognize consecutive unary operators.	1. Print tokens for `--5`. You should see two `MINUS` tokens followed by `NUMBER`. 2. Check if `parse_unary()` correctly handles multiple consecutive unary operators by calling itself recursively.	In `parse_unary()`, if the current token is `MINUS`, consume it and recursively call `parse_unary()` for the operand, then wrap in a `UnaryOp` node. This allows `--5` → `-(-5)`.
`2^3^2` evaluates to `64` instead of `512`	Exponentiation has **left associativity** instead of right associativity, so it's evaluated as `(2^3)^2 = 8^2 = 64`.	1. Print AST for `2^3^2`. If it shows `((2^3)^2)`, associativity is wrong. 2. Check `parse_power()`: after parsing the left operand, it should check for `CARET` and recursively call `parse_power()` (not itself) for the right operand.	Implement right associativity: In `parse_power()`, after parsing the base, if you see `CARET`, consume it and recursively call `parse_power()` (not `parse_unary()`) for the exponent. This makes the AST `(2^(3^2))`.
`-2^2` evaluates to `4` instead of `-4`	**Unary minus binds too tightly**—it's evaluated after exponentiation (`(-2)^2`) instead of before (`-(2^2)`).	1. Print AST for `-2^2`. If it shows `((-2)^2)`, unary has higher precedence than power. 2. Check precedence hierarchy: `parse_unary()` should call `parse_power()` (not vice versa).	Ensure the precedence chain: `parse_expression()` → `parse_term()` → `parse_factor()` → `parse_power()` → `parse_unary()` → `parse_primary()`. Unary operators are parsed *before* exponentiation in the primary context, but exponentiation has higher precedence in evaluation.
`x = 5` followed by `x + 2` says "Variable 'x' not defined"	1. **Assignment not storing** to environment. 2. **Assignment returning a value** but not modifying environment. 3. **Environment scope issue**—different environments used for assignment and lookup.	1. After parsing `x = 5`, print the environment's variables dict. 2. Check `evaluate()` for `Assign` node: it should call `env.set()` *and* return the value. 3. Ensure the same environment instance is passed to all `evaluate()` calls.	In `evaluate()` for `Assign` node: evaluate the value expression, call `env.set(name, value)`, then return the value. Use a single `Environment` instance for the entire evaluation session.
`sqrt(16)` causes "Expected RPAREN" error	Function call parsing stops at the identifier, treating `sqrt` as a variable, not expecting parentheses.	1. Print tokens for `sqrt(16)`: should be `IDENTIFIER("sqrt")`, `LPAREN`, `NUMBER(16)`, `RPAREN`. 2. Check `parse_primary()`: after parsing an identifier, it must check for `LPAREN` to differentiate variable from function call.	In `parse_primary()`, when you see an `IDENTIFIER`, peek ahead. If next token is `LPAREN`, consume it, parse the argument expression, consume `RPAREN`, and return a `FunctionCall` node. Otherwise, return a `Variable` node.
`2 +` (trailing operator) causes infinite recursion or cryptic error	Parser doesn't handle **missing operand** gracefully—it tries to parse an expression after `+` and gets stuck.	1. The parser will call `parse_term()`, which calls `parse_factor()` for the right operand, which expects a number/parenthesis/identifier. 2. When it sees `EOF` or unexpected token, it may loop or crash.	In all parsing methods, when you need to parse an operand (e.g., after consuming an operator), check if the next token can start an expression. If not, raise a `SyntaxError` with position. Use `consume()` with appropriate error messages.
`3 / 0` crashes with Python's ZeroDivisionError	No **runtime error handling** for division by zero—the evaluator directly uses Python's division operator.	The evaluator's division case doesn't check the right operand's value before dividing.	In `evaluate()` for `BinaryOp` with operator `/`, check if the right operand evaluates to 0 (within floating-point epsilon). If so, raise a `RuntimeError` with descriptive message.
`2.5.3` is accepted as a valid number	Lexer's **number scanning** doesn't stop at the second decimal point—it greedily consumes `2.5.3` as one number.	The lexer's number parsing logic allows multiple decimal points.	Implement finite-state scanning for numbers: consume digits, then if you see a `.`, consume it and subsequent digits. If you see another `.`, stop—it's the start of a new token.
`sin ( 45 )` (with spaces) fails to parse	Lexer produces correct tokens, but parser's `match()` or `consume()` doesn't skip whitespace—tokens have positions but parser doesn't advance.	Actually, whitespace is handled by the lexer (ignored). The issue might be in function call parsing: `parse_primary()` sees `IDENTIFIER`, then peeks for `LPAREN`, but there's whitespace between. The lexer should have skipped whitespace, so	Ensure lexer's `tokenize()` completely skips whitespace characters (space, tab, newline) before starting to scan each token. The token stream should contain no whitespace tokens.

Symptom	Likely Cause	How to Diagnose	Fix
		`peek()` should return `LPAREN`. If not, check lexer's whitespace handling.	
`(2+3` (unclosed parenthesis) causes infinite recursion	Parser doesn't detect **mismatched parentheses**—when looking for `RPAREN` in `parse_primary()`, it hits `EOF` and may loop.	In `parse_primary()`, when you see `LPAREN`, you consume it, parse an expression, then call `consume(RPAREN, ...)`. If the token isn't `RPAREN` (maybe `EOF`), `consume()` should raise `SyntaxError`.	Ensure `consume()` raises `SyntaxError` with the expected token type and position. Include the position of the opening `LPAREN` in the error message for context.
`2 + 3 *` followed by Enter in interactive mode hangs	In an **interactive REPL**, the parser waits for more input because expression is incomplete.	This is actually expected behavior for an interactive calculator—it should wait for the complete expression. If you want to report an error immediately, you'd need a different approach (like checking if the last token before EOF is an operator).	For a beginner project, this is acceptable. For better UX, you could detect that the last token is a binary operator and raise a syntax error.

> **Key Insight:** Most parsing bugs stem from **incorrect precedence/associativity** or **incomplete handling of edge cases**. The AST is your best debugging tool—if the tree structure is wrong, the result will be wrong regardless of correct evaluation logic.

Debugging Techniques

When the symptom-cause table doesn't match your issue, or you need to understand why a particular bug occurs, you need to **inspect the internal state** of your system. Think of this as putting your calculator under an X-ray machine—you can see the bones (tokens) and organs (AST) without running the whole body.

1. Printing the Token Stream

The token stream is the first transformation of the input. If tokens are wrong, everything downstream will be wrong. Add a debug flag or temporary print statement in your lexer to see exactly what tokens are produced.

Mental Model: Imagine you're proofreading a document that was typed by someone speaking the words aloud. The token stream is like the typed transcript—you need to verify that "two plus three times four" was transcribed as `[2, +, 3, *, 4]` and not `[2, +, 3, times, 4]` (where "times" is an unknown word).

What to Look For:

- Are numbers correctly recognized? (Check `NUMBER` tokens with correct `lexeme`)
- Are operators the expected token types? (`PLUS`, `MINUS`, etc.)
- Are parentheses separate tokens? (`LPAREN`, `RPAREN`)
- Are identifiers correctly captured? (`IDENTIFIER` with name)
- Is there an `EOF` token at the end?
- Is whitespace completely absent from the token list?

Example Diagnostic Output for `"2 + 3 * 4"`:

```
Tokens:
  [0] NUMBER("2") at pos 0
  [1] PLUS("+") at pos 2
  [2] NUMBER("3") at pos 4
  [3] STAR("*") at pos 6
  [4] NUMBER("4") at pos 8
  [5] EOF("") at pos 9
```

2. Visualizing the Abstract Syntax Tree

The AST captures the intended structure of the expression. A visual representation reveals whether precedence and associativity are correctly encoded.

Mental Model: Consider a family tree diagram showing parent-child relationships. The AST is similar—each operator is a parent with its operands as children. Printing the tree lets you verify that multiplication is a child of addition (correct: `+(2, *(3, 4))`) rather than the opposite (incorrect: `*(+(2, 3), 4)`).

Implementation Approaches:

- **Indented Text Tree:** Use recursion with increasing indentation to show nesting.
- **Parenthesized Notation:** Print the expression with parentheses around every operation, revealing the implicit grouping.
- **Graphical Output:** For advanced debugging, use Graphviz to generate actual tree diagrams (beyond beginner scope).

Example AST Visualization for `"2 + 3 * 4"`:

```
BinaryOp('+')
├── Number(2)
└── BinaryOp('*')
    ├── Number(3)
    └── Number(4)
```

Or in parenthesized form:

```
(+ 2 (* 3 4))
```

What to Look For in the AST:

- Are binary operators linked to the correct left and right sub-expressions?
- For exponentiation, is the tree right-associative? (`^ 2 (^ 3 2)` not `(^ (^ 2 3) 2)`)
- Do unary operators appear as direct parents of their operand?
- Are function calls structured as `FunctionCall` with an argument sub-tree?
- Are assignment nodes separate from binary operators?

3. Step-by-Step Debugging Strategy

When you encounter a bug:

1. **Write a minimal test case** that reproduces the bug (e.g., `"2+3*4"`).
2. **Print the token stream**—verify lexical analysis is correct.
3. **Print the AST**—verify parsing captures the intended structure.
4. **Trace evaluation manually** using the AST—simulate what your evaluator should do.
5. **Add debug prints to evaluator** to see actual traversal order and intermediate results.
6. **Compare expected vs. actual** at each stage to locate the first deviation.

> **Pro Tip:** Create a helper function `debug_calc(expr)` that runs the entire pipeline and prints tokens, AST, and result. Use this for rapid experimentation during development.

Implementation Guidance

This subsection provides concrete code to implement the debugging techniques described above. For a beginner project, simple print-based debugging is most appropriate.

Technology Recommendations:

Component	Simple Option	Advanced Option
Token Visualization	Print tokens with `__repr__`	Use rich/colorful terminal output
AST Visualization	Recursive print with indentation	Generate Graphviz DOT files
Interactive Debugging	Add debug flags to classes	Use Python's `pdb` debugger

Recommended File/Module Structure: Add debugging utilities to your existing modules—no new files needed, though you could create a `debug.py` helper if desired.

```
calculator/
  lexer.py        # Add debug_print_tokens() function
  parser.py       # Add debug_print_ast() function
  ast_nodes.py    # Add __repr__ or __str__ methods to AST nodes
```

Infrastructure Starter Code (Complete): Here are complete, ready-to-use debugging functions that you can add to your project.

Token Debugging (add to `lexer.py`):

```python
def debug_print_tokens(tokens):
    """Print token stream for debugging."""
    print("Tokens:")
    for i, token in enumerate(tokens):
        # Show position as index for simplicity; you could show line/col
        print(f"  [{i}] {token.type}({repr(token.lexeme)}) at pos {token.position}")
```

AST Debugging (add to `parser.py` or a new `debug.py`):

```python
def debug_print_ast(node, indent=0):
    """Recursively print AST with indentation."""
    space = "    " * indent
    if isinstance(node, Number):
        print(f"{space}Number({node.value})")
    elif isinstance(node, UnaryOp):
        print(f"{space}UnaryOp('{node.operator}')")
        debug_print_ast(node.operand, indent + 1)
    elif isinstance(node, BinaryOp):
        print(f"{space}BinaryOp('{node.operator}')")
        debug_print_ast(node.left, indent + 1)
        debug_print_ast(node.right, indent + 1)
    elif isinstance(node, Variable):
        print(f"{space}Variable('{node.name}')")
    elif isinstance(node, Assign):
        print(f"{space}Assign('{node.name}')")
        debug_print_ast(node.value, indent + 1)
    elif isinstance(node, FunctionCall):
        print(f"{space}FunctionCall('{node.name}')")
        debug_print_ast(node.argument, indent + 1)
    else:
        print(f"{space}Unknown node type: {type(node)}")
```

Core Logic Skeleton Code: Enhance your AST node classes with readable `__repr__` methods to make debugging easier.

In `ast_nodes.py` (or wherever you define AST classes):

```python
class Number:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f"Number({self.value})"

    # TODO: Keep existing evaluate logic...

class BinaryOp:
    def __init__(self, operator, left, right):
        self.operator = operator
        self.left = left
        self.right = right

    def __repr__(self):
        return f"BinaryOp('{self.operator}', {self.left}, {self.right})"

    # TODO: Keep existing evaluate logic...

# TODO: Add similar __repr__ for UnaryOp, Variable, Assign, FunctionCall
```

Language-Specific Hints (Python):

- Use `repr()` to safely print strings with quotes (shows escape characters).
- For more detailed token debugging, add line and column numbers to your `Token` class.
- Consider using `dataclasses` for AST nodes to automatically get a decent `__repr__`.
- For interactive debugging, add a conditional debug flag:

```python
DEBUG = True  # Set to False in production

def parse(self, tokens):
    if DEBUG:
        debug_print_tokens(tokens)
    # ... parsing logic
    if DEBUG:
        debug_print_ast(ast)
    return ast
```

Debugging Tips (Structured Approach):

Symptom	How to Diagnose	Fix
Can't tell where bug is	Insert print at each pipeline stage: `print("LEXER:", tokens); print("PARSER:", ast); print("EVAL:", result)`	The stage where output looks wrong is where to focus.
AST too large to read	Use the indented tree printer with limited depth, or print only problematic subtree.	Focus on the specific expression part causing issues.
Floating-point rounding issues	Print values with high precision: `print(f"{result:.15f}")`.	Understand that `0.1 + 0.2 != 0.3` exactly due to binary floating-point representation.
Infinite recursion in parser	Add recursion depth limit or print entering/exiting each parsing method.	Check grammar left-recursion: `parse_term()` shouldn't call `parse_term()` first thing.

Milestone Checkpoint for Debugging: After implementing debugging helpers, test them with these expressions:

1. `"2+3*4"` → Should show multiplication nested under addition in AST.
2. `"--5"` → Should show two nested `UnaryOp` nodes.
3. `"2^3^2"` → Should show right-associative tree: `^` with left=`2`, right=`^` with left=`3`, right=`2`.
4. `"x = 5"` → Should show `Assign` node with `Variable('x')` and `Number(5)`.
5. `"sin(45)"` → Should show `FunctionCall('sin')` with `Number(45)` argument.

If your debugging outputs match these expectations, your parser is likely correct. If not, the visual representation directly shows what needs fixing.

12. Future Extensions

> **Milestone(s):** Beyond All Milestones (Post-Project Exploration) — This forward-looking section explores how the calculator's architecture can accommodate advanced features beyond the core three milestones, turning it from a learning project into a more capable tool.

Mental Model: The Calculator as a LEGO Set

Think of the current calculator implementation as a basic LEGO set with fundamental building blocks: the red bricks of lexing, the blue bricks of parsing, and the yellow bricks of evaluation. **Future Extensions** are about adding specialized pieces — technic gears for new operators, transparent windows for a user interface, or minifigures for advanced functionality — that click into the existing structure without requiring a complete rebuild. The beauty of your current **separation of concerns** architecture is that adding most new features involves modifying or extending *just one or two components*, not redesigning the entire system. This section explores what those new LEGO pieces might look like and how they'd connect.

Architectural Extensibility Assessment

Before diving into specific features, let's review why the current design enables clean extension:

Component	Extensibility Point	How to Extend	Example Feature
Lexer	Token type definitions	Add new token types to the `Token` class's `type` field (e.g., `COMMA`, `PERCENT`). Update scanning logic in `tokenize()` to recognize new character sequences.	Adding a modulo `%` operator.
Parser	Grammar methods	Add new parsing methods for new syntax, or extend existing precedence levels in the recursive descent methods (`parse_term`, `parse_factor`, etc.).	Adding comparison operators (`<`, `>`, `==`).
AST	Node class hierarchy	Create new `ASTNode` subclasses (e.g., `ComparisonOp`, `Constant`). The evaluator must be updated to handle these new node types.	Adding a ternary conditional operator (`a ? b : c`).
Evaluator	Visitor-like `evaluate`	Add new `if` branches to handle new AST node types. Extend `BUILTINS` dictionary with new function implementations.	Adding hyperbolic functions (`sinh`, `cosh`).
Environment	Symbol table	Enhance the `Environment` class to support nested scopes, constants, or type information.	Adding user-defined functions.

The pipeline's modular nature means you can prototype a new feature by working on one component at a time, testing it in isolation before integrating it with the whole system.

Potential Features

The following table outlines advanced features that learners could implement to deepen their understanding of language implementation. Each entry includes the architectural impact and considerations for implementation.

Feature Category	Example Features	Brief Description	Design Considerations & Architectural Impact	Difficulty
Extended Math Functions	`tan`, `log`, `ln`, `exp`, `floor`, `ceil`, `round`	Additional trigonometric, logarithmic, and rounding functions beyond `sin`, `cos`, `sqrt`.	**Impact:** Low. Primarily affects the Evaluator's `BUILTINS` dictionary. **Considerations:** Must decide on numeric domain (e.g., `log(0)` errors). May want to add a constant for `pi` and `e` (see Constants).	🟢 Easy
Mathematical Constants	`pi`, `e`, `phi` (golden ratio)	Predefined, read-only numeric values that can be used in expressions.	**Impact:** Medium. Affects Parser (treat identifiers as constants) and Environment (distinguish constants from variables). **Considerations:** Should constants be stored in a separate `constants` dict in `Environment`? Should they be immutable? Parser's `parse_primary` must resolve `IDENTIFIER` to either a variable, constant, or function call.	🟡 Medium
Comparison & Boolean Operators	`>`, `<`, `>=`, `<=`, `==`, `!=`, `and`, `or`, `not`	Relational comparisons and logical operations, returning a boolean (or 1/0).	**Impact:** High. Affects Lexer (new tokens), Parser (new precedence level below addition? New `parse_comparison` method), AST (new `ComparisonOp` node), Evaluator (returns boolean/numeric). **Considerations:** Major design decision: Should expressions now have a boolean type, or map `True` to 1.0 and `False` to 0.0? This introduces a **type system**.	🔴 Hard
Conditional (Ternary) Expression	`condition ? true_expr : false_expr`	Returns one of two values based on a boolean condition.	**Impact:** High. Affects Lexer (`?`, `:` tokens), Parser (new grammar rule, likely right-associative), AST (new `TernaryOp` node), Evaluator (must evaluate condition first, then only the chosen branch). **Considerations:** This requires boolean expressions (see above). Introduces **short-circuit evaluation**, a new control flow concept.	🔴 Hard
Modulo & Integer Division	`%` (modulo), `//` (floor division)	Remainder after division and integer division.	**Impact:** Low-Medium. Affects Lexer (new tokens), Parser (add to `parse_factor` level with same precedence as `*` and `/`). **Considerations:** Should `%` work on floats? How to handle negative operands (language-specific behavior)?	🟢 Easy
REPL Interface	Interactive Read-Eval-Print Loop	A command-line interface that reads an expression, evaluates it, prints the result, and loops.	**Impact:** Medium. Adds a new top-level driver component. **Considerations:** Needs to handle multi-line input, maintain environment state across commands, and provide commands like `quit` or `clear`. The `Environment` must persist between evaluations. Great for usability testing.	🟡 Medium
Persistent Variables (File I/O)	`save "vars.txt"`, `load "vars.txt"`	Save the current environment's variables to a file and load them later.	**Impact:** Medium. Affects the top-level driver or a new utility module. **Considerations:** Requires file I/O and a simple serialization format (e.g., JSON, or `name=value` per line). The `Environment` class might need a `to_dict()` and `from_dict()` method.	🟡 Medium
Multiple Statements / Expression Lists	`x=5; y=x*2; x+y`	Evaluate multiple expressions separated by semicolons, returning the value of the last one.	**Impact:** Medium-High. Affects Lexer (`SEMICOLON` token), Parser (new top-level `parse_program` that returns a list of statements), AST (new `Program` or `Block` node), Evaluator (evaluate sequentially). **Considerations:** This fundamentally changes the grammar from a single expression to a list of statements. Requires distinguishing between expressions (which produce values) and statements (which have side effects).	🔴 Hard
User-Defined Functions	`f(x) = x^2 + 1`	Allow the user to define and later call their own functions.	**Impact:** Very High. Affects all components. **Considerations:** Requires a major extension to the Environment to store function definitions (parameters, body AST). Parser must handle function definition syntax and function calls with multiple arguments. Evaluator must implement **function call frames** and **parameter binding**. This is a mini-language implementation project in itself.	🔴🔴 Very Hard
Implicit Multiplication	`2π`, `3(4+5)`	Allow multiplication to be implied when a number is next to a parenthesized expression or constant.	**Impact:** Medium. Primarily affects the Parser's `parse_primary` or a new lookahead logic. **Considerations:** The lexer must still produce separate tokens (`NUMBER`, `IDENTIFIER`, `LPAREN`). The parser must detect sequences like `NUMBER IDENTIFIER` and insert an implicit `*` token (or directly build a `BinaryOp('*', ...)` node). Can be tricky to get right without breaking existing parsing.	🟡 Medium

ADR: Designing for Extensibility — Plugin Architecture for Functions

> **Decision: Use a Centralized BUILTINS Dictionary for Function Registration**
>
> - **Context**: The calculator needs to support an expanding set of built-in mathematical functions. We want to make adding new functions trivial without modifying core evaluation logic.
> - **Options Considered**:
>   1. **Hardcoded `if-elif` chain in `evaluate()`** : Check function name against known strings and call corresponding math library functions.
>   2. **Centralized `BUILTINS` dictionary**: Map function name strings to Python callable objects (e.g., `{'sqrt': math.sqrt}`). The evaluator looks up the name in this dict.
>   3. **Plugin module discovery**: Dynamically import functions from a `functions/` directory, allowing third-party extensions.
> - **Decision**: **Option 2 (Centralized BUILTINS dictionary)**.
> - **Rationale**: This provides a clean separation between the evaluation engine and the function implementations. Adding a new function is a one-line addition to the dictionary. It's more maintainable and testable than a sprawling `if-elif` chain (Option 1). Option 3 is overkill for a beginner project and introduces complexity (dynamic loading, error handling) without sufficient benefit at this stage.
> - **Consequences**: New functions must be explicitly registered. The dictionary approach naturally accommodates future extension to user-defined functions (which could be stored in a separate, user-accessible dictionary).

Option	Pros	Cons	Suitability for Learning
Hardcoded `if-elif` chain	Simple to understand; direct mapping.	Becomes unwieldy with many functions; mixes evaluation logic with function definitions; harder to test in isolation.	Low — teaches poor separation of concerns.
Centralized `BUILTINS` dictionary	Clean separation; easy to extend; functions are data, not logic; facilitates testing (mock functions).	Slightly more abstract for beginners; requires understanding of first-class functions.	**High** — introduces a powerful pattern (registry).
Plugin module discovery	Highly extensible; supports third-party contributions; advanced architecture pattern.	Significant complexity; requires understanding of modules, reflection, error handling; over-engineering for a calculator.	Low — too advanced for core project.

Common Pitfalls When Extending

⚠️ Pitfall: Breaking Backwards Compatibility

- **Description**: Adding a new operator (e.g., `%`) without carefully considering its precedence relative to existing operators can change the meaning of existing valid expressions. For example, if `%` is placed at the wrong precedence level, `5 + 3 % 2` might evaluate differently than standard mathematical convention.
- **Why It's Wrong**: It introduces subtle bugs for any existing tests or user expressions that coincidentally contain the new operator's characters (though unlikely, they could exist). More importantly, it violates the **principle of least surprise**.
- **How to Fix**: Before implementing, research the standard precedence for the new operator in established languages (C, Python). Write comprehensive tests for expressions mixing the new operator with existing ones to verify precedence and associativity.

⚠️ Pitfall: Ignoring Error Semantics of New Features

- **Description**: Adding a new function like `log(x)` without defining what happens for `x <= 0` leads to inconsistent behavior — it might crash with a Python `ValueError` or return `NaN`, confusing users.
- **Why It's Wrong**: A robust calculator should provide clear, consistent error messages. Letting underlying library errors bubble up creates a poor user experience and makes debugging harder.
- **How to Fix**: For each new operation or function, define its domain and error conditions explicitly. Wrap the core math call in a `try-except` block in the evaluator and raise a descriptive `RuntimeError` (e.g., `"log domain error: argument must be positive"`).

⚠️ Pitfall: Ad-Hoc Parser Modifications

- **Description**: When adding a new operator like `//`, one might be tempted to hack it into `parse_factor` with a quick `elif` without updating the formal mental model of the grammar.
- **Why It's Wrong**: This makes the parser harder to understand and more brittle. Future extensions become increasingly difficult as the ad-hoc patches accumulate.
- **How to Fix**: Always update the **mental model** of your grammar first. Sketch the new precedence table on paper. Then, methodically adjust the relevant parsing method(s), ensuring the new operator is integrated at the correct precedence level and with the correct associativity. Add clear comments in the code referencing the updated precedence table.

Implementation Guidance

> **Note**: This section provides starter code and structure for *some* of the easier extensions. For complex extensions like user-defined functions, the implementation would be a project in itself and is beyond the scope of this guidance.

A. Technology Recommendations Table

Component	Simple Option (for learning)	Advanced Option (for robustness)
New Operators	Add token and extend existing parser method (e.g., `parse_factor`).	Define a formal grammar file and use a parser generator (like `ply` in Python).
REPL Interface	Simple `while` loop with `input()` and `print()`.	Use `readline` library for history, autocompletion, and multi-line editing.
Persistent Variables	Save as plain text `name=value` lines.	Use JSON serialization for structured data and error resilience.
Constants	Hardcode in `Environment` initialization.	Create a `Constants` class that inherits from `Environment` and overrides `set` to disallow modification.

B. Recommended File/Module Structure for Extended Project

```
calculator/
├── __main__.py             # Entry point, could launch REPL
├── lexer.py                # Extended with new token types
├── parser.py               # Extended with new grammar methods
├── ast_nodes.py            # Extended with new node classes (e.g., ComparisonOp)
├── evaluator.py            # Extended BUILTINS and evaluation cases
├── environment.py          # Enhanced with constants, nested scopes
├── functions/              # NEW: Directory for plugin functions (advanced)
│   ├── __init__.py
│   ├── trigonometric.py
│   └── statistical.py
├── repl.py                 # NEW: REPL driver
└── tests/                  # Extended test suites
    ├── test_lexer_extended.py
    └── test_comparison_operators.py
```

C. Infrastructure Starter Code: Simple REPL Driver

Create a new file `repl.py` with a complete, working REPL that integrates the existing components.

PYTHON

```python
# repl.py

import sys

from lexer import Lexer, LexerError

from parser import Parser, SyntaxError

from evaluator import Evaluator, SemanticError, RuntimeError

from environment import Environment


class CalculatorREPL:

    """A simple Read-Eval-Print Loop for the calculator."""


    def __init__(self):

        self.env = Environment()

        self.lexer = Lexer()

        self.parser = Parser()


    def run(self):

        print("Calculator REPL (type 'quit' to exit)")

        while True:

            try:

                # Read

                line = input(">> ").strip()

                if line.lower() in ('quit', 'exit', 'q'):

                    break

                if not line:

                    continue


                # Evaluate

                result = self._evaluate_line(line)


                # Print

                if result is not None:

                    print(result)


            except (LexerError, SyntaxError, SemanticError, RuntimeError) as e:

                print(f"Error: {e}")

            except EOFError:

                print("\nExiting...")

                break

            except KeyboardInterrupt:

                print("\nInterrupted. Use 'quit' to exit.")
```

```
    def _evaluate_line(self, line: str):
        """Evaluate a single line of input."""
        tokens = self.lexer.tokenize(line)
        ast = self.parser.parse(tokens)
        return Evaluator.evaluate(ast, self.env)

if __name__ == "__main__":
    repl = CalculatorREPL()
    repl.run()
```

D. Core Logic Skeleton: Adding Modulo and Floor Division

Extend the `lexer.py`, `parser.py`, and `evaluator.py` to support `%` and `//`.

In `lexer.py` (add to token types and scanning logic):

PYTHON

```
# In constants section
MODULO = 'MODULO'
FLOOR_DIV = 'FLOOR_DIV'

# In Lexer.tokenize method's character scanning loop
# TODO: Add cases for '%' and '/' followed by another '/'
# Example for '%':
# elif char == '%':
#     tokens.append(Token(MODULO, '%', self.pos))
#     self.pos += 1
```

In `parser.py` (extend `parse_factor` method):

PYTHON

```
def parse_factor(self) -> ASTNode:
    """Parses multiplication, division, modulo, and floor division."""
    node = self.parse_unary()

    while self.match(STAR, SLASH, MODULO, FLOOR_DIV):
        operator = self.previous_token().type
        right = self.parse_unary()
        node = BinaryOp(operator, node, right)

    return node
```

In `evaluator.py` (extend `evaluate` function for `BinaryOp`):

```python
# In the BinaryOp evaluation section
# TODO: Add cases for MODULO and FLOOR_DIV
# Example for MODULO:
# elif node.operator == MODULO:
#     right_val = evaluate(node.right, env)
#     if right_val == 0:
#         raise RuntimeError(f"Modulo by zero at position {node.position}")
#     return evaluate(node.left, env) % right_val
# For FLOOR_DIV:
# elif node.operator == FLOOR_DIV:
#     right_val = evaluate(node.right, env)
#     if right_val == 0:
#         raise RuntimeError(f"Floor division by zero at position {node.position}")
#     return evaluate(node.left, env) // right_val
```

E. Language-Specific Hints (Python)

- **Math Functions**: Use `math.tan`, `math.log`, `math.exp`. Remember `math.log` is natural log; for log base 10, use `math.log10`.
- **Constants**: Define `pi` and `e` in the `Environment.__init__` as `self.set('pi', math.pi)`. Consider making them read-only by overriding `Environment.set` to raise an error if trying to reassign a constant.
- **Comparison Operators**: Python's `operator` module (`import operator`) provides functions like `operator.gt`, `operator.lt` that can be used in a dispatch table, similar to `BUILTINS`.
- **REPL History**: For a better REPL, install the `readline` module on Unix/macOS (`pip install gnureadline` on macOS) or use `pyreadline3` on Windows.

F. Milestone Checkpoint for Extended Features

After implementing a new feature (e.g., modulo and floor division):

1. **Run the extended test suite**:

```bash
python -m pytest tests/ -v
```

2. **Manual verification in the REPL**:

```bash
python -m calculator.repl
```

Test expressions:

- `10 % 3` should output `1`
- `10 // 3` should output `3`
- `5 + 3 % 2` should output `6` (since `%` has same precedence as `*` and `/`, it binds tighter than `+`)
- `10 % 0` should show a clear "Modulo by zero" error.

3. **Signs something is wrong**:
   - `5 + 3 % 2` gives `4`: This means `%` has lower precedence than `+` — check it's in `parse_factor`.
   - `10 // 3` gives `3.3333`: You're using float division (`/`) instead of floor division — check token type and evaluation logic.
   - `//` token not recognized: Check lexer scanning for two consecutive `/` characters.

G. Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
New operator works alone but not mixed with others	Wrong precedence level in parser.	Use `debug_print_ast` on expression `2 + 3 * 4 % 2`. See if `%` is grouped with `*` (correct) or with `+` (wrong).	Move operator to correct parsing method (e.g., `parse_factor` vs `parse_term`).
New function `log(x)` crashes with Python error	Missing error handling for domain.	Test `log(-1)` and see if a `ValueError: math domain error` bubbles up.	Wrap the `math.log` call in `try-except` and raise a descriptive `RuntimeError`.
REPL doesn't remember variables between lines	Creating a new `Environment` for each evaluation.	Check if your REPL creates a new `Environment` inside `_evaluate_line`.	Move `self.env = Environment()` to `__init__` and reuse it.
Constants like `pi` can be reassigned	No protection against overwriting.	In REPL, try `pi = 5`. If it succeeds, constants are mutable.	Override `Environment.set` to check against a `_constants` set and raise `SemanticError`.

13. Glossary

> **Milestone(s):** All Milestones (1-3) — The glossary serves as a central reference for all technical terminology used throughout the design document. Understanding these terms is essential for comprehending the architectural decisions, implementation details, and testing strategies described in previous sections.

Terms and Definitions

This glossary provides clear definitions for key terms used in the calculator parser project, arranged alphabetically. Each entry includes the primary section where the concept is first introduced in depth.

Term	Definition	Introduced In
Abstract Syntax Tree (AST)	A hierarchical, tree-shaped data structure that represents the grammatical structure of an expression, with operators as internal nodes and operands (numbers, variables) as leaves. It is "abstract" because it omits syntactic details like parentheses and whitespace, focusing solely on the expression's meaning. The `ASTNode` class hierarchy (`Number`, `BinaryOp`, `UnaryOp`, etc.) defines the structure of this tree.	Section 4: Data Model
Associativity	The direction in which operators of the same precedence are evaluated when they appear consecutively in an expression. *Left-associative* operators (e.g., `+`, `-`, `*`, `/`) group from left to right (`8 / 4 / 2` equals `(8 / 4) / 2`). *Right-associative* operators (e.g., `^` for exponentiation) group from right to left (`2 ^ 3 ^ 2` equals `2 ^ (3 ^ 2)`).	Section 1: Context and Problem Statement
Binary Operator	An operator that takes two operands, such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`^`). In the AST, a binary operator is represented by a `BinaryOp` node with `left` and `right` child nodes.	Section 4: Data Model
Built-in Function	A predefined mathematical function, such as `sin`, `cos`, or `sqrt`, that is natively provided by the calculator. These functions are called by name followed by parentheses containing their argument (e.g., `sqrt(16)`). The `BUILTINS` dictionary in the `Evaluator` maps function names to their Python implementations.	Section 7: Component Design: Evaluator
Environment (Symbol Table)	A data structure, implemented as a dictionary, that maintains a mapping between variable names (identifiers) and their stored numeric values during the evaluation phase. The `Environment` class provides `get()` and `set()` methods for variable lookup and assignment, respectively.	Section 4: Data Model
Error Recovery	A parsing strategy where, after detecting a syntax error, the parser attempts to synchronize its state and continue parsing to find additional errors. For this beginner project, error recovery is an advanced technique and is not required; the parser can stop and report the first error found.	Section 9: Error Handling and Edge Cases
Expression	A valid combination of numbers, variables, operators, and function calls that can be evaluated to produce a single numeric value. Examples include `5`, `x + 2`, and `3 * sin(0.5)`. The parser's job is to transform a string representation of an expression into an AST.	Section 1: Context and Problem Statement
Grammar	A formal set of rules that defines the syntactic structure of all valid expressions in the calculator's language. It specifies how different language elements (numbers, operators, parentheses) can be combined. The recursive descent parser's method structure (`parse_expression`, `parse_term`, etc.) directly mirrors these grammar rules.	Section 6: Component Design: Parser
Greedy Scanning	The lexer's strategy of consuming as many consecutive characters as possible to form a single valid token. For example, when encountering the character sequence "123.45", the lexer will consume all characters to form a single `NUMBER` token, rather than stopping after "123".	Section 5: Component Design: Lexer (Tokenizer)
Intermediate Representation (IR)	A data structure, such as the AST, that serves as a bridge between the source code (input string) and the final execution. It decouples parsing from evaluation, allowing for analysis, optimization, or alternative execution strategies without modifying the parser.	Section 3: High-Level Architecture
Lexeme	The exact sequence of characters from the input source that forms a single token. For example, in the input "total = 42", the lexemes are "total", "=", and "42". The `lexeme` field of the `Token` data structure stores this raw string.	Section 4: Data Model
Lexer (Tokenizer/Scanner)	The component responsible for the first stage of processing: reading the raw input string character by character and grouping them into a sequence of meaningful **tokens**. It strips whitespace and identifies the fundamental building blocks (like numbers, operators, and identifiers) for the parser.	Section 5: Component Design: Lexer (Tokenizer)
Lexical Error	An error that occurs during the lexing phase when the scanner encounters a sequence of characters it cannot interpret as any valid token (e.g., `@` or `$` in the calculator). The `Lexer` raises a `LexerError` with the invalid character and its position.	Section 9: Error Handling and Edge Cases
Lookahead	A parsing technique where the parser examines the next token(s) in the stream without consuming it, to decide which grammar rule to apply. Methods like `Parser.peek()` provide this capability, which is essential for parsing constructs like function calls (is this `IDENTIFIER` a variable or a function name?).	Section 6: Component Design: Parser
Parser	The component that consumes the stream of **tokens** produced by the lexer and, according to the rules of the **grammar**, builds an **Abstract Syntax Tree (AST)**. It is responsible for enforcing **operator precedence** and **associativity**.	Section 6: Component Design: Parser
Position Tracking	The practice of recording the character index (or line/column numbers) where a token or error occurs in the original input string. This information is stored in the `position` field of `Token` and error objects, enabling the generation of user-friendly error messages that point to the exact location of a problem.	Section 9: Error Handling and Edge Cases
Precedence (Order of Operations)	A ranking system that determines which operators are evaluated first in an expression containing multiple operators. Operators with higher precedence bind more tightly to their operands. In this calculator, the precedence hierarchy (from highest to lowest) is: Parentheses → Function Calls → Exponentiation → Unary Plus/Minus → Multiplication/Division → Addition/Subtraction.	Section 1: Context and Problem Statement

Term	Definition	Introduced In
REPL (Read-Eval-Print Loop)	An interactive programming environment that repeatedly (1) Reads a user's input, (2) Evaluates it, and (3) Prints the result, forming a loop. While not a required deliverable for the core milestones, it is a common and useful extension for the calculator.	Section 12: Future Extensions
Recursive Descent	A top-down **parsing** technique where the grammar is directly translated into a set of mutually recursive functions (e.g., `parse_expression`, `parse_term`). Each function is responsible for parsing a specific grammatical construct. It is chosen for this project for its clarity and beginner-friendliness.	Section 6: Component Design: Parser
Regression Test	A test that is added to a test suite to ensure that a bug, once fixed, does not re-appear in future modifications to the code. It protects against unintended side effects when new features are added.	Section 10: Testing Strategy
Runtime Error	An error that occurs during the **evaluation** phase when a computation is mathematically invalid or impossible, such as division by zero or calling `sqrt()` on a negative number (if not handling complex numbers). The `Evaluator` raises a `RuntimeError`.	Section 9: Error Handling and Edge Cases
Semantic Error	An error in the *meaning* of an otherwise syntactically correct expression. The primary example in this project is referencing an **undefined variable** (e.g., using `y` before `y = 5`). The `Evaluator` detects this and raises a `SemanticError`.	Section 9: Error Handling and Edge Cases
Separation of Concerns	A design principle where a system is divided into distinct components, each responsible for a specific piece of functionality. In this project, it manifests as the clear division between the **Lexer** (tokenizing), **Parser** (structure building), and **Evaluator** (computation) components.	Section 3: High-Level Architecture
Side Effect	An operation during **evaluation** that modifies the state of the **Environment**, specifically the assignment of a value to a variable (e.g., `x = 5`). This contrasts with pure expression evaluation, which only computes a value without changing state.	Section 7: Component Design: Evaluator
Symbol Table	See **Environment**.	Section 4: Data Model
Syntax Error	An error in the grammatical structure of an expression, such as mismatched parentheses (`(2 + 3`), a missing operator (`2 3`), or an unexpected token. The **Parser** detects these errors and raises a `SyntaxError`.	Section 9: Error Handling and Edge Cases
Token	A meaningful, atomic unit of the language, produced by the **Lexer**. Each token has a `type` (e.g., `NUMBER`, `PLUS`), a `lexeme` (the raw text), and a `position` (its location in the input). The token stream is the parser's input.	Section 4: Data Model
Tree Walk	The recursive process by which the **Evaluator** traverses the **AST**, visiting each node to perform its operation. It starts at the root and recursively **evaluates** child nodes to obtain operand values before applying the operation at each parent node.	Section 7: Component Design: Evaluator
Unary Operator	An operator that takes a single operand. In this calculator, the unary minus (`-`) is the primary example, which negates the value of the expression that follows it (e.g., `-5` or `-(2 + 3)`). It is represented in the AST by a `UnaryOp` node.	Section 4: Data Model
Whitespace	Characters such as spaces (` `), tabs (`\t`), and newlines (`\n`) that are used to separate tokens in the input but are otherwise insignificant to the expression's meaning. The **Lexer** skips over all whitespace during **tokenization**.	Section 5: Component Design: Lexer (Tokenizer)
