The C Interpreter

From Static Syntax to Dynamic Execution

Team 02

Students:

320206102

316255819

423031180

320117174

320340312

Compilers Course | UNAM | December 2025

Project Goals: A Transparent Interpreter

We established technical goals to develop a robust and functional interpreter.

Objective

Build a functional C interpreter using Python & PLY.

Philosophy

"White Box" approach – making internal processes visible.

Scope

- Lexical & Syntactic Analysis (LALR 1).
- Static Semantic Validation (Type checking before running).
- Dynamic Execution (Recursive AST traversal).

System Architecture: Sequential Analysis Phases

The construction of an interpreter is based on fundamental principles of language theory, operational semantics, and execution environment design.

Lexical Analysis -> Tokens

Transforms source code into tokens. Identifies keywords, operators, and identifiers.

Syntactic Analysis -> AST

Constructs the Abstract Syntax Tree (AST). Verifies the grammatical structure of the code.

Execution -> Results

Recursive Traversal. The Interpreter walks the AST, simulating stack frames and memory state.

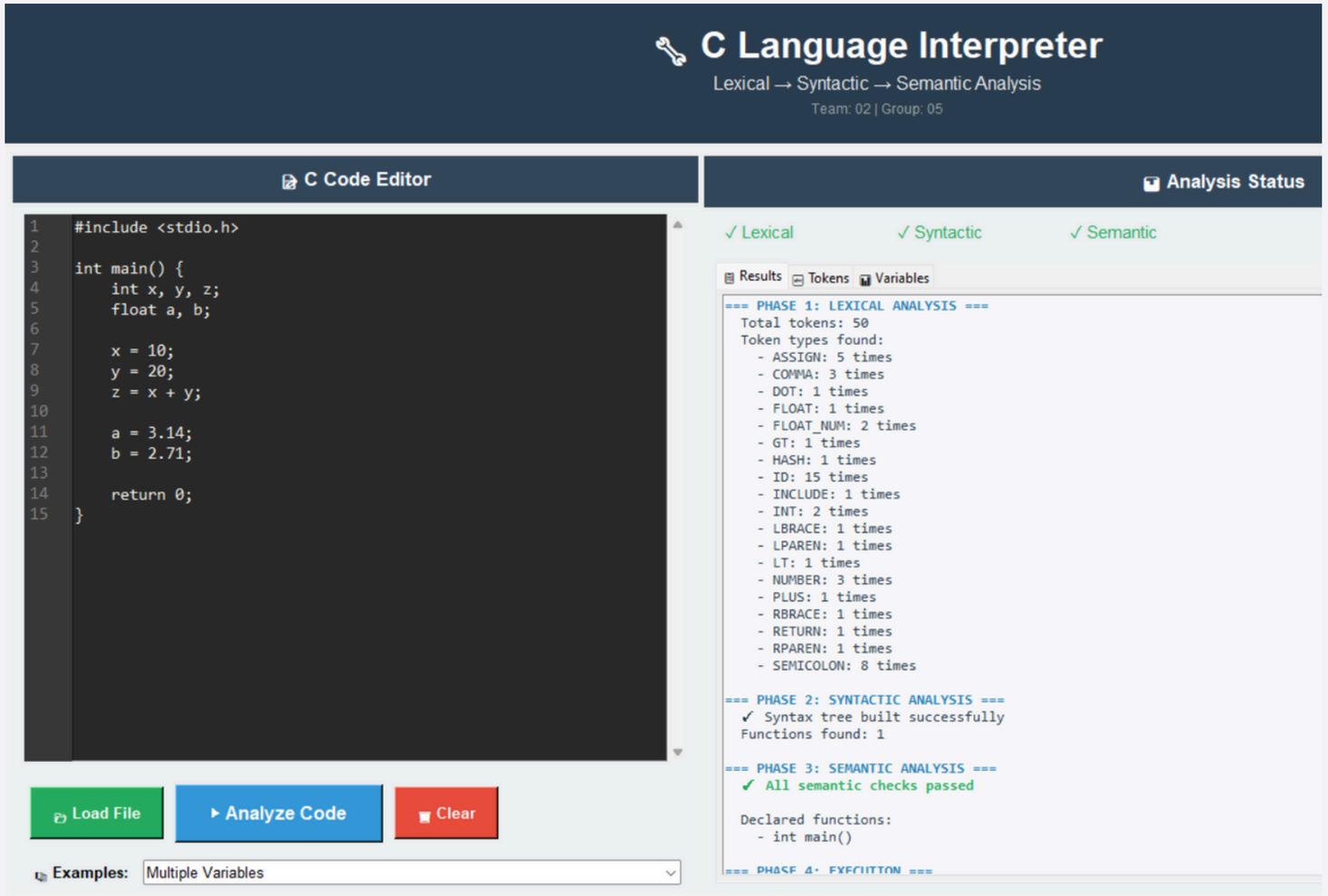
Semantic Analysis -> Validation

Validates types and manages the symbol table. Ensures the logical consistency of the program.

Implementation & Capabilities

User Interface

Real-time visualization of Tokens, AST status, and Memory.



Technologies Used

Python

PLY (Lex-Yacc)

Tkinter GUI

Key Capabilities

Functions & Recursion (Context switching).

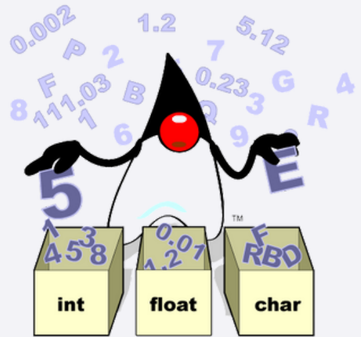
Data Structures: Arrays with bounds checking.

Control Flow: Nested loops (While/For) & Conditionals.

Deep Dive: Lexical Analysis

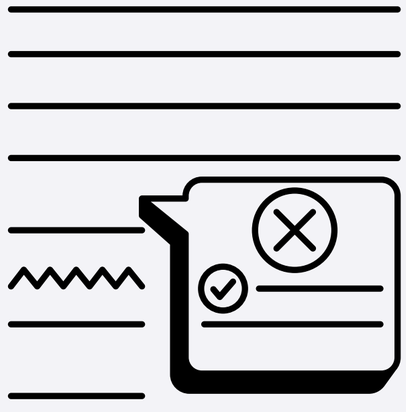
The Lexer (CLexer Class):

- Technology: Regex-based Tokenization.
- Responsibilities:
  i. Filtering: Removes comments (//, /* */) & whitespace.
  ii. Classification: Identifies Keywords vs. Identifiers.
  iii. Tracking: Line & Column counting for error reporting.



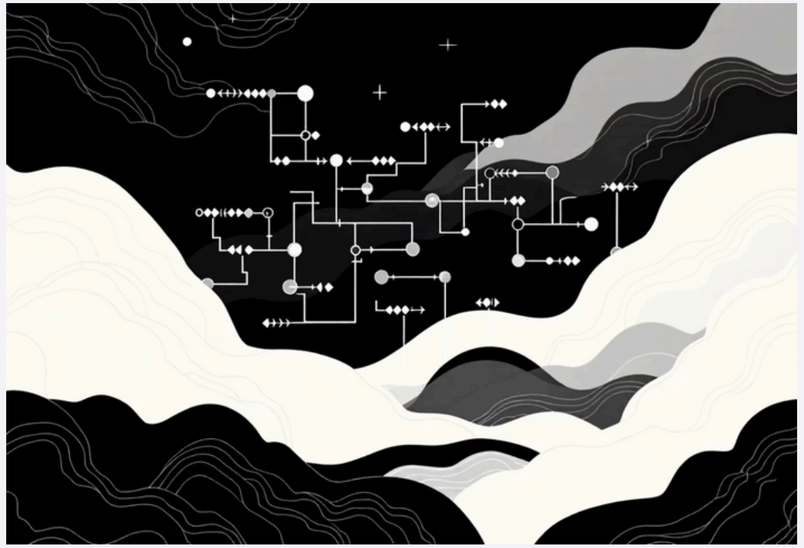
Deep Dive: Syntactic Analysis

- **The Parser (CParser Class):**
  - Algorithm: LALR(1) Bottom-Up Parsing.
  - Grammar: Context-Free Grammar (CFG) rules.
- **Precedence Handling:**
  - PEMDAS: Defined via precedence tuple.
  - Associativity: Left (+, -, *, /) vs. Right (!, - unary).
- **Output:**
  - Validated hierarchical structure.



Abstract Syntax Tree (AST): From Text to Objects

Plain text is linear. We need a hierarchical representation where each statement becomes a Python class to enable non-linear execution with jumps and branches.



Why AST?

- Non-linear evaluation
- Conditional jumps
- Branches and loops
- Value propagation

Node Types

- BinaryOpNode: Operations
- IfNode: Conditions
- WhileNode: Loops
- FunctionNode: Functions

This abstraction allows the engine to 'jump' directly to the correct branch, simulating how real processors execute conditional jumps.

Semantic Analysis: The Proactive Security Layer

The SemanticAnalyzer is our security layer. It performs a complete sweep of the AST before execution, intercepting errors before they cause failures.

Declaration

Are variables defined?

Prevents invalid references.

Typing

Are types compatible?

Validates logical operations.

Initialization

Are structures correct?

Ensures data consistency.

Integrity

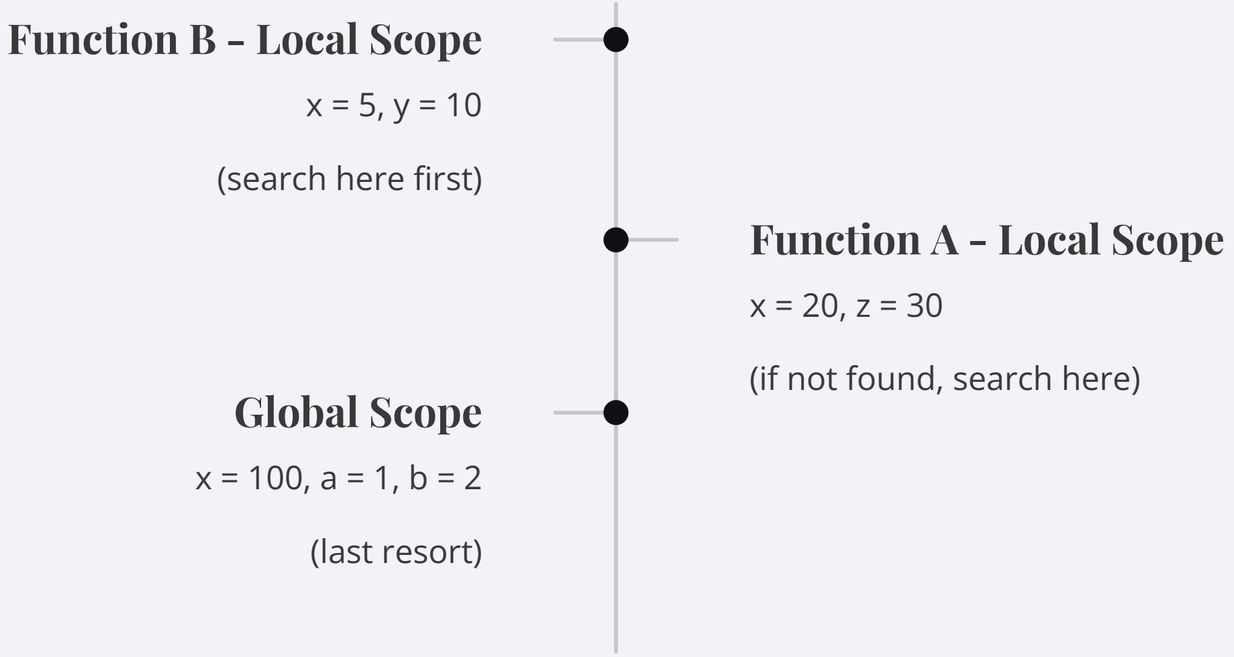
Proactive AST sweep.

Guarantees a stable environment.

If inconsistencies are detected, the error is immediately intercepted. This ensures that the engine never receives corrupted instructions.

Symbol Table: Hierarchical Memory Management

A dynamic stack of dictionaries representing each scope or execution context, enabling efficient resolution and robust isolation.



RESOLUTION

- Top-down search
- Shadowing allowed
- O(1) search time

LIFECYCLE

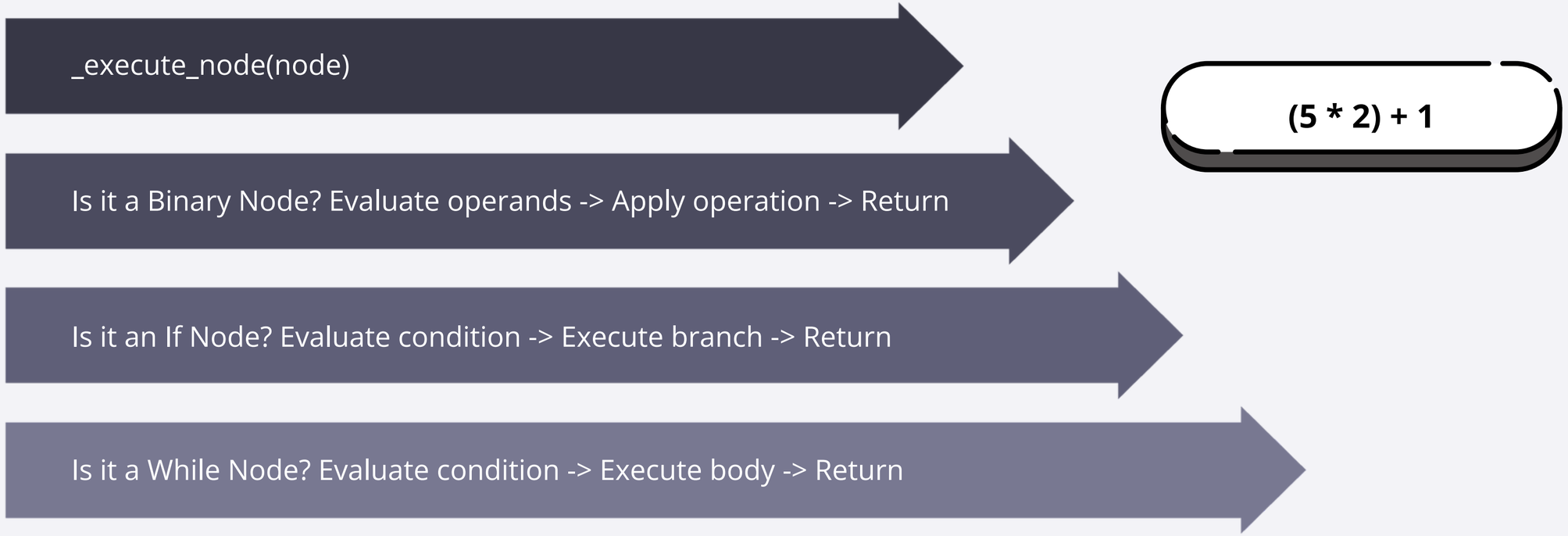
- Creation upon entry
- Destruction upon return
- Automatic cleanup

ADVANTAGES

- Variable isolation
- No collisions
- Automatic management

Execution Engine: The Heart of the Interpreter

It operates under a Recursive Dispatcher pattern. The _execute_node method receives a node, identifies its type, and delegates execution to the corresponding logic.



This value propagation mechanism allows support for arbitrarily complex nested expressions, ensuring correct operator precedence.

Functions and Stack Frames: Simulating the Call Stack

Full support for functions and recursion requires replicating the behavior of a real processor's call stack.

Save State

Stores the current global state before the call.

Create Scope

Generates a new, isolated local scope for the function.

Bind Arguments

Maps the call arguments to the local parameters.

Execute Body

Proceeds to execute the code within the function.

Restore State

Destroys the local scope and restores the previous state.

Each recursive call has its own private copy of variables, preventing corruption between instances. This allows for robust implementation of complex algorithms like factorial and fibonacci.

Core Engines: PLY & Tkinter

PLY (Python Lex-Yacc): The Engine

- **ply.lex**: Tokenizes input using regex rules (t_ID, t_NUMBER). Handles line/column tracking automatically.
- **ply.yacc**: Implements LALR(1) Parsing.
    - **Feature**: Uses function docstrings ('"program : declaration"') to define BNF Grammar rules.
    - **Output**: Triggers the creation of ASTNode objects.

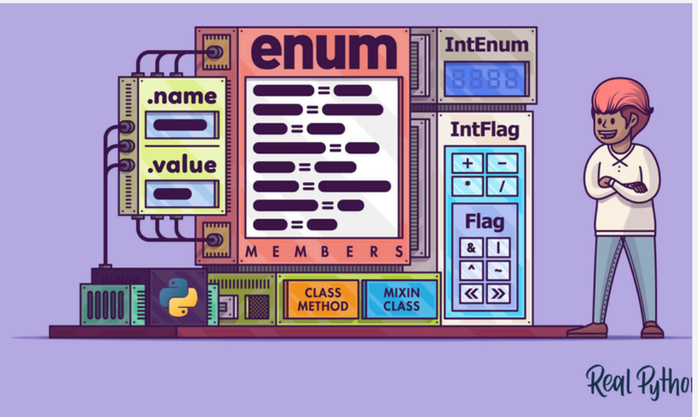
Tkinter: The Visualization Layer

- **ScrolledText**: Used for the Code Editor & Log Console.
- **ttk.Treeview**: Renders the Token Table and Variable Memory state.
- **Event Loop**: Manages the run_interpreter triggers.

Structural Integrity Libraries

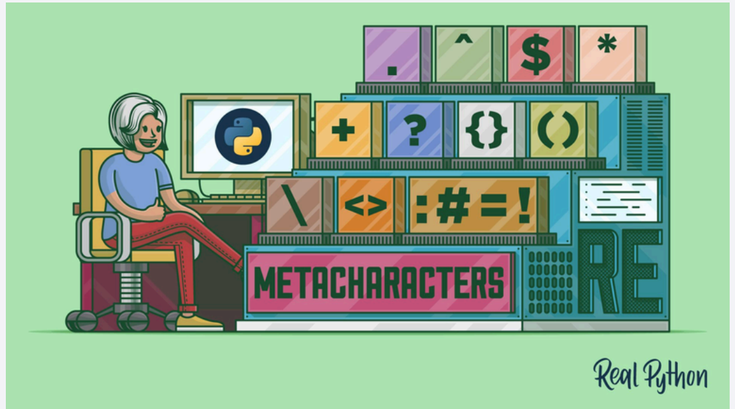
Enum: Strict Typing

- Usage: Definition of class NodeType(Enum)
- Benefit: Eliminates "magic strings"
- Ensures consistency across Parser and Interpreter



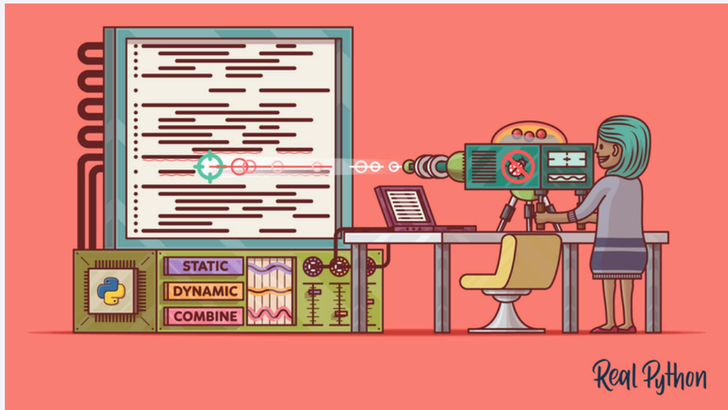
Re (Regular Expressions): Runtime Processing

- Usage: Inside _builtin_printf
- Dynamically injects Python values into format strings
- Simulates C's printf behavior accurately



Typing: Code Robustness

- Usage: Defining Symbol Table structures (Dict, List, Any)
- Benefit: Ensures consistency in memory management and scope stacking
- Catches type errors during development



Technical Challenges: Arrays and Non-Linear Flow Control

These are two critical engineering challenges, solved with solutions to ensure system stability and efficiency.

Challenge 1: Arrays and Strings

In C, an invalid memory access is fatal.

Solution:

- Arrays as contiguous Python Lists
- Index validation: 0 ≤ i < N
- Controlled exceptions
- Memory corruption prevention

Challenge 2: Non-Linear Flow Control

How does a deep break communicate to an outer loop that it should stop?

```
while (i < 10) {
    if (i == 5) {
        break;
    }
    i++;
}
```

Solution:

- Status flag system
- Signal propagation upwards in the tree
- Each parent level checks the flag
- Clean exit without corruption

User Interface: Engineering Cockpit

A complete graphical interface with Tkinter that functions as an engineering 'cockpit', allowing auditing of every step of the process.

Code Editor Panel

Dynamic line numbering

Syntax highlighting and navigation

File I/O operations

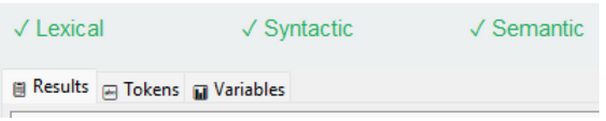
```c
#include <stdio.h>

int main() {
    int x, y, z;
    float a, b;

    x = 10;
    y = 20;
    z = x + y;

    a = 3.14;
    b = 2.71;

    return 0;
}
```

Multi-Tab Output View

Results Tab: Chronological phase-by-phase

Tokens Tab: TreeView table

Variables Tab

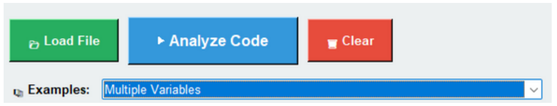


Control Dashboard

Analyze button

Example selector dropdown

Clear/Reset functionality



"Fail Loudly" Philosophy

Indicates exact error phase

Explains technical reason

Highlights the offending line

If the user makes an error, the system doesn't just report it; it explains it.

C Language Interpreter - Lexical, Syntactic, and Semantic Analysis

C Language Interpreter

Lexical → Syntactic → Semantic Analysis

Team: 02 | Group: 05

C Code Editor

```c
#include <stdio.h>

int main() {
    int counter;
    counter = 0;

    while (counter < 5) {
        printf("Counter: %d", counter);
        counter++;
    }

    return 0;
}
```

Load File | Analyze Code | Clear

Examples: While Loop

Analysis Status

✓ Lexical ✓ Syntactic ✓ Semantic

Results | Tokens | Variables

```
=== PHASE 2: SYNTACTIC ANALYSIS ===
  ✓ Syntax tree built successfully
  Functions found: 1

=== PHASE 3: SEMANTIC ANALYSIS ===
  ✓ All semantic checks passed

  Declared functions:
    - int main()

=== PHASE 4: EXECUTION ===
  Execution Output:

Executing main() function
    Declaration: int counter = 0
    Assignment: counter = 0
    printf: Counter: 0
    Increment: counter++ = 1
    printf: Counter: 1
    Increment: counter++ = 2
    printf: Counter: 2
    Increment: counter++ = 3
    printf: Counter: 3
    Increment: counter++ = 4
    printf: Counter: 4
    Increment: counter++ = 5
    WHILE: Executed 5 iterations
    Return: 0

  Return Value: 0

=== ANALYSIS AND EXECUTION COMPLETED ===
✓ Lexical: OK
✓ Syntactic: OK
✓ Semantic: OK
✓ Execution: OK
```

This interpreter analyzes C code using PLY (Python Lex-Yacc)

Results: Detailed Execution Phases

We executed a complete test program through all interpreter phases, demonstrating pipeline integrity.

Phase 1: Lexical Analysis

- 41 tokens identified
- Regular expressions validated
- Special symbols processed

```
PHASE 1: LEXICAL ANALYSIS
tal tokens: 41
```

```
Token types found:
  - ASSIGN: 1 times
  - COMMA: 1 times
  - DOT: 1 times
  - GT: 1 times
  - HASH: 1 times
  - ID: 9 times
  - INCLUDE: 1 times
  - INCREMENT: 1 times
  - INT: 2 times
  - LBRACE: 2 times
  - LPAREN: 3 times
  - LT: 2 times
  - NUMBER: 3 times
  - RBRACE: 2 times
  - RETURN: 1 times
  - RPAREN: 3 times
  - SEMICOLON: 5 times
  - STRING_LITERAL: 1 times
  - WHILE: 1 times
```

Phase 2: Syntactic Analysis

```
=== PHASE 2: SYNTACTIC ANALYSIS
===
  ✓ Syntax tree built
successfully
  Functions found: 1
```

- Syntax tree constructed
- Grammar structure validated
- 1 function detected

Phase 3: Semantic Analysis

- Successful checks
- Symbol table populated
- main() function declared

```
=== PHASE 3: SEMANTIC ANALYSIS ===
  ✓ All semantic checks passed

  Declared functions:
    - int main()
```

Phase 4: Execution

- Variables assigned correctly
- Successful value return (0)

```
=== PHASE 4: EXECUTION ===
  Execution Output:

Executing main() function
    Declaration: int counter = 0
    Assignment: counter = 0
    printf: Counter: 0
    Increment: counter++ = 1
    printf: Counter: 1
    Increment: counter++ = 2
    printf: Counter: 2
    Increment: counter++ = 3
    printf: Counter: 3
    Increment: counter++ = 4
    printf: Counter: 4
    Increment: counter++ = 5
    WHILE: Executed 5 iterations
    Return: 0

  Return Value: 0
```

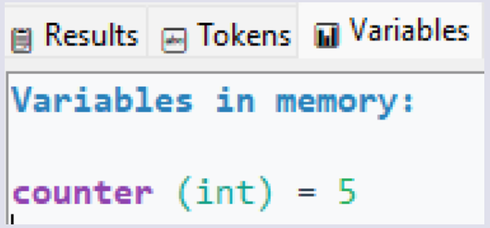
The process completed successfully, showing positive signs at each stage, demonstrating the robustness of the integrated pipeline.

Tokens & Variable Section

#	Token Type	Frequency	Example Values
1	ASSIGN	1	=
2	COMMA	1	,
3	DOT	1	.
4	GT	1	>
5	HASH	1	#
6	ID	9	stdio, h, main, counter, counter...
7	INCLUDE	1	include
8	INCREMENT	1	++
9	INT	2	int, int
10	LBRACE	2	{, {
11	LPAREN	3	(, (, (
12	LT	2	<, <
13	NUMBER	3	0, 5, 0
14	RBRACE	2	}, }
15	RETURN	1	return
16	RPAREN	3	),),)
17	SEMICOLON	5	;, ;, ;, ;, ;
18	STRING_LITERAL	1	Counter: %d
19	WHILE	1	while

Variables tab: You can inspect the final state of the execution, showing a list of Variables in memory.

Tokens tab: You can view a comprehensive table that categorizes the lexical analysis, displaying columns for Token Type, Frequency, and Example Values for every element detected, such as ID, NUMBER, and ASSIGN



Error Case

C Code Editor

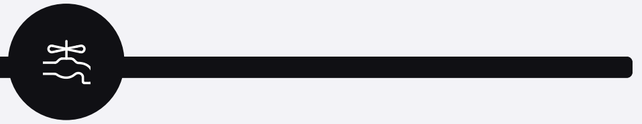
```c
#include <stdio.h>
int main() {
    int counter;
    // ERROR SYNTAC ERROR: A ";" is missing
    counter = 0
    while (counter < 5) {
        printf("Counter: %d", counter);
        counter++;
    }
    return 0;
}
```

A **“;”** after **counter = 0** is missing, so the system will detect a **syntax error**

```
=== PHASE 1: LEXICAL ANALYSIS ===
  Total tokens: 40
  Token types found:
    - ASSIGN: 1 times
    - COMMA: 1 times
    - DOT: 1 times
    - GT: 1 times
    - HASH: 1 times
    - ID: 9 times
    - INCLUDE: 1 times
    - INCREMENT: 1 times
    - INT: 2 times
    - LBRACE: 2 times
    - LPAREN: 3 times
    - LT: 2 times
    - NUMBER: 3 times
    - RBRACE: 2 times
    - RETURN: 1 times
    - RPAREN: 3 times
    - SEMICOLON: 4 times
    - STRING_LITERAL: 1 times
    - WHILE: 1 times

=== PHASE 2: SYNTACTIC ANALYSIS ===

✗ ERROR: Syntax error at 'while' (line 6)
```

Robustness Tests: Fundamental Cases

We designed an exhaustive test bench to validate each component of the interpreter, prioritizing the essential pillars of execution.



TEST CASE 1: Flow Control (While Loop)

Input: Counter incrementing from 0 to 4

```
while(i < 5) { printf(i); i++; }
```

Result: Verified, 5 sequential outputs (0,1,2,3,4)

Validation: Correct condition logic and fluid execution.

TASTE CASE 2: Recursion (Factorial)

Input: int fact(int n) with n=5

```
if (n <= 1) {return 1;}
else {return n * fact(n - 1);}
```

Result: Verified, returns 120

Validation: Correct Stack Frame isolation and accurate calculation.

These cases validate the correct evaluation of conditions and efficient call stack management.

Robustness Tests: Advanced Cases

We continue with tests that validate data structure and memory management, demonstrating the interpreter's ability to handle real-world complexity.

TEST CASE 3: Arrays & Bounds Checking

Input: Accessing index 10 of an array of size 5.

```
int arr[5]; arr[10] = 99;
```

Result: Error captured - "Index out of range". Validation: Correct detection of memory violation.

TEST CASE 4: String Mutation

Input: Modifying a char[] string index by index.

```
str[0] = 'H'; str[1] = 'i';
```

Result: String successfully reconstructed. Validation: Accurate simulation of mutability in C.

These cases demonstrate the interpreter's safety in handling complex data structures and its faithfulness to C behavior.

Robustness Tests: Advanced Logic

We validate the interpreter's ability to handle logical complexity and control flow in advanced scenarios.

TEST CASE 5: Complex Nesting (Break)

Input: break statement inside an if nested within a while loop.

```
while(i < 10) { if(i == 5) break; i++; }
```

Result: Loop terminated correctly.

Validation: Correct management of control flow signals.

These cases demonstrate the interpreter's fidelity in managing identifier resolution and execution flow in complex structures.

Conclusions: Bridging the Theory-Practice

We have transformed a 'black box' into a transparent, educational, and functional tool.

INTEGRATION

- Fluid 4-phase pipeline
- Lexer + Parser + Semantic Analysis + Execution
- Seamless component interaction

SECURITY

- Robust semantic validation
- Array bounds checking
- Scope isolation

POWER

- Full recursion support
- Complex data structures
- Advanced algorithms

This project demystifies compilers, transforming theoretical abstraction into a tangible engineering tool.