
# Universidad Nacional Autónoma de México

Computer Engineering

Compilers

# SYNTAX & SEMANTIC ANALYZER

Students:
320206102
316255819
423031180
320117174
320340312

Group:
05

Semester:
2026 - I

Mexico City, November 2025

# Index

# 1 Introduction

One of the most critical components in the design of a compiler is the parser, which serves as the bridge between lexical analysis and semantic interpretation. Its main purpose is to verify that the source program follows the grammatical structure of the programming language and to produce a syntactic representation, such as a parse tree or abstract syntax tree (AST). The parser must handle ambiguous grammars, manage precedence and associativity rules, and ensure accurate error detection and recovery mechanisms.

In addition to syntactic analysis, compilers often rely on Syntax-Directed Translation (SDT) to associate semantic actions with grammar productions. SDT enables the compiler to perform semantic analysis and intermediate code generation alongside parsing. Integrating SDT into a parser adds another layer of difficulty, as it requires maintaining correct attribute propagation, managing scope and symbol tables, and ensuring that semantic rules are executed in the proper order.

The central challenge lies in designing and implementing a Parser and SDT system that can accurately analyze the syntax of a source program, detect and report errors, and execute semantic actions essential for subsequent compilation stages. This requires defining an appropriate grammar, selecting an efficient parsing strategy (such as LL, LR, or recursive descent), and integrating SDT rules to maintain correctness and consistency throughout the translation process.

## 1.1 Objectives

1. Design and implement an LL(1) top-down parser capable of analyzing the syntactic structure of source programs based on a defined grammar.
2. Develop a context-free grammar that supports the declaration of variables with different data types, including `int`, `float`, `double`, `char`, `long`, and their constant variants (e.g., `const int`, `const double`).
3. Integrate a Syntax-Directed Translation (SDT) mechanism that performs semantic actions during parsing to evaluate mathematical expressions involving addition, subtraction, multiplication, and division.
4. Provide user feedback to display whether the parsing and semantic analysis processes were successfully completed or encountered errors.
5. Generate and visualize the SDT or parse tree to represent the syntactic and semantic structure of the analyzed source code.
6. Enable file loading and processing functionality to allow users to input source programs directly from external files for analysis.
7. Maintain a symbol table to store identifiers, data types, and values for later stages of semantic analysis.

# 2 Theoretical Framework

The design and construction of compilers are grounded in the theoretical foundations of formal languages and automata theory. Within this framework, the compiler's front end involves a series of processes designed to translate high-level source code into a structured, intermediate representation that a computer can understand. These processes, primarily lexical, syntactic, and semantic analysis, are fundamental in ensuring that a program is not only grammatically correct, but also logically coherent [1], [2].

The process begins with lexical analysis, the first phase of the compiler. The lexical analyzer reads the source code as a stream of characters and groups them into meaningful sequences called lexemes. For each lexeme, it produces a token representing its classified category (e.g., an identifier, a keyword, a constant, or an operator). This resulting stream of tokens becomes the input for the next phase, the syntactic analyzer.

Following semantic analysis, many compilers generate an intermediate representation (IR) of the program. This IR serves as a bridge between the front-end (lexical, syntactic, and semantic phases) and the back-end (code generation and optimization). By operating on the IR, the compiler can perform high-level optimizations, target different machine architectures more easily, allowing the compiler to separate language-specific rules (such as syntax and type checking) from machine-specific details (such as instruction selection and register allocation), which improves modularity, portability, and maintainability [3].

After tokenization, the syntactic analyzer, or parser, interprets this token sequence according to the grammar of the programming language. The syntax of modern programming languages is formally specified using a Context-Free Grammar (CFG). A CFG is a formal specification composed of four key components: a set of terminals (the tokens), a set of non-terminals (syntactic variables), a set of production rules that describe how non-terminals can be expanded, and a designated start symbol [2]. The parser's objective is to verify whether the input stream of tokens can be derived from the start symbol of the grammar. According to Aho et al. [1], this step constructs a parse tree or syntax tree that reveals the hierarchical structure and the precedence of operations within the program, confirming its syntactic correctness.

Parsers can be implemented using two primary methodologies: Top Down or Bottom Up. A top-down parser, as the name implies, attempts to build the parse tree from the root (the start symbol) down to the leaves (the tokens). Conversely, a bottom-up parser builds the tree from the leaves up to the root.

A common and straightforward implementation of top-down parsing is the recursive descent parser. This technique uses a set of mutually recursive procedures, one for each non-terminal in the grammar, to process the input. In the context of compiler implementation, one of the most efficient top-down strategies is the predictive LL(1)

parser. The "LL(1)" designation signifies that it scans the input from left to right, constructs a leftmost derivation, and uses 1 symbol of lookahead to make its decisions. This is a non-backtracking form of recursive descent parsing. Grune et al. [4] emphasize that the predictability of this method allows for efficient and deterministic parsing.

To use this method, the grammar must satisfy the LL (1) conditions. Two common issues must be resolved: left recursion, which can cause infinite loops in a top-down parser, and non-determinism, which arises when two productions share a common prefix, making it difficult for the parser to "predict" the correct rule. These issues are resolved through grammar transformations, namely left recursion elimination and left factoring [2]. The parser's logic is often guided by a parsing table constructed using the FIRST and FOLLOW sets derived from the grammar's non-terminals [1].

While syntactic analysis verifies the structural correctness of a program, semantic analysis ensures that it conveys valid meaning within the rules of the language. This phase is often implemented through Syntax Directed Translation (SDT), a process that augments parsing by attaching semantic actions (snippets of code) to the grammar's production rules. These actions are executed as the parser recognizes syntactic constructs, allowing for type checking, symbol management, and the computation of intermediate results. As Aho et al. [1] explain, SDT combines syntactic structure with semantic rules, forming the bridge between grammatical form and computational meaning.

A fundamental data structure in this process is the symbol table, which records identifiers along with their associated attributes, such as type, scope, and value. The symbol table supports semantic validation by enabling the compiler to check that variables are declared before use, constants are not reassigned, and operations are type consistent. Through these mechanisms, the compiler guarantees contextual integrity and enforces the language's semantic rules [1], [2].

In many compiler implementations, after semantic analysis the compiler also performs early optimization passes. These may include dead code elimination, simplification of control flow graphs, and early detection of redundant computations. Early optimizations improve both execution performance and memory usage of the compiled program and lay the groundwork for more advanced optimizations in later back-end phases [3].

Once the front-end phases (lexical, syntactic, semantic) complete their work and produce a sound intermediate representation along with symbol information and any early optimizations, the compiler enters its back-end stage. The back-end is responsible for transforming the IR into target code, performing machine level optimizations, register allocation, instruction scheduling, and code emission. While the front-end ensures the program is correct and meaningful, the back-end ensures it is efficient and executable on the target platform [4].

In summary, the compiler front-end relies on a sequence of analyses. Syntactic and semantic analysis form the core of this process. Their theoretical principles, which come from formal grammar and semantic theory, guarantee the accurate and consistent interpretation of a program. Syntax Directed Translation serves to unify these phases, translating grammatical structures into semantically valid actions. This enables the systematic transformation of human readable code into an executable or intermediate form.

# 3 Development

## 3.1 Implementation Details

The project was implemented in Python and designed to analyze and visualize the structure of source code using lexical, syntactic, and semantic analysis. It follows a top-down LL(1) parsing approach, meaning the parser reads the code from left to right and decides which grammar rule to apply by looking only one token ahead. This makes the analysis deterministic and efficient, while also making it easier to map how each part of the program fits into the overall structure.

The grammar used defines the basic structure of a simple programming language that includes variable declarations, assignments, and arithmetic expressions. The main non-terminal symbols represent constructs such as Program, Statement, Declaration, Assignment, Expr, Term, and Factor.

- A **Program** consists of one or more **Statements**.
- A **Statement** can be either a declaration or an assignment.
- A **Declaration** typically follows the form of a type, an identifier, an assignment operator, and an expression, ending with a semicolon.
- An **Assignment** reuses an existing identifier and assigns it a new value.
- **Expressions** are built using the recursive rules `Expr → Term Expr'` and `Term → Factor Term'`, where `Expr'` and `Term'` handle recursive cases for operators like `+`, `-`, `*`, and `/`.
- **Factors** can be numeric or character constants, identifiers, or nested expressions enclosed in parentheses.

The lexical analyzer identifies the tokens. These include identifiers, keywords (like `int`, `float`, `char`, or `const`), operators (`+`, `-`, `*`, `/`, `=`), numbers, and special symbols. Each token carries a type and value, allowing the parser to interpret its role in the program.

The syntactic analyzer applies the grammar rules to the list of tokens to ensure that the code follows the correct structure. It also constructs an Abstract Syntax Tree (AST) in real time. Each grammar element, like an expression, statement, or operator, is represented as a node in the tree. These nodes are connected hierarchically, allowing the entire structure of the program to be visualized. Once the parsing process finishes successfully, the tree is passed to Graphviz, which generates a graphical image that shows the relationships between operations, variables, and expressions.

The semantic analyzer is integrated directly into the parsing process through Syntax-Directed Translation (SDT). Each grammar rule has attached actions (embedded in Python code) that perform semantic operations, such as evaluating expressions, updating variable values, or checking for rule violations.

This semantic component ensures that the code is not only syntactically valid but also meaningful. For instance, it prevents using undeclared variables, re-declaring identifiers, dividing by zero, or modifying constants. All declared variables are stored in a symbol table, which maintains information about each identifier:

- **Name** (e.g., `x`)
- **Type** (e.g., `int`, `float`)
- **Value** (assigned value)
- **Line position**
- **is_const flag** (true if declared with `const`)

Whenever a declaration is parsed, an entry is created in the symbol table. If an identifier appears in an expression, its value is retrieved and applied during arithmetic evaluation. If a rule violation is detected, such as referencing a variable that does not exist in the table, a Semantic Error is raised.

All these actions are triggered inside the parser methods like `declaration()`, `assignment()`, and `expr()`. Each step logs semantic operations to the output, so the GUI can display messages such as:

```
SDT: Declared 'x' (int) = 10
SDT: 5 + 3 * 2 = 11
SDT: Division by zero
```

When a semantic error occurs, the program reports: **Parsing Success! | SDT error...** This distinction ensures that syntactically valid code that fails semantic checks is still visualized in the AST, helping identify where the logic breaks.

For example, when the user inputs:

```
int x = 10;
```

the process unfolds as follows:

**1. Lexical Analysis** : The lexer converts the code into tokens:

```
<KEYWORD, 'int'>, <ID, 'x'>, <OP, '='>, <NUM, '10'>, <SPECIAL, ';'>, <EOF, '$'>
```

These tokens are sent to the parser.

**2. Syntactic Analysis** : The parser matches the grammar rule for a declaration:

```
Declaration → type ID = Expr ;
```

It builds the following AST hierarchy:

```
Program
 +-- Statement
     +-- Declaration
         |-- Type: int
         |-- Identifier: x
         |-- Operator: =
         +-- Expr
             +-- Term
                 +-- Factor -> 10
```

This structure is displayed as a text-based tree and also rendered graphically through Graphviz.

**3. Semantic Actions (SDT Execution)** : When the parser recognizes the declaration rule, it executes:

```
self.symbol_table.add('x', 'int', 10, line, is_const=False)
```

A new entry is added to the symbol table:

| Name | Type | Value | Const |
|---|---|---|---|
| x | int | 10 | No |

The following semantic log message appears:

```
SDT: Declared 'x' (int) = 10
```

**4. Final Output** :

- Parsing completes successfully: Parsing Success!
- SDT passes with no semantic violations: SDT Verified!
- The GUI displays the token list, the syntax tree, and the updated symbol table.

## 3.2 Testing

**Valid Programs:** Basic declarations like `int x = 10;` produced correct tokenization, symbol table entries, and generated simple trees showing `Program → Statement → Declaration` structure. Arithmetic expressions `int result = 5 + 3 * 2;` applied proper precedence (result: 11) with the tree clearly showing multiplication nested deeper than addition. Complex expressions like `int value = (10 + 5) * 2 - 8 / 4;` evaluated to 28, with tree visualization making operator precedence and parenthesized subexpressions obvious. Multiple variables, unary operators, character literals, and type inference all worked as expected.

**Syntax Errors:** Missing semicolons (`int x = 10`) and unmatched parentheses (`int x = (10 + 5;`) were caught appropriately. No trees were generated since parsing did not complete. The GUI showed red status indicators and clear error messages describing what was expected versus what was found.

**Semantic Errors:** Undeclared variables (`int x = 10; x = y + 5;`) generated trees (syntax was correct) but caught semantic violations, showing "Parsing Success! | SDT error...". Division by zero and constant modification attempts were similarly handled; trees generated, but semantic checks failed appropriately.

# 4 Results

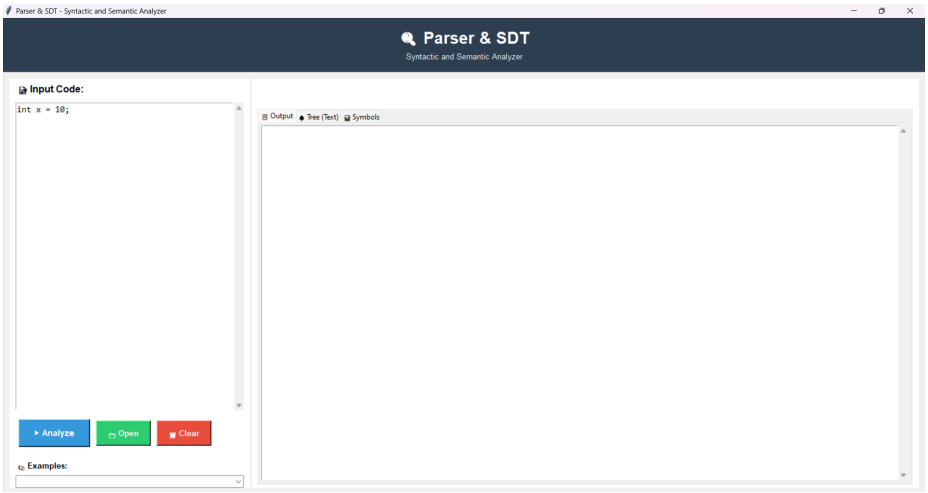


Figure 1: Start interface

The interface of the Parser & SDT (Syntactic and Semantic Analyzer) program is divided into two main sections. On the left, the "Input Code" area allows users to enter or paste source code to be analyzed. Below this area are three control buttons: Analyze, which processes the input and performs the syntactic and semantic analysis; Open, which allows loading a code file from the system; and Clear, which erases both

the input and output results. There is also an Examples dropdown menu that provides predefined code snippets for testing purposes. On the right side, the Output section presents the results of the analysis through three tabs: Output, where messages and analysis results are displayed; Tree (Text), which shows the syntactic tree in textual form; and Symbols, which lists the detected symbols, such as variables and constants, along with their attributes.

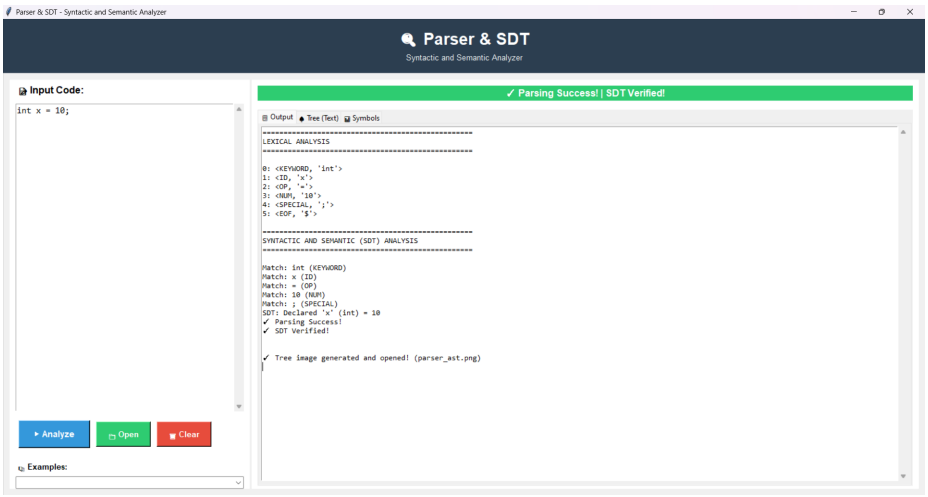


Figure 2: Successfully processing an input

The output panel of the Parser & SDT program displays the successful analysis of the statement `int x = 10;`, confirmed by the "Parsing Success! | SDT Verified!" message. The initial Lexical Analysis phase successfully tokenized the input into its component parts, such as the keyword '`int`', the identifier '`x`', the operator '`=`', the number '`10`', and the punctuation character '`;`'.

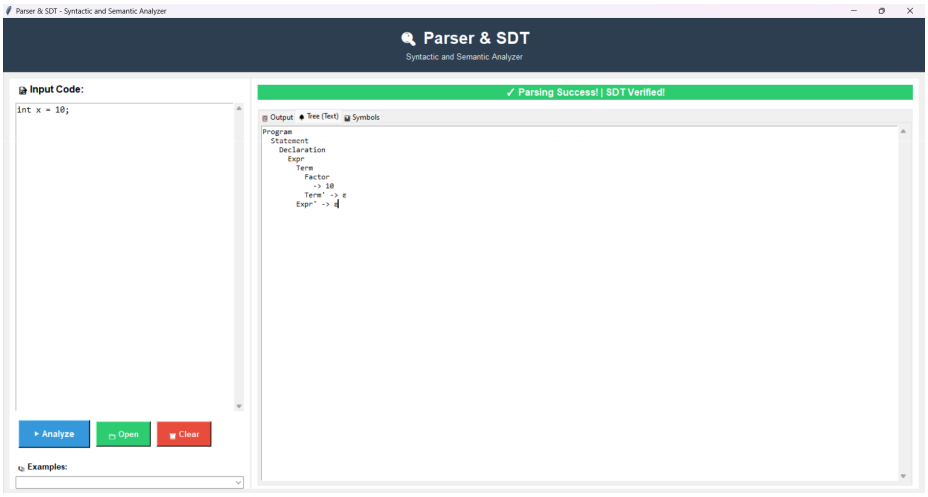


Figure 3: Abstract Syntax Tree

The Tree (Text) tab of the output, which provides a detailed, textual representation of the syntactic structure for the input code `int x = 10;`. This view is crucial for

confirming that the statement successfully adhered to the compiler's grammar rules by displaying the derivation from the highest-level non-terminal symbol down to the actual tokens. The output begins with the structure's root, Program, and immediately refines this into a Statement, which, in this context, represents the variable declaration being analyzed.

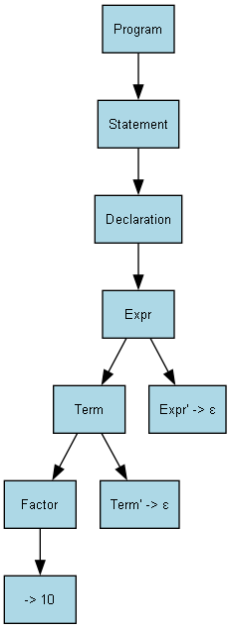


Figure 4: Abstract Syntax Tree (graphical version)

The image displays the Abstract Syntax Tree (AST) or SDT (Syntax Directed Translation) tree generated for a component of the input statement, specifically representing the structure of the expression used for initialization (the number `10`). This tree visually confirms the hierarchical structure and successful parsing of the numeric value according to the program's grammar.

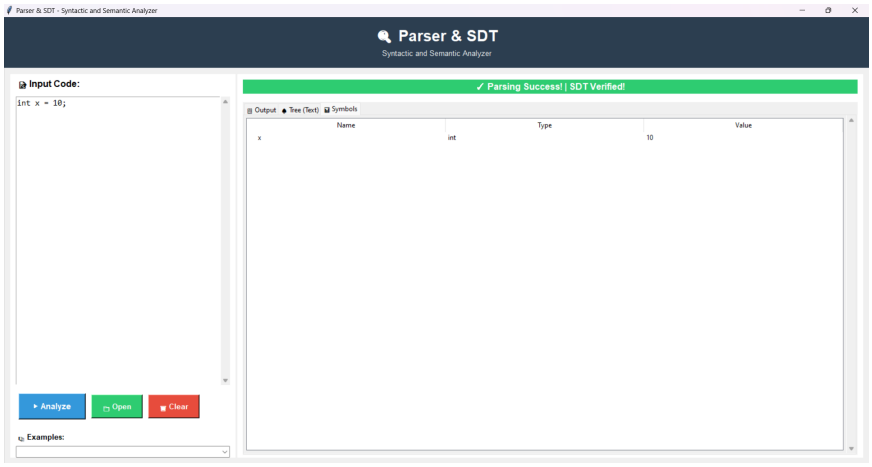


Figure 5: Symbols Table

The symbol table the single entity processed in the input code. It lists the Name of the identifier as '`x`', confirms its Type as '`int`' (integer), and records its initialized Value as '`10`'. This compact table verifies that the program correctly identified the variable, assigned it the appropriate data type as declared, and stored its initial assigned value, successfully completing the semantic processing required for compilation.

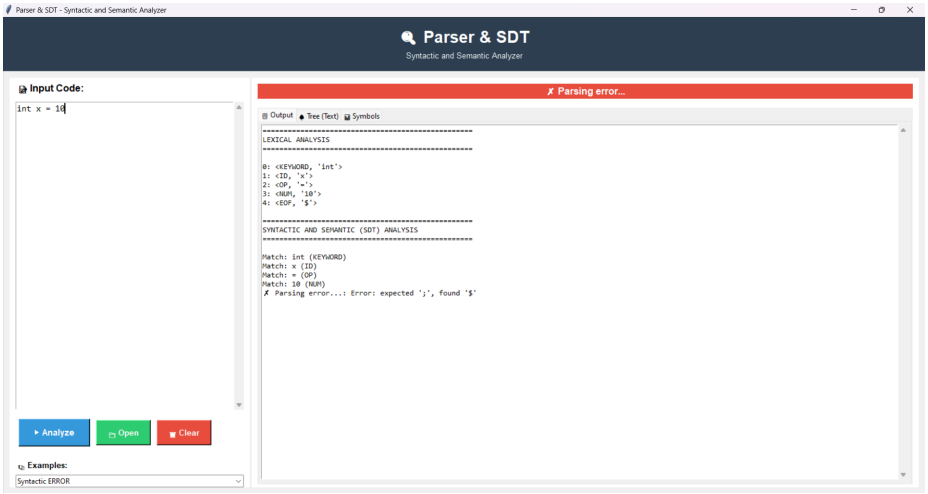


Figure 6: Parsing Error

In the first error scenario (`int x = 10`), a Parsing Error occurred because the required statement terminator was missing. The Lexical Analysis successfully tokenized the input, but the Syntactic Analysis failed when it reached the end of the input (`$`) while still expecting a semicolon (`;`) to complete the declaration statement. The program reported: "Parsing error...: Error: expected '`;`', found '`$`'".

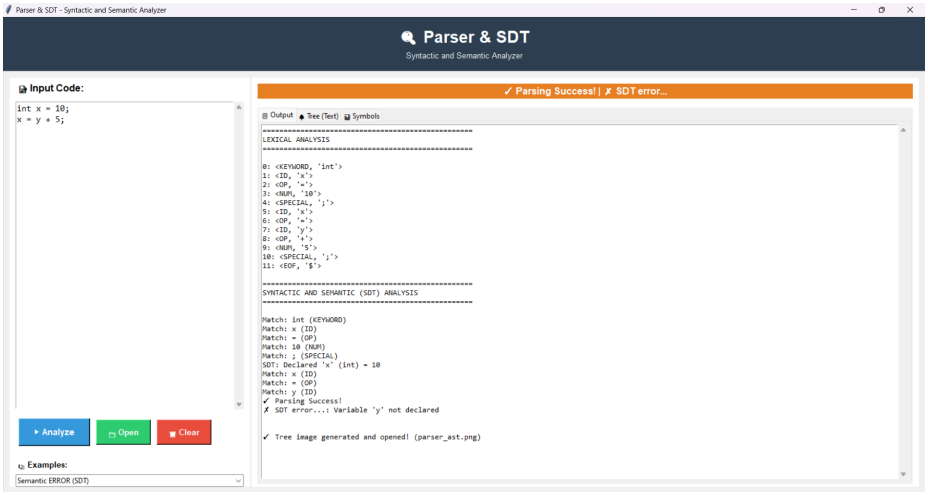


Figure 7: SDT Error

In the second scenario (`int x = 10; x = y + 5;`), the program executed both the Lexical and Syntactic analyses successfully ("Parsing Success!"). However, a Semantic

(SDT) Error was detected during the analysis of the second statement. While the statement was grammatically valid, the identifier 'y' had not been declared anywhere in the code. The semantic check correctly identified this violation of language rules, issuing the message: "SDT error... Variable 'y' not declared".

# 5 Conclusions

The Syntax and Semantic Analyzer's implementation demonstrated the effective application of compiler theory principles, particularly context-free grammars, LL(1) parsing, and Syntax-Directed Translation (SDT). The goal was to analyze variable declarations, expressions, and assignments while maintaining deterministic and efficient parsing, and the parser successfully verified the syntactic correctness of source programs, meeting the objective of analyzing variable declarations, expressions, and assignments while maintaining deterministic and efficient parsing.

By incorporating SDT, syntactically valid structures were guaranteed to possess semantic meaning by enabling the simultaneous execution of semantic actions during parsing. This integration enabled operations such as type validation, constant protection, and expression evaluation, reflecting the theoretical concept that SDT connects grammatical form with computational semantics. The construction of symbol tables and error handling mechanisms reinforced these concepts by validating variable usage and detecting semantic violations.

In relation to the defined objectives, the system satisfied the requirement of supporting multiple data types (including constant variants), establishing a functioning symbol table, and processing input from external files. The generation of both textual and graphical Abstract Syntax Trees (ASTs) confirmed the correct derivation of program structures and the successful implementation of the syntactic component. The separation between syntactic and semantic validation made it possible to distinguish programs that were structurally valid but semantically incorrect, illustrating the layered design of a compiler front end.

The system's ability to parse input correctly, build valid Abstract Syntax Trees, and identify both syntactic and semantic errors with precision showed that the project objectives were fully achieved. Each stage of the analysis worked as a separate component but remained interconnected, proving in practice how the theoretical elements—such as FIRST and FOLLOW sets, LL(1) predictive parsing, attribute flow, and context-based checks—can be integrated into a functional implementation.

Overall, the system successfully translated the theoretical foundations of formal language theory and compiler design into a functional prototype, illustrating how syntax and semantics interact to ensure correctness, consistency, and structured interpretation in program translation.

# 6 References


[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2006. [Online]. Available: https://www-2.dc.uba.ar/staff/becher/Hopcroft-Motwani-Ullman-2001.pdf.

[2] K. C. Louden and K. A. Lambert, *Programming Languages: Principles and Practice*, 3rd ed. Boston, MA: Cengage Learning, 2011, ISBN: 978-1111529413. [Online]. Available: https://nibmehub.com/opac-service/pdf/read/Programming%20Languages-%20Kenneth%20C.%20Louden%20-%203ed.pdf.

[3] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN: 1558603204. [Online]. Available: https://books.google.com.br/books?hl=es&lr=&id=Pq7pHwG1_OkC&oi=fnd&pg=PA1&dq=Advanced+Compiler+Design+and+Implementation.&ots=4-9YFrj7mL&sig=DTFl1aGWqYQgFSegeHrWfbvwbPA#v=onepage&q=Advanced%20Compiler%20Design%20and%20Implementation.&f=false.

[4] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, 2nd ed. New York, NY: Springer, 2012. [Online]. Available: https://dpvipracollege.ac.in/wp-content/uploads/2023/01/Modern.Compiler.Design.2nd.pdf.