Phase 3 : Semantic Analyzer

CISC 458

Shams Sajid - 20217791

Harshil Patel - 20250563

Cj Gonsalves - 20230353

Devin Chawla - 20223139

# Overview

The Semantic Analysis Phase builds upon the previous lexical and syntactic analyses to ensure that a program is not only well-formed in terms of grammar but also semantically correct. This project focuses on verifying that programs adhere to language rules such as proper variable declaration, scope management, and type consistency. By extending the capabilities of the compiler, the semantic analyzer contributes significantly to catching logical and contextual errors that are not detectable during parsing.

The primary goals of this phase are to:

* **Ensure Semantic Correctness:**  
  Validate that all variables and functions are declared before use, check that no redeclarations occur within the same scope, and verify that expressions are type-compatible.
* **Manage Scopes and Symbol Tables:**  
  Construct and maintain a symbol table that tracks variables, their types, scope levels, and initialization status.
* **Provide Meaningful Error Reporting:**  
  Detect semantic errors such as undeclared or uninitialized variables, type mismatches, and improper control flow usage, and output informative error messages including line numbers.
* **Enhance the Compiler Front-End:**  
  Integrate semantic analysis with existing lexical and syntactic analysis to create a more robust compiler that prevents a wide range of potential runtime issues.

# Design Decision and Implementation

### Symbol Table Implementation

The symbol table is a fundamental data structure implemented to track identifiers (variables and functions), their types, scope levels, declaration locations, and initialization statuses throughout semantic analysis. It is implemented as a linked list for flexibility and simplicity. Each symbol table entry contains the following information:

* **Identifier Name**: A string that uniquely identifies the variable or function.
* **Type**: Represents the data type associated with the identifier (e.g., integer).
* **Scope Level**: An integer value that indicates the current nesting depth of the scope, incremented upon entering a new scope and decremented when exiting.
* **Line Declared**: The exact line number in the source code where the identifier was first declared, useful for accurate error reporting.
* **Initialization Status**: A flag that tracks whether a variable has been initialized (assigned a value).

The supported operations of the symbol table include:

* **Initialization**: Creating an empty symbol table with no symbols and the scope level set to zero.
* **Insertion**: Adding a newly declared identifier into the current scope level, while ensuring no redeclarations occur.
* **Lookup**: Searching the symbol table for identifiers across all available scopes or restricted to the current scope, crucial for both ensuring variable declaration and handling shadowing scenarios.
* **Scope Management**: Operations to enter a new scope (such as when entering a block statement) and exit an existing scope, including automatic removal of identifiers declared in scopes that are no longer accessible.
* **Memory Cleanup**: Deallocating all dynamically allocated symbols and the symbol table structure itself upon completion of semantic analysis, ensuring no memory leaks remain.

### Semantic Checking Rules

The semantic analyzer enforces several crucial rules to verify that the program not only adheres to the grammar rules but is also logically and semantically meaningful. These rules include:

* **Variable Declaration**: A variable must be declared explicitly before being used. The analyzer checks that each identifier appearing in expressions or assignments exists in the symbol table, reporting undeclared variable errors if needed.
* **Redeclaration Check**: Each scope cannot have multiple declarations of the same identifier. If a redeclaration within the same scope occurs, the analyzer issues a redeclaration error.
* **Variable Initialization**: Variables must be initialized before use. The analyzer tracks initialization status and issues a warning or error if it detects the usage of an uninitialized variable.
* **Type Checking**: Expressions involving operators or assignments are checked to ensure operand compatibility. Although the current implementation primarily deals with integers, the architecture supports the future inclusion of additional types and more advanced type checking rules.
* **Scope Management and Variable Accessibility**: The semantic analyzer enforces strict scoping rules, making variables declared within inner scopes inaccessible from outer or parallel scopes. This ensures clarity and prevents unintended variable conflicts or shadowing issues.
* **Control Flow Statements**: Conditions within control flow structures (if, while, repeat-until) are evaluated to verify their correctness and to ensure variables used within these conditions are declared and initialized appropriately.

The implementation of the symbol table and semantic checking rules in our code is carried out through structured procedures integrated within the semantic analyzer, working closely with the abstract syntax tree (AST) generated by the parser.

Our symbol table implementation is realized using a dynamic linked-list data structure. Each entry in this list represents a symbol holding critical information such as its identifier name, data type, scope level, line of declaration, and initialization status. When the semantic analyzer encounters a new variable declaration during the AST traversal, the symbol is inserted into this linked list, provided it doesn't already exist in the current scope, thereby preventing redeclarations. For instance, if the variable x is declared within a certain block of code, an entry for x is created with the current scope level. As the analyzer navigates through nested scopes (such as within if- or while-blocks), the scope level increments, and new symbols are marked accordingly, ensuring precise scope management. When a scope is exited, the analyzer systematically removes symbols associated with that scope, maintaining a clean and accurate symbol table state at all times.

Semantic checking rules are systematically enforced by traversing the AST nodes recursively. During this traversal, the analyzer checks multiple conditions to ensure semantic correctness. For example, when encountering an assignment operation like x = 5, the analyzer performs a lookup in the symbol table to verify if the identifier x has been previously declared. If x is undeclared, the analyzer generates a semantic error indicating the undeclared variable's line number and name. Additionally, if a declared variable is being used before assignment without prior initialization of x, the analyzer flags this usage, highlighting that the variable might be used in an uninitialized state.

Control flow structures, such as if-statements and loops, undergo similar scrutiny. The analyzer checks the semantic validity of conditions, ensuring all identifiers involved are declared and initialized. Nested scopes within these constructs are handled meticulously; upon entering a new block, the analyzer increments the scope level and adds any new declarations to the symbol table. When exiting the block, these symbols are cleaned up appropriately.

To summarize, our semantic analyzer's code achieves symbol table implementation and semantic checking through a robust linked-list structure coupled with detailed AST traversal algorithms. This ensures strict adherence to language semantics regarding variable declarations, scope rules, initialization requirements, and the integrity of control structures.

### Error Handling Approach

Error handling in the semantic analyzer is designed for clarity, precision, and immediate feedback. Semantic errors detected during AST traversal trigger immediate error reporting through a dedicated semantic error function. The analyzer outputs descriptive error messages specifying:

* **Error Type**: Clearly indicating the kind of semantic issue, such as undeclared variables, redeclarations, uninitialized usage, or invalid operations.
* **Affected Identifier**: Naming the specific identifier involved in the semantic issue to assist in quick identification and resolution.
* **Source Line Number**: Providing the line number where the error occurs, enabling easy location of the problematic code within the source file.

The semantic analyzer handles errors by employing a structured and informative approach that provides clear and precise feedback during compilation. Specifically, when the analyzer detects a semantic violation, such as the usage of an undeclared variable, redeclaration of a variable within the same scope, or the use of an uninitialized variable, it immediately reports these errors with descriptive messages. These error messages include details like the specific nature of the error, the identifier involved, and the exact line number where the issue was encountered. For example, if an identifier is used without prior declaration, the analyzer explicitly states the undeclared identifier's name and its line of usage, significantly aiding the debugging process. Similarly, redeclaration errors clearly indicate which variable is involved, facilitating quick identification and correction of the issue.

### Extensions and Enhancements

Several enhancements were implemented beyond the basic requirements, significantly improving the semantic analyzer’s capabilities and preparing it for potential future extensions:

* **Enhanced AST Structure for Conditional Statements**: The Abstract Syntax Tree was extended to include optional else branches within if statements, thereby accurately modeling conditional logic and allowing future enhancements such as else if chains or nested conditionals.
* **Function Call Validation**: Demonstrated through the factorial function example, the semantic analyzer includes the capability to handle function calls semantically, verifying correct argument usage. This serves as a robust basis for adding more advanced function call handling features.
* **Debugging and Visualization Tools**: The project incorporates helper functions that allow for detailed output of the symbol table and AST structures. The symbol table dump provides a clear view of identifiers’ scopes and initialization states, while AST printing offers insights into the compiler's internal representation of the analyzed program.
* **Modular and Extensible Architecture**: The semantic analyzer’s modular design facilitates the addition of new semantic checks or language features. Its clean separation of concerns between AST traversal, symbol table management, and error reporting promotes maintainability and scalability for larger and more complex compiler projects.

#### Outputs

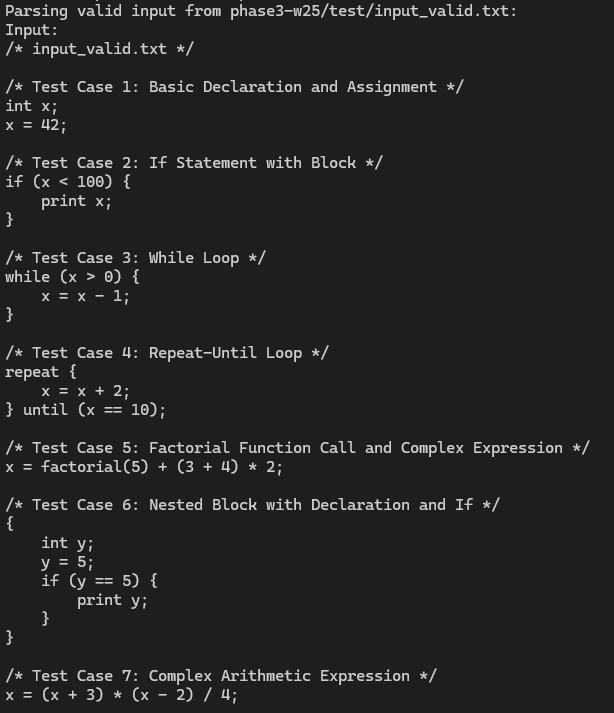


Figure 1: Output after Running the Valid Test File

The output results demonstrate that a test file containing a variety of programming elements, such as variable declarations, function calls, arithmetic expressions, and control flow structures (if statements, while loops, and repeat-until loops), was successfully parsed. This output's code implementation is a C semantic analyzer, which verifies a program's validity by examining variable declarations, assignments, and scope management. Declared variables and their properties, including type, scope level, and initialization state, are tracked by the code's symbol table. Additionally, it has functions to notify semantic issues (using an undeclared variable), lookup variables, and handle scope entry and exit. The analysis process traverses an Abstract Syntax Tree (AST), validating statements and expressions, ensuring that all operations conform to semantic rules. The output confirms that the code correctly processes and validates the test cases without any semantic errors.

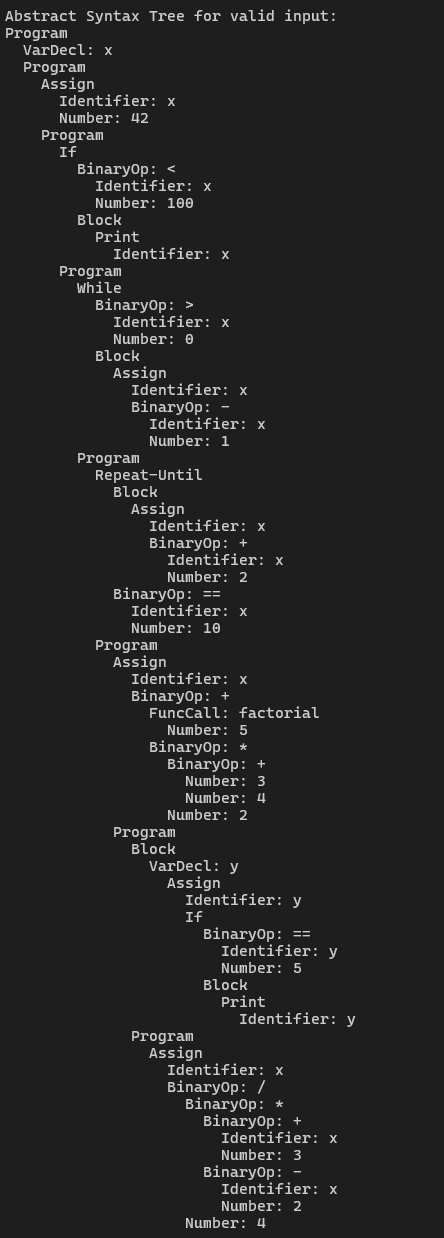


Figure 2: Output Showing the Syntax Tree

The output image shows the hierarchical structure of expressions and statements and reflects the Abstract Syntax Tree (AST) for a valid input program. Variable declarations, assignments, function calls, arithmetic expressions, and control flow structures (if statements, while loops, and repeat-until loops) are all graphically broken down by the AST. An operation, such as a binary operator (+, -, \*, /), a function call (factorial(5)), or a control statement (if, while loops), is represented by each node in the tree.  
  
This structure is processed by the associated C implementation from the given code utilizing recursive analysis of various AST nodes by functions such as check\_statement(), check\_expression(), and check\_block(). The functions add\_symbol() and lookup\_symbol() in the symbol table make sure that variables like x and y are declared before being used. The symbol table tracks variables like x and y, making sure they exist before they’re used. The semantic analysis steps in to check for errors like type mismatches or uninitialized variables. The fact that this AST was generated means the parser and analyzer successfully understood and structured the input program without issues.

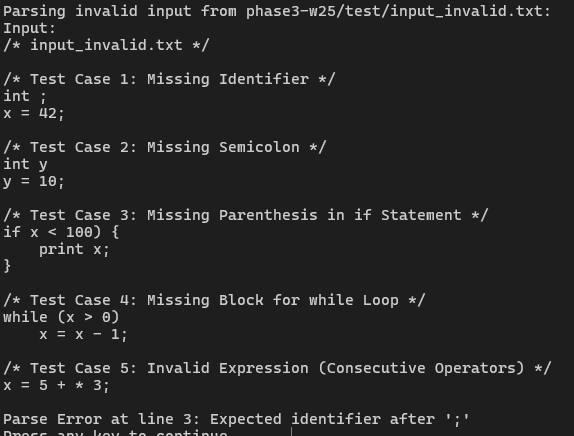


Figure 3: Output from the Invalid Test File

This output displays syntax issues discovered while parsing input\_invalid.txt, highlighting different errors that would cause a program to fail. The first issue is a missing identifier following int;, indicating that a variable name was expected but not provided. The second problem is a missing semicolon following the declaration and assignment of y = 10, which is necessary to correctly terminate statements. The third error is a missing closing parenthesis in an if condition (if x < 100 {), leading to a parsing failure. The fourth issue is a while loop without a suitable block, which means the loop's body is not enclosed by curly braces. The last mistake is an incorrect expression with consecutive operators (x = 5 + \* 3;) that is not syntactically correct. The parser catches the first critical error at line 3, reporting "Expected identifier after ';'", which stops further parsing. This output is crucial for debugging, as it helps identify syntax mistakes early before compilation, making it easier for developers to correct their code efficiently.

# Conclusion

In conclusion, the semantic analyzer implemented for this compiler project effectively ensures semantic correctness and logical coherence of programs by systematically enforcing rules regarding identifier declaration, scope management, and variable initialization. Through its robust symbol table implementation, detailed semantic checking, and informative error-handling strategies, the analyzer provides immediate and precise feedback, aiding developers significantly in debugging and refining their code. Moreover, enhancements such as extended AST structures, preliminary support for function call validation, and comprehensive debugging tools demonstrate thoughtful consideration toward future scalability and maintainability. These deliberate design choices ensure that the semantic analyzer not only meets the project's current requirements but also serves as a stable and flexible foundation for further expansion, making it well-suited to handle the increasing complexity of future compiler developments.