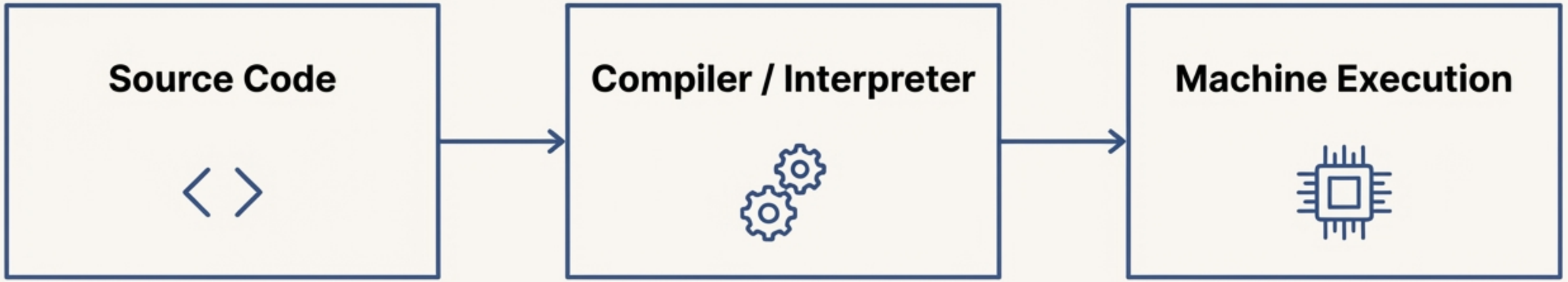
From Text to Action

The Life of a Line of Code

Every Program is a Translation

At its core, a compiler or interpreter is a specialized program that translates source code written in a high-level, human-readable language into a representation that a computer can execute. It bridges the gap between how we write code and how the machine runs it.



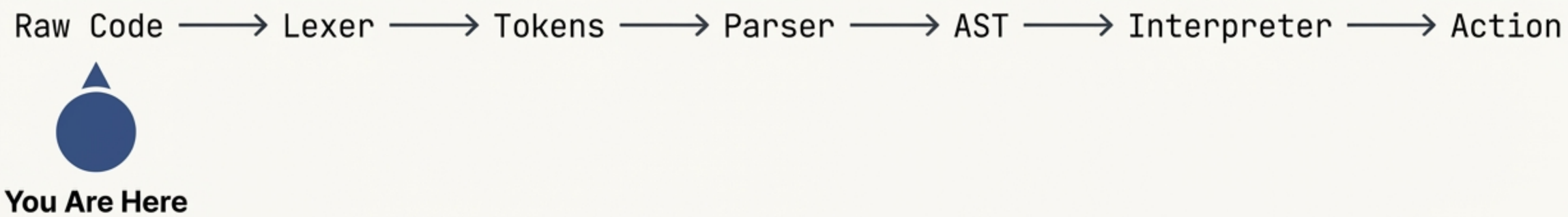
Meet Our Line of Code

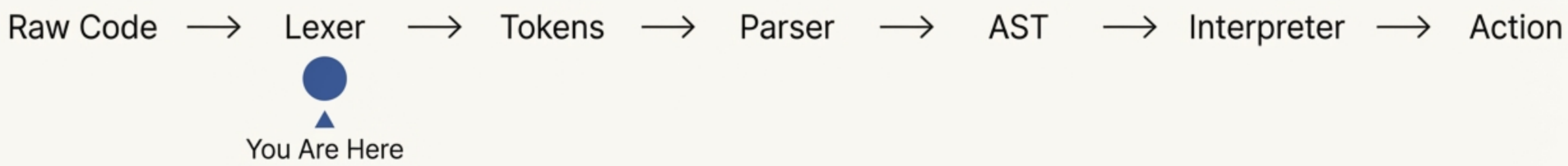
To understand the journey from text to action, we will follow a single piece of source code from start to finish.

```
let result = 5 + 3;
```

The Path from Code to Execution

Our code will pass through a series of distinct stages, each transforming it into a more structured and executable form.





Stage 1: The Lexer Breaks Down the Code

Raw String

```
"let result = 5 + 3;"
```

Lexical Analysis

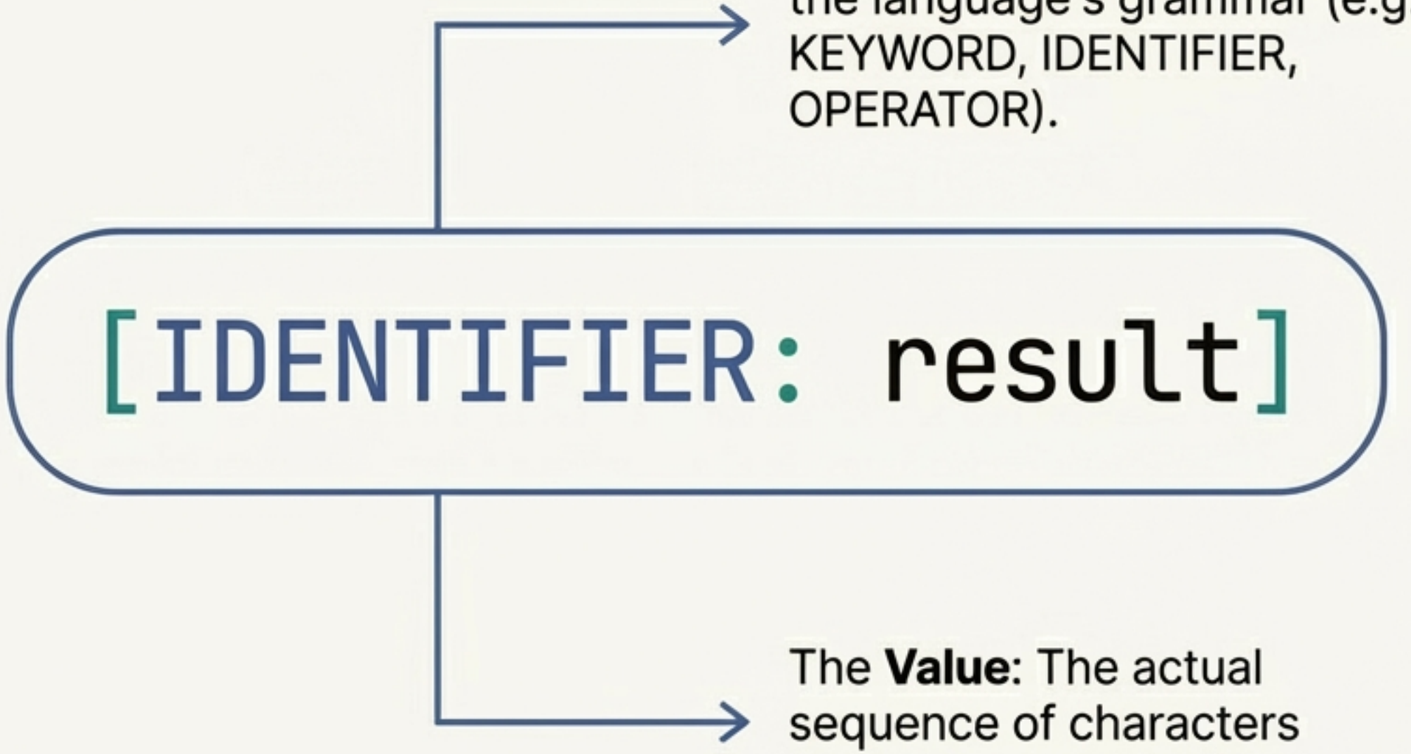
The Lexer, or Tokenizer, scans the input string. It identifies sequences of characters that represent keywords, identifiers, and operators, while discarding non-essential elements like whitespace.

A Stream of Tokens

```
[KEYWORD: let]
[IDENTIFIER: result]
[OPERATOR: =]
[INTEGER_LITERAL: 5]
[OPERATOR: +]
[INTEGER_LITERAL: 3]
[PUNCTUATION: ;]
```

Tokens are the Vocabulary of Our Language

A token is the fundamental, indivisible unit of a programming language. It's a structured piece of data that categorizes a piece of source code, typically containing two parts: its type and its value.



Stage 2: The Parser Understands the Grammar

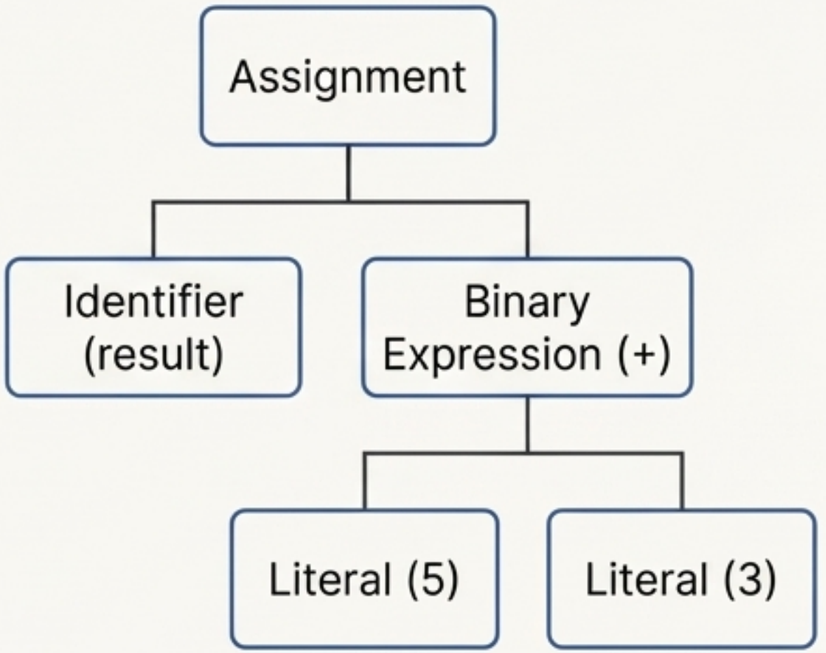
Stream of Tokens

```
[KEYWORD: let]
[IDENTIFIER: result]
[OPERATOR: =]
[INTEGER_LITERAL: 5]
[OPERATOR: +]
[INTEGER_LITERAL: 3]
[PUNCTUATION: ;]
```

Syntactic Analysis

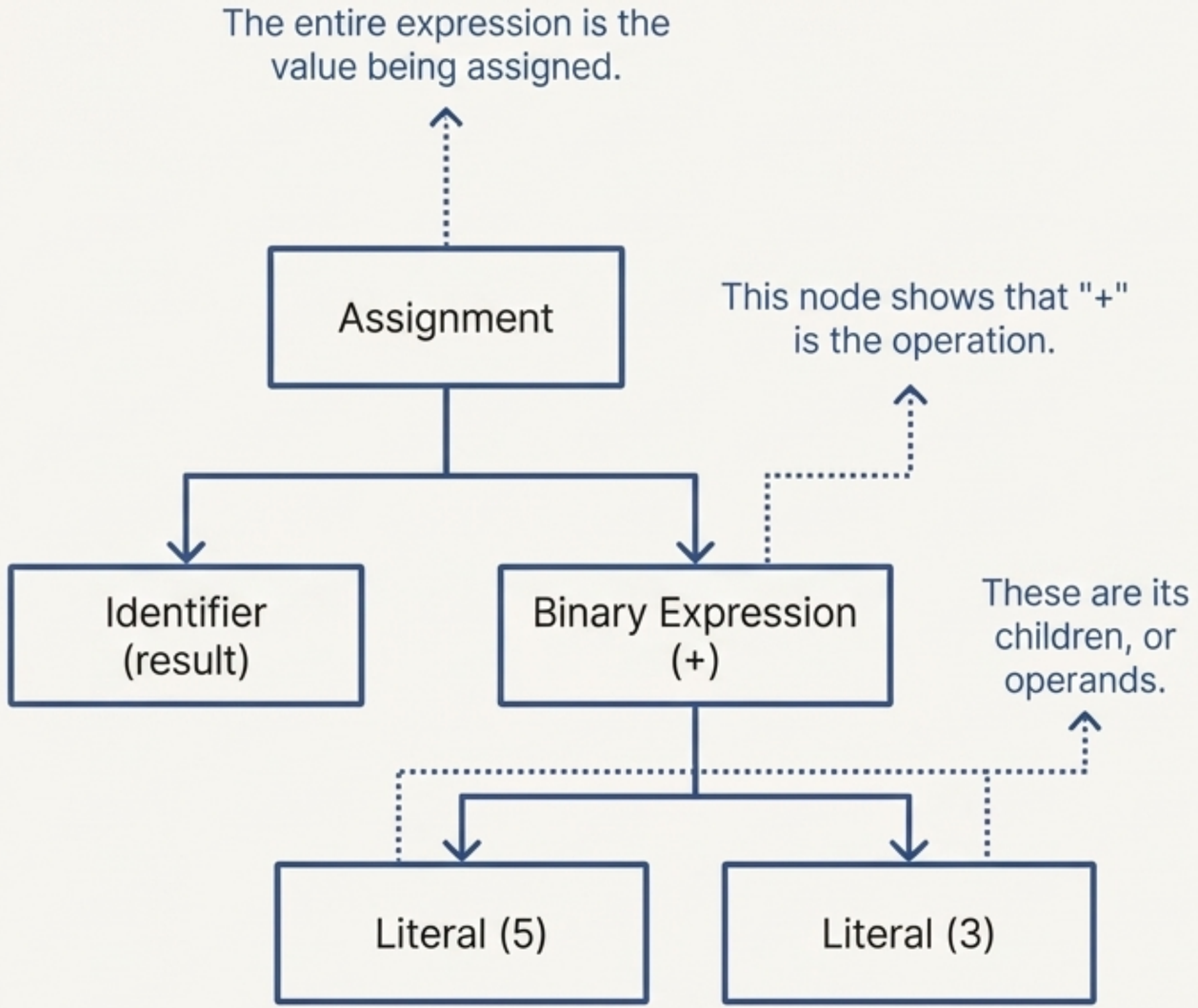
The Parser receives the stream of tokens and analyzes their sequence to ensure they conform to the language's grammatical rules (syntax). It then constructs a structured representation of the code.

The Abstract Syntax Tree (AST)



The AST Represents the Code's True Structure

The Abstract Syntax Tree (AST) is a hierarchical representation of the code. Unlike a flat list of tokens, it explicitly shows the relationships between operators and their operands, reflecting the program's intended logic and order of operations. It is the blueprint for execution.

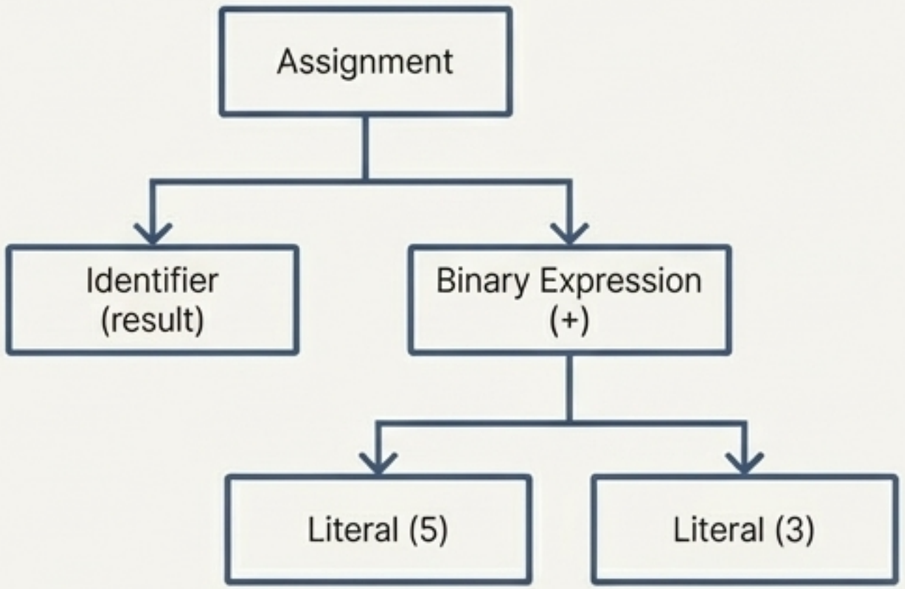


Raw Code → Lexer → Tokens → Parser → AST → **Interpreter** → Action

You Are Here

Stage 3: The Interpreter Takes Action

Abstract Syntax Tree (AST)



Execution

The Interpreter directly executes the instructions represented in the AST.

It traverses the tree, evaluating expressions and performing the operations specified by the program's logic.

The Result

Program Memory
`result = 8`

The expression `5 + 3` is evaluated to `8`, and this value is assigned to the identifier `result`.

From a String of Characters to an Executed Result

We have followed our line of code through its complete lifecycle. Each stage played a critical role in transforming it from ambiguous text into an explicit set of instructions that the computer could understand and execute.

Raw Code → Lexer → Tokens → Parser → AST → Interpreter → Action

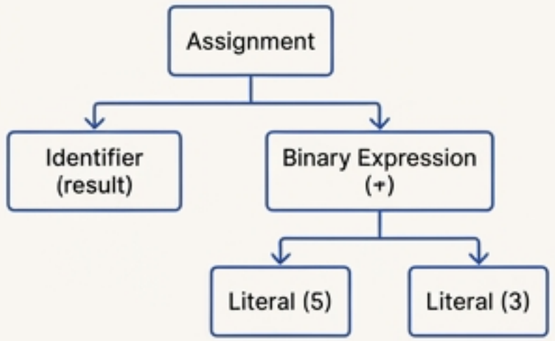
Input

```
let result = 5 + 3;
```

Lexer Output

`[KEYWORD: let]` `[IDENTIFIER: result]` `[OPERATOR: =]` ...

Parser Output



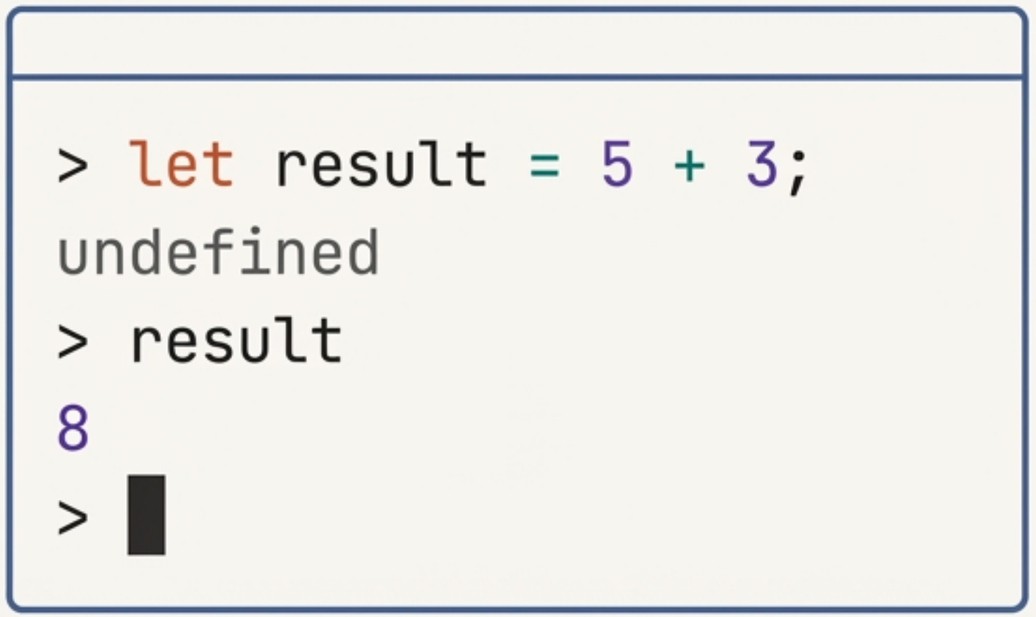
Interpreter Output

Program Memory

```
result = 8
```

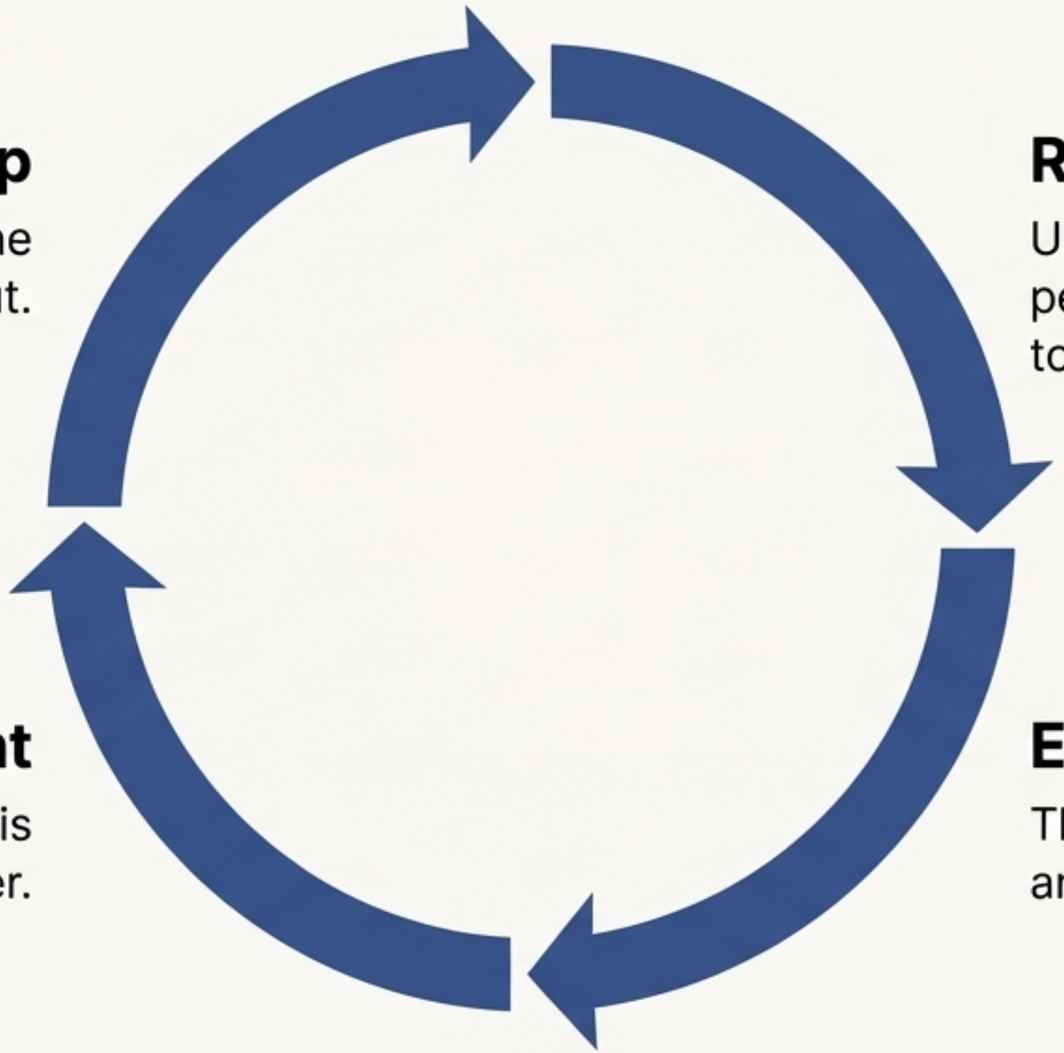
Putting It All Together: The REPL

This entire process happens instantly in common development tools. A **REPL**, or **Read-Eval-Print Loop**, is an interactive programming environment that uses these very steps to give you immediate feedback.



The Journey is the Loop

The REPL's name describes the exact process we've just explored.



Read

User types `5 + 3`. The system performs **Lexing** and **Parsing** to create an AST.

Eval

The **Interpreter** walks the AST and calculates the result, 8.

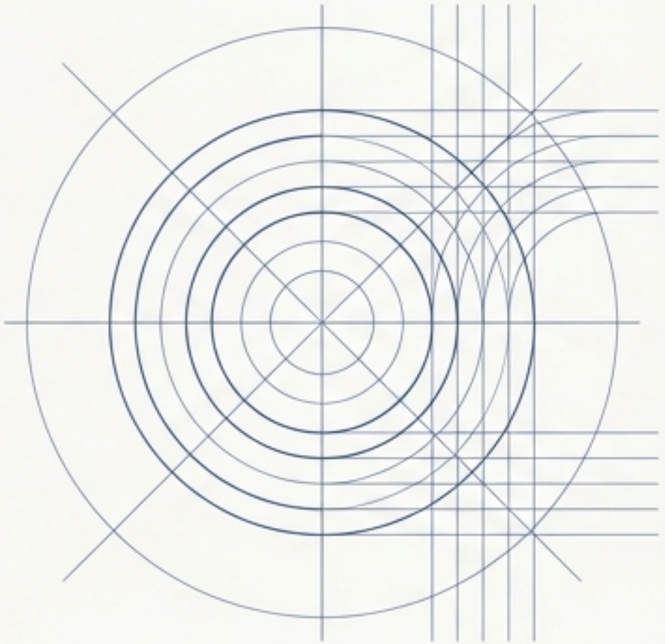
Print

The resulting value, 8, is displayed to the user.

Loop

The system waits for the next input.

Understanding the Journey Empowers the Developer



The transformation from text to action is the fundamental process that powers all software. A clear mental model of how code is understood by the machine is not just academic—it's essential for effective debugging, writing performant code, and even creating new programming languages. It is the machinery behind the magic.