Universidad Nacional Autónoma de México

Computer Engineering

Compilers

INTERPRETER

Students:
320206102
316255819
423031180
320117174
320340312

Group:
05

Semester:
2026 - I

Mexico City, December 2025

Index

1 Introduction

1.1 Problem Statement

This project marks the final phase of our compiler construction process, addressing the critical transition from merely recognizing a program's structure to actually executing it. Following the creation of the lexer and parser, we established a system capable of identifying valid tokens and ensuring that they adhered to the language's grammatical rules. However, despite these elements, the program remained static.

The central issue addressed here is that gap: the need for a component that can take the Abstract Syntax Tree (AST) generated by the parser and transform it into meaningful actions. Unlike a compiler, which translates the source code into machine language for later execution, an interpreter must read, analyze, and execute instructions immediately. This presents a complex challenge requiring a runtime engine capable of simulating a virtual machine to manage memory, control flow, and function calls in real-time.

1.2 Motivation

The motivation behind creating this interpreter goes beyond just completing the compilation pipeline; it is about breathing real functionality into the language we have designed. Building this system provides deep insight into how actual interpreters operate behind the scenes from managing variable environments and applying operator precedence to handling runtime errors.

Furthermore, implementing an interpreter is one of the most comprehensive challenges in compiler design. It requires mastering the complex interaction between lexical analysis, syntactic verification, and semantic validation. A critical component of this motivation is the integration of a Semantic Analyzer before execution. This ensures that the program is not only grammatically correct but logically sound, preventing errors such as type mismatches or the use of undeclared variables. Ultimately, this stage evolves the project into a complete system that can read, validate, and execute programs from the beginning to the end.

1.3 Objectives

The primary objective of this stage is to define and implement the semantics of a subset of the C programming language using Python and the PLY (Python Lex-Yacc) library. The purpose of the system is to process the AST, accurately evaluate expressions, and store values within a functional internal environment.

The specific objectives are as follows:

- **Execute the AST:** Process the tree produced by the parser to apply the semantic rules of the language, simulating the behavior of a virtual machine.
- **Evaluate Expressions:** Handle arithmetic expressions accurately while respecting operator precedence and associativity.
- **Implement Control Flow:** Support complex programming constructs, including `if-else` conditionals, `while` and `for` loops, and `switch-case` structures.
- **Manage Scope and Functions:** Support variable assignments, storage, and user-defined functions with local scope management.
- **Semantic Analysis:** Identify and report semantic errors, such as undefined variables, invalid operations, or type mismatches, providing clear feedback to the user.
- **Develop a GUI:** Design an intuitive Graphical User Interface (GUI) using Tkinter that includes a code editor, real-time tokenization visualization, symbol table inspection, and execution output.

2 Theoretical Framework

The construction of an interpreter relies on several foundational principles from the theory of programming languages, operational semantics, and the design of runtime environments. Interpreters differ from compilers in that they execute programs directly rather than producing a target representation; however, they share essential analytical mechanisms such as lexical analysis, syntactic parsing, semantic validation, and structured state management. Modern literature on interpreter implementation describes this process as a sequence of conceptual phases that gradually transform raw source code into meaningful computational behavior [1], [2].

The first stage of interpretation is **lexical analysis**, where the source program is transformed into a sequence of tokens. This process is grounded in regular languages and finite automata, which enable the systematic recognition of identifiers, literals, operators, delimiters, and reserved words. Lexical analysis abstracts away character-level details such as whitespace and comments, producing a structured token stream that simplifies syntactic reasoning. Scott emphasizes that this transformation is essential for reducing ambiguity and enabling deterministic parsing [3].

Once a token stream is available, the interpreter determines the program's **syntactic structure** using a grammar based on Context-Free Grammars (CFGs). Parsing provides a formal means of verifying that the program conforms to the language specification. Bottom-up parsing methods such as LALR(1), commonly used in interpreter design, apply grammar productions in reverse to build the syntactic structure from the input tokens. This process constructs an **Abstract Syntax Tree (AST)**, a hierarchical representation that captures the essential semantic content of the program while omitting superficial syntactic markers. The AST is a core theoretical concept in interpreter architecture because it decouples the execution model from the concrete syntax. As Nystrom notes, ASTs provide a clean, uniform structure through which all semantic and runtime behaviors can be expressed [1].

With the AST constructed, the interpreter proceeds to **static semantic analysis**, where it enforces context-sensitive rules that cannot be handled by the grammar. These rules include identifier declaration and use, type compatibility in expressions, validation of array sizes and indices, function signature conformance, and restrictions on redefinitions or scope violations. Symbol tables, which map identifiers to their associated attributes, provide the essential infrastructure for this analysis.

This phase relies heavily on a hierarchical **Symbol Table strategy** to enforce scope resolution rules. The interpreter distinguishes between global visibility and local scope (e.g., variables declared inside functions or blocks). According to the rules of block-structured languages like C, a variable declared in a local scope shadows a variable of the same name in the outer scope. The implementation validates these constraints by maintaining a stack of active scopes; when an identifier is referenced, the analyzer searches from the innermost scope outward. This ensures that the program adheres to the lexical scoping discipline (static scoping) defined by the C language standard, preventing illegal access to variables outside of their defined lifetime [4]. Krishnamurthi emphasizes that static semantics bridge the gap between syntax and execution, ensuring that programs respect logical and contextual constraints before they are evaluated [2].

Execution is governed by **operational semantics**, which define how each language construct transforms the program state. To navigate the Abstract Syntax Tree, the interpreter employs a structural design pattern akin to the **Visitor Pattern**. Rather than embedding the execution logic directly within the AST nodes, the interpreter utilizes a centralized dispatch mechanism. This mechanism recursively traverses the tree, identifying the type of each node, such as `IfNode`, `WhileNode`, or `BinaryOpNode`, and routing it to the appropriate evaluation logic. This approach allows for the modular extension of language features without modifying the underlying syntax tree definition. Friedman and Wand describe this evaluation model as a direct interpretation of the language's semantic equations, where each AST node corresponds to a reduction rule or a state transition [5].

In the presence of functions, interpretation requires the creation of new local environments for each invocation. A critical aspect of supporting user-defined functions is the management of **Activation Records (or Stack Frames)**. When a function is called, the interpreter must suspend the current execution context and create a new environment for the callee. Theoretically, this is achieved by pushing a new frame onto the control stack, which contains the function's local variables, parameters, and the return address. In the implemented interpreter, this concept is reified by explicitly saving the current symbol table state before a call and restoring it upon return. This mechanism ensures that local scope is preserved and that recursive function calls maintain distinct variable instances across different invocation levels. Aho et al. define this management of storage allocation as fundamental to the runtime support of any block-structured language [4].

Control-flow constructs, including conditional statements, loops, and multi-branch selection mechanisms, are defined by their operational impact on the program state. The interpreter must manage execution flags such as loop continuation, early termination, and the propagation of return signals through nested contexts. Scott notes that control flow requires interpreters to maintain precise execution state and to apply semantic rules consistently in order to guarantee predictable behavior [3]. In addition to user-defined constructs, interpreters commonly include a set of built-in functions whose semantics are defined as part of the language runtime. Parr observes that built-in functions constitute a small but essential standard library that must integrate cleanly with the interpreter's internal environment model [6].

3 Development

The interpreter is designed as a multi-phase system composed of lexical analysis, syntactic parsing, abstract syntax construction, semantic validation, environment management, and execution. Each subsystem is implemented through specialized classes and functions that interact to transform raw source code into executable behavior.

The lexical analysis phase is defined using token rules and regular-expression specifications that identify identifiers, numeric literals, strings, operators, and reserved keywords. These specifications are expressed through token functions such as `t_ID`, `t_NUMBER`, and `t_STRING`, as well as symbolic definitions for punctuation and operators. The lexer relies on the lexical facilities provided by the PLY library, whose internal mechanisms tokenize the input stream and attach metadata such as line and column information. Lexical errors are reported using the `t_error` function, which constructs structured diagnostic messages.

Syntactic parsing is constructed using the `yacc` module of the PLY library. The parser is defined through a set of grammar functions whose names begin with the prefix `p_` including `p_program`, `p_statement`, `p_expression`, `p_function_definition`, and many others. Each function corresponds to a context-free grammar production and is annotated with a rule signature in its docstring. The PLY Yacc engine processes these signatures to automatically generate an LALR(1) parser. Through the parsing table generated internally by Yacc, the system performs shift-reduce operations and builds the syntactic structure of the program. The parsing behavior is further refined using the precedence declaration, which establishes operator associativity and hierarchical relationships among arithmetic and logical operators. The start symbol is defined implicitly through the `p_program` production, which constructs the top-level `ProgramNode` of the abstract syntax tree.

The Abstract Syntax Tree is built through a hierarchy of node classes that represent the structure of the program. These include classes such as `ProgramNode`, `DeclarationNode`, `AssignmentNode`, `BinaryOpNode`, `UnaryOpNode`, `IfNode`, `WhileNode`, `SwitchNode`, `CaseNode`, `FunctionNode`, and `FunctionCallNode`. During parsing, each grammar function uses these classes to instantiate nodes that capture the semantics of their respective constructs. The AST forms a tree structure where each node references its children, creating a representation that reflects the hierarchical nature of the source program.

Semantic analysis is delegated to the `SemanticAnalyzer` class, whose `analyze` method traverses the AST and enforces contextual rules. During this traversal, the semantic analyzer uses the `symbol_table` dictionary to manage identifier declarations, type information, and scoping. It ensures that variables are declared before use, verifies type compatibility in expressions, validates array bounds, and confirms that function definitions and calls match in arity and parameter types. Function metadata is stored using a function table accessed through the internal dictionary. The semantic analyzer generates structured diagnostic messages via internal lists when inconsistencies or violations are detected.

The execution engine is implemented in the `CInterpreter` class, which uses a set of evaluation methods to traverse and execute the AST. Its `execute` method initializes the global environment and delegates execution to the recursive method `_execute_node`. For each kind of AST node, specialized logic within this method computes values, updates variable bindings, executes control flow, and manages function invocation. Environments are managed through Python dictionaries, which provide operations to maintain lexical scoping. Function calls create a new environment linked to the caller's environment, evaluate arguments, bind them to parameters, and execute the function body. The return mechanism is implemented internally using structured signals that propagate values back through nested evaluation contexts.

Control-flow constructs such as conditionals, loops, and switch-case statements are executed by manipulating interpreter state. The interpreter uses flags such as `break_flag` and `continue_flag` to implement loop interruption, and it evaluates case blocks sequentially in the logic for `SwitchNode`. These mechanisms ensure that the operational semantics of the language are faithfully preserved.

The interpreter includes a suite of built-in functions stored in the `functions` dictionary. These include operations such as `printf`, `scanf`, `print`, `strlen`, `strcmp`, and `strcat`. Built-in functions are registered during interpreter initialization and provide essential functionality for input/output and string manipulation.

Error handling is integrated across all phases of interpretation. During lexical, syntactic, semantic, and runtime phases, errors are generated and appended to diagnostic lists associated with the lexer, parser, semantic analyzer, or interpreter. The system accumulates these diagnostics and presents them to the user if execution cannot continue. The use of PLY's internal error recovery mechanisms, combined with custom error functions such as `p_error`, ensures that syntactic anomalies are captured with precise location information.

The full interpretation sequence is managed by the `interpret` method inside the `CInterpreterEngine` class, which coordinates all phases. It constructs the lexer with PLY's lex module, builds the LALR(1) parser with Yacc's yacc builder, generates the AST, performs semantic validation using `SemanticAnalyzer.analyze`, and finally executes the program through `CInterpreter.execute`. If diagnostics are produced at any stage, the system halts execution and returns the accumulated error information. When no errors are detected, the interpreter executes the program and produces the corresponding output.

Finally, the system integrates a graphical user interface (GUI) developed using the Python `tkinter` library to facilitate user interaction. The `CInterpreterGUI` class manages the application lifecycle, employing widgets such as `ScrolledText` for the source code editor and output logs, and `ttk.Treeview` to visualize the generated tokens. This interface acts as a wrapper for the `CInterpreterEngine`, capturing user input, triggering the analysis pipeline, and displaying real-time feedback.

3.1 Testing

To validate the robustness of the interpreter, a series of tests were designed to target specific features of the language. These tests focused on control flow, scope isolation, and error handling.

Case 1: Control Flow and Arithmetic

Input: A program containing a `while` loop that iterates from 0 to 4. Inside the loop, a counter variable is incremented, and its value is printed at each step using `printf`.
Output: The system correctly interpreted the condition, entered the loop, and produced five sequential lines of output ("Counter: 0" through "Counter: 4"). The loop terminated exactly when the condition became false, verifying the correctness of the evaluation logic.

Case 2: Function Calls and Parameter Passing

Input: A program defining a function `sum(int a, int b)` that returns the addition of two integers. The `main` function declares two variables, `x` and `y`, assigns them values, and passes them to `sum`.

Output: The interpreter successfully switched contexts. It calculated the sum in the local scope of the function and returned the result to the `main` scope. The final output displayed the correct sum, confirming that parameter passing and return statements function correctly without corrupting the variables in the main program.

Case 3: Recursion and Global Scope

Input: A recursive function designed to calculate the factorial of a number. The input included a base case to stop recursion and a recursive step calling the function with `n-1`.
Output: For an input of 5, the interpreter correctly navigated the recursive stack depth and returned 120. This test confirmed that the mechanism for saving and restoring execution contexts (stack frames) works as intended, preventing variable collisions between different recursive calls.

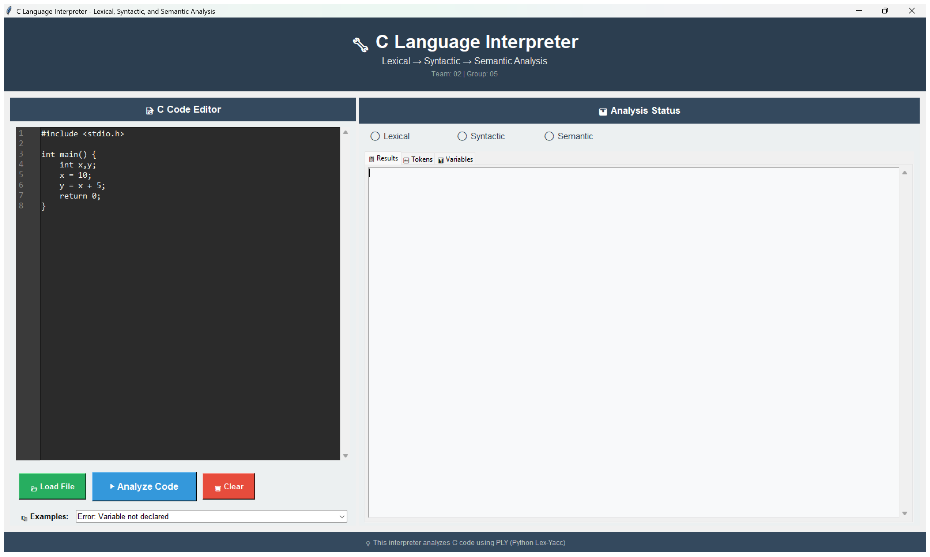
Case 4: Array Manipulation

Input: Code declaring an integer array of size 5 with specific initial values (e.g., `{10, 20, 30...}`). A loop was used to iterate through the array, modifying specific indices and printing the values.
Output: The output showed the initial values correctly retrieved and the modified values updated in memory. Additionally, a deliberate attempt to access an out-of-bounds index (e.g., index 10) resulted in a controlled runtime error message rather than a system crash, validating the bounds-checking logic.

Case 5: Semantic Error Detection

Input: A snippet of code where a variable is used in an arithmetic operation without being declared, and another snippet where a string is assigned to an integer variable.
Output: The execution did not start. Instead, the Semantic Analyzer intercepted the request and displayed specific error messages in the GUI logs, identifying the exact line numbers of the undeclared variable and the type mismatch. This confirmed that the static analysis phase successfully acts as a safety filter before runtime.

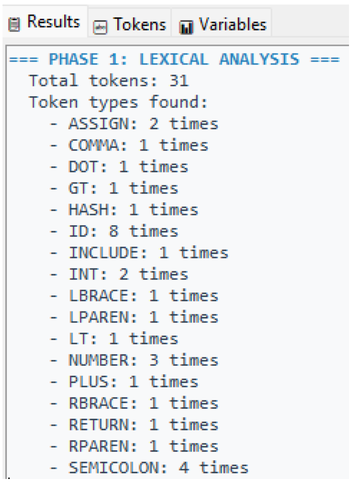
4 Results


C Language Interpreter - Lexical, Syntactic, and Semantic Analysis
C Language Interpreter
Lexical → Syntactic → Semantic Analysis
Team: 02 | Group: 05
C Code Editor
Analysis Status
Lexical
Syntactic
Semantic
Results
Tokens
Variables
Load File
Analyze Code
Clear
Examples:
Error: Variable not declared
This interpreter analyzes C code using PLY (Python Lex-Yacc)

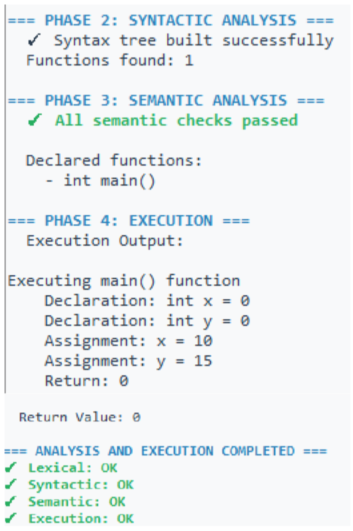
```c
#include <stdio.h>

int main() {
    int x,y;
    x = 10;
    y = x + 5;
    return 0;
}
```

Figure 1: Program Start

The program starts with an interface divided into two main panels: The first one is called C Code Editor, and it has operational buttons, starting with one that allows you to Load File saved on the computer; the second one is for Analyze Code to run the interpreter, and the last one is for Clear to clean the entry, while below these buttons there is a bar where you can select a precharged example, accompanied by a second panel called Analysis Status, where you can see the Results, the table of Tokens, and the Variables.



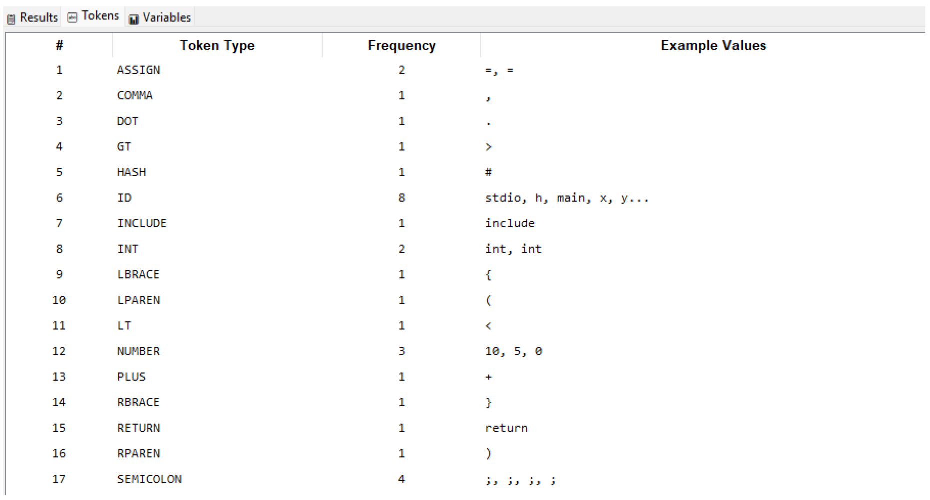
```
=== PHASE 1: LEXICAL ANALYSIS ===
  Total tokens: 31
  Token types found:
    - ASSIGN: 2 times
    - COMMA: 1 times
    - DOT: 1 times
    - GT: 1 times
    - HASH: 1 times
    - ID: 8 times
    - INCLUDE: 1 times
    - INT: 2 times
    - LBRACE: 1 times
    - LPAREN: 1 times
    - LT: 1 times
    - NUMBER: 3 times
    - PLUS: 1 times
    - RBRACE: 1 times
    - RETURN: 1 times
    - RPAREN: 1 times
    - SEMICOLON: 4 times
```



```
=== 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 x = 0
    Declaration: int y = 0
    Assignment: x = 10
    Assignment: y = 15
    Return: 0

 Return Value: 0

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

Figure 2: Results section

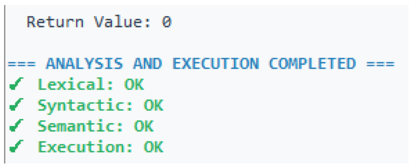
Once you insert a code into the C Code Editor and click the Analyze Code button, you will see in the Results panel a detailed breakdown starting with PHASE 1: LEXICAL ANALYSIS, which lists the total tokens and types found, followed by PHASE 2: SYNTACTIC ANALYSIS confirming the successful construction of the syntax tree, continuing with PHASE 3: SEMANTIC ANALYSIS to verify that all checks passed, and concluding with PHASE 4: EXECUTION displaying the values and operations performed, ending with a summary confirming that the ANALYSIS AND EXECUTION COMPLETED successfully with green indicators for each stage.



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

Figure 3: Tokens section

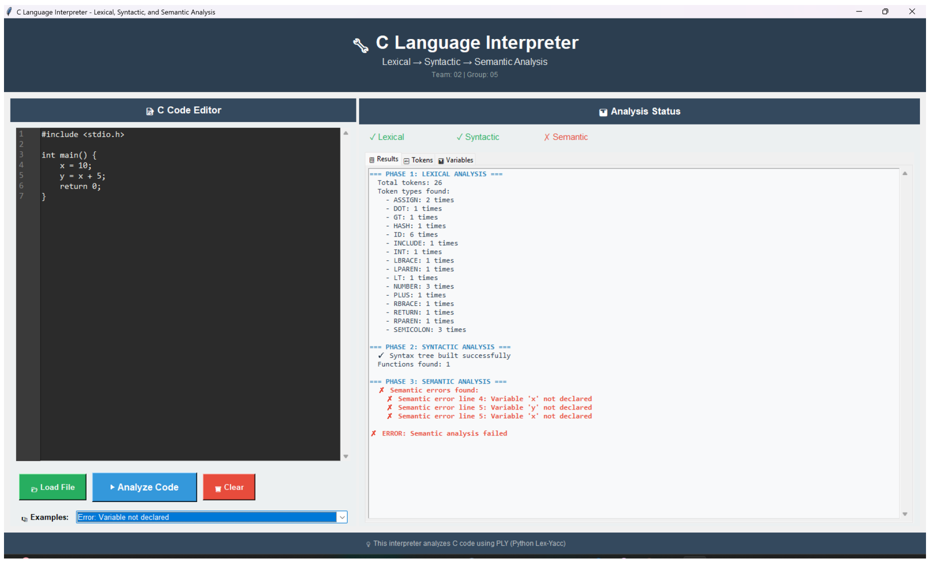
By navigating to the Tokens tab within the Analysis Status panel, you can view a comprehensive table that categorizes the lexical analysis, displaying columns for Token Type, Frequency, and Example Values for every element detected, such as ID, NUMBER, and ASSIGN.



```
 Return Value: 0

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

Figure 4: Variables section

Then, the Variables tab within the Analysis Status panel, you can inspect the final state of the execution, showing a list of Variables in memory that details the identifier, data type, and assigned value for each



C Language Interpreter - Lexical, Syntactic, and Semantic Analysis
C Language Interpreter
Lexical → Syntactic → Semantic Analysis
Team: 02 | Group: 05
C Code Editor
Analysis Status
✓ Lexical
✓ Syntactic
✗ Semantic
Results
Tokens
Variables
Load File
Analyze Code
Clear
Examples:
Error: Variable not declared
This interpreter analyzes C code using PLY (Python Lex-Yacc)

```
1  #include <stdio.h>
2
3  int main() {
4      x = 10;
5      y = x + 5;
6      return 0;
7  }
```

```
=== PHASE 1: LEXICAL ANALYSIS ===
  Total tokens: 26
  Token types found:
    - ASSIGN: 2 times
    - DOT: 1 times
    - GT: 1 times
    - HASH: 1 times
    - ID: 6 times
    - INCLUDE: 1 times
    - INT: 1 times
    - LBRACE: 1 times
    - LPAREN: 1 times
    - LT: 1 times
    - NUMBER: 3 times
    - PLUS: 1 times
    - RBRACE: 1 times
    - RETURN: 1 times
    - RPAREN: 1 times
    - SEMICOLON: 3 times

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

=== PHASE 3: SEMANTIC ANALYSIS ===
   ✗ Semantic errors found:
     ✗ Semantic error line 4: Variable 'x' not declared
     ✗ Semantic error line 5: Variable 'y' not declared
     ✗ Semantic error line 5: Variable 'x' not declared

✗ ERROR: Semantic analysis failed
```

Figure 5: Error Case

Finally, when the code contains logical faults, the Results window highlights the specific Types of error encountered; in this instance, it displays Semantic errors marked with red crosses, detailing that the variables 'x' and 'y' were not declared on specific lines, causing the Semantic analysis failed message to appear despite successful lexical and syntactic passes.

5 Conclusions

The implementation of the C Language Interpreter demonstrated the effective application of compiler theory principles, specifically the transition from static analysis to dynamic execution. The system successfully leveraged the PLY library to construct an LALR(1) parser, validating the theoretical framework of Context-Free Grammars (CFGs) to transform raw source code into a structured Abstract Syntax Tree (AST). This process ensured that the syntactic structure of the program was accurately recognized, serving as the foundational layer for subsequent semantic interpretation.

By distinguishing between static semantic analysis and operational semantics, the system guaranteed that only logically sound programs proceeded to execution. The integration of a hierarchical symbol table enabled the strict enforcement of scoping rules and type safety, effectively preventing runtime errors related to undeclared variables or type mismatches. This verified the theoretical concept that static semantics must bridge the gap between syntax and execution, acting as a safety filter before the evaluation of the AST.

In relation to the defined objectives, the interpreter satisfied the requirements for handling complex control flow, arithmetic operations, and modular programming. The execution engine, built upon a recursive traversal mechanism (Visitor Pattern), successfully simulated a virtual machine environment capable of managing memory states, stack frames for function calls, and recursion. The successful processing of test cases involving while loops, array manipulation, and recursive factorials confirmed that the environment management logic correctly maintained variable lifecycles across different scopes.

Overall, the system successfully translated the theoretical phases of interpretation (lexical, syntactic, semantic, and execution) into a functional prototype. The inclusion of the Graphical User Interface (GUI) provided necessary visualization of these phases, confirming the correlation between the internal token stream and the final output. The project results illustrate how formal language specifications can be synthesized into a runtime engine that faithfully reproduces the operational behavior of the C programming language subset.

6 References

[1] R. Nystrom, *Crafting Interpreters*, 1st ed. London: Genever Benning, 2021.

[2] S. Krishnamurthi, *Programming Languages: Application and Interpretation.* Providence, RI: Brown University, 2007. [Online]. Available: https://cs.brown.edu/~sk/Publications/Books/ProgLangs/.

[3] M. L. Scott, *Programming Language Pragmatics*, 3rd ed. Burlington, MA: Morgan Kaufmann, 2009.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2006.

[5] D. P. Friedman and M. Wand, *Essentials of Programming Languages*, 3rd ed. Cambridge, MA: MIT Press, 2008.

[6] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Raleigh, NC: Pragmatic Bookshelf, 2010.