



Universidad Nacional Autónoma de México

Ingeniería en Computación
Compilers

Parser & SDT Report

Students:

320021312

320112179

320184624

320565661

423093137

Group: 5 Semester: 2026-I

Mexico City, Mexico. November 2025

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Theoretical Framework | 3 |
| 2.1 | Syntax Analysis (Parsing) | 3 |
| 2.1.1 | Parsing Methods | 4 |
| 2.2 | Semantic Analysis and Syntax-Directed Translation (SDT) | 4 |
| 3 | Development | 5 |
| 3.1 | Grammar Definition | 5 |
| 3.2 | Syntax-Directed Translation (SDT) | 6 |
| 3.3 | Architecture and Components | 6 |
| 3.4 | Error Handling and Validation | 7 |
| 4 | Results | 8 |
| 5 | Conclusions | 11 |

1 Introduction

Modern compiler design requires precise mechanisms that transform high-level source code into structured, verifiable representations. One of the key challenges arises when transforming a flat stream of tokens—provided by a lexical analyzer—into a hierarchical structure that represents syntactic and semantic meaning. This process is critical for ensuring that code follows the formal rules of a language and that type consistency and variable usage are logically sound.

The main goal of this project is to design and implement a **Syntax and Semantic Analyzer** capable of processing token sequences produced by a previous lexical analyzer. By combining deterministic parsing with **Syntax-Directed Translation (SDT)**, the system achieves both syntactic validation and semantic checking in a single execution stage. This ensures efficiency, traceability, and consistency between the lexical and syntactic phases.

Our implementation defines a clear grammar aligned with the lexical categories (**KEYWORD**, **IDENTIFIER**, **OPERATOR**, **CONSTANT**, and **PUNCTUATION**). It applies transformations such as left-factoring and left-recursion elimination to guarantee determinism and simplicity in parsing. The integration of SDT actions provides immediate semantic validation by tracking declared identifiers and verifying assignment compatibility through a symbol table.

This report also illustrates the automaton that governs the declaration pattern, a fundamental component for verifying statements like `int x = 10;.` Figure 1 shows the DFA.

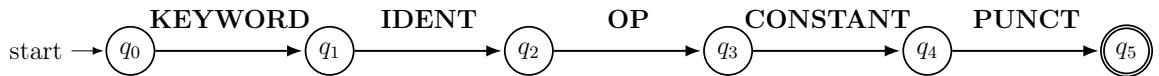


Figure 1: DFA representation for declaration statements following the pattern `TYPE id = const ;.`

The automaton defines a clear sequence of token recognition that transitions deterministically through each state, ensuring valid declarations before the parser advances to expression handling or higher-level constructs. It forms the structural basis of our syntactic validation logic.

| Component | Implementation Summary |
|----------------------|---|
| Lexical Interface | Direct integration with our lexer. Token categories and literals are preserved to keep the pipeline consistent. |
| Grammar Definition | Declarations, assignments, and arithmetic expressions; transformed to an LL(1)-friendly form. |
| Parsing Strategy | Predictive LL(1) table; if extensions introduce conflicts, an SLR(1) driver will be generated with the same CLI. |
| Semantic Layer (SDT) | Symbol table insertion/lookup and assignment type checks executed during reductions or predictive matches. |
| Testing Example | Input <code>int x = 10;</code> yields <i>Parsing Success! SDT Verified!</i> . Invalid type uses produce targeted diagnostics. |
| Error Handling | Reports the offending token, expected set, and the production context for traceability. |

Table 1: Core components of the implemented Parser and Semantic Analyzer.

2 Theoretical Framework

The development of a compiler is a multi-phase process that translates a high-level language into a low-level one [3]. The first phase of this process, lexical analysis, reads the source code, groups characters into meaningful "lexemes," and produces a sequence of tokens [3]. This project addresses the subsequent critical phases: **syntax analysis (parsing)** and **semantic analysis**.

2.1 Syntax Analysis (Parsing)

Syntax analysis, also known as parsing, is the second phase of compilation [3]. Its primary function is to take the stream of tokens generated by the lexical analyzer and verify that this sequence is syntactically correct [3]. This validation is performed by checking the tokens against the rules of a Context-Free Grammar (CFG), which is a formal grammar used to define all possible valid strings (structures) in a language [7].

The parser's main goal is to construct a hierarchical representation of the token stream, known as a parse tree (or syntax tree), which demonstrates the syntactic validity of the code [3, 6].

2.1.1 Parsing Methods

Parsing techniques are broadly categorized into two main families, based on how the parse tree is constructed [6]:

- **Top-Down Parsing:** This method constructs the parse tree starting from the root (the grammar's start symbol) and works its way down to the leaves (the tokens) [6]. A common implementation is the **Recursive Descent Parser** [6].
- **Bottom-Up Parsing:** This method works in the opposite direction. It starts with the input tokens (leaves) and attempts to "reduce" them back to the start symbol (root) [6]. The most common technique in this category is Shift-Reduce Parsing, which forms the basis for LR parsers [2, 6].

2.2 Semantic Analysis and Syntax-Directed Translation (SDT)

A successful parse only confirms that the code is syntactically valid. The third phase, semantic analysis, is responsible for checking the *meaning* and logical consistency of the code [3, 4]. It takes the parse tree as input and performs crucial checks, such as:

- **Type Checking:** Verifying that operands in an expression have compatible types (e.g., not adding a string to an integer) [4].
- **Scope Management:** Ensuring that variables are declared before they are used, often by managing a Symbol Table [4].

Syntax-Directed Translation (SDT) is a formal technique used to implement this semantic analysis. SDT works by augmenting the Context-Free Grammar: semantic rules (or actions) are attached to the grammar's productions [5]. As the parser recognizes a production (e.g., performs a reduction), it executes the corresponding semantic rule [5]. This allows the compiler to perform semantic checks, build the symbol table, and ensure the code is logically sound concurrently with the parsing process [5, 4].

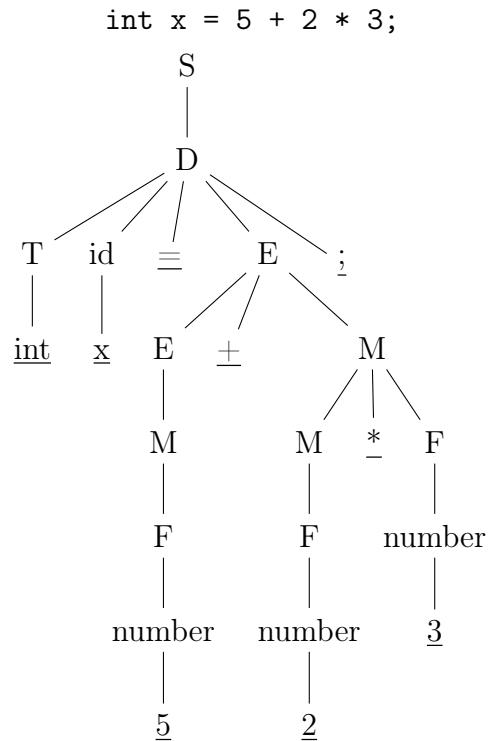
3 Development

3.1 Grammar Definition

The following grammar definition represents the one implemented by the parser.

$$\begin{aligned} S &\rightarrow D \mid P -\text{Asignación o impresión} \\ D &\rightarrow T \text{ id} = E ; -\text{Asignación} \\ T &\rightarrow \text{int} \mid \text{float} -\text{Tipo de dato} \\ P &\rightarrow \text{print} (\text{STRING}) ; -\text{Impresión} \\ E &\rightarrow E + M \mid E - M \mid M -\text{Operaciones menor jerarquía} \\ M &\rightarrow M * F \mid M / F \mid F -\text{Operaciones mayor jerarquía} \\ F &\rightarrow \text{number} \mid \text{decimal} \mid \text{id} \mid (E) \end{aligned}$$

The following diagram represents the parsing tree for an example input.



3.2 Syntax-Directed Translation (SDT)

The syntax-directed translation (SDT) rules associate semantic actions with each grammar production. These actions are executed during parsing to perform type verification and basic semantic validation. In this project, the grammar supports variable declarations, arithmetic expressions, and print statements. Each production carries an attribute that stores the computed semantic result or a Boolean value indicating whether the SDT verification succeeded.

Below, the SDT rules are shown in pseudocode form.

$$\begin{aligned} S &\rightarrow D \mid P \\ D &\rightarrow T \text{ id } = E ; \{ \text{if } (T = \text{int}) \wedge (E.type = \text{int}) \text{ then } ok \text{ else error} \} \\ P &\rightarrow \text{print(string)} ; \{ \text{print(string.value)} \} \\ E &\rightarrow E_1 + M \{ E.value = E_1.value + M.value \} \\ E &\rightarrow E_1 - M \{ E.value = E_1.value - M.value \} \\ M &\rightarrow M_1 * F \{ M.value = M_1.value * F.value \} \\ M &\rightarrow M_1 / F \{ \text{if } F.value = 0 \text{ then error else } M.value = M_1.value / F.value \} \\ F &\rightarrow \text{number} \mid \text{decimal} \mid \text{id} \mid (E) \end{aligned}$$

3.3 Architecture and Components

The system was developed using the Python Lex-Yacc (PLY) library, which provides tools to define both the lexical and syntactic analysis phases of a compiler. The project is structured into three main components: the lexical analyzer, the parser, and the main file. Each file fulfills a specific role in the implementation of the syntax-directed translation (SDT) process.

- **lexer.py:**

This file defines the lexical analyzer, responsible for identifying tokens such as reserved words (`int`, `float`, `print`), identifiers, numeric literals (integers and decimals), operators, and delimiters. It also handles lexical errors by printing an error message whenever an unrecognized symbol is found, ensuring that only valid tokens are passed to the parser.

- **parser.py:**

This module defines the grammar rules and their associated semantic actions according to the principles of syntax-directed translation. Each production rule is implemented as a Python function that computes semantic attributes or

returns Boolean values to indicate whether semantic verification succeeded. The parser is also responsible for managing operator precedence, detecting semantic errors, such as type mismatches or division by zero, and controlling the final output messages: "*Parsing Success! SDT Verified!*" or "*Parsing Success! SDT error...*" depending on the result.

- **main.py:**

The main driver program serves as the user interface for testing. It initializes the lexer and parser, and allows the user to enter statements via the command terminal. Finally it shows the parsing and SDT result for the given string.

The general workflow of the system is as follows: first, the lexer analyzes the input stream and generates tokens; then, the parser checks whether the token sequence conforms to the defined grammar; finally, the corresponding SDT actions are executed to verify semantic correctness.

3.4 Error Handling and Validation

The following table summarizes the main types of errors detected and the messages generated by the program.

Table 2: Classification of errors detected by the parser and SDT system.

| Error Type | Example | Description |
|----------------------------------|--------------------------------|--|
| Lexical Error | <code>int \$x = 10;</code> | Invalid character detected by the lexer. The system reports Illegal character '\$' |
| Syntax Error | <code>int x = 10</code> | Missing semicolon or incorrect token order. Reported as Parsing error in token '10' . |
| Semantic Error. Type mismatch | <code>int x = 3.14;</code> | Type mismatch between declared variable and assigned value. Reported as SDT error.... |
| Semantic Error. Division by zero | <code>float y = 10 / 0;</code> | Arithmetic evaluation with division by zero. Detected by SDT rules and reported as a semantic error. |

4 Results

Execution 1 — Startup and readiness. When running `python src/main.py`, the program initializes the parser module. The message `Generating LALR tables` indicates that the grammar has been successfully compiled into parsing tables used by the LR engine. Immediately after, the prompt `Parser & SDT` confirms that both the syntactic and semantic modules have been loaded. Finally, the instruction `Escribe 'exit' para salir.` appears, meaning the program has entered its interactive mode and is waiting for user input. At this stage, no tokens are yet parsed — the system is simply confirming correct initialization and readiness to process statements.

```
PS C:\Users\echav\OneDrive\Documentos\Compiladores\unam.fi.compilers.g5.06-main\parser> python src/main.py
Generating LALR tables
Parser & SDT
Escribe 'exit' para salir.
```

Figure 2: First execution.

Execution 2 — Project layout and generated artifacts. After the first run, the parser package shows source files and build products. This confirms that the grammar was compiled and the LR tables were written: `lexer.py` (tokenizer), `parser.py` (grammar + SDT), `parsetab.py` (auto-generated LALR tables), `parser.out` (LR automaton report), `main.py` (CLI), and `__pycache__` (bytecode cache).

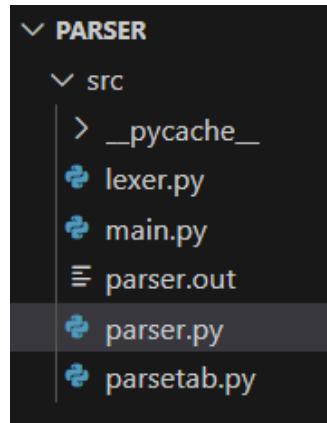


Figure 3: Project layout: source code and generated artifacts (`parsetab.py`, `parser.out`).

Execution 3 — Parsing and semantic validation. At this stage, the parser receives the input `int x = 10;` and proceeds through the deterministic grammar. The sequence of tokens (KEYWORD, IDENT, OP, CONST, PUNCT) is recognized according

to the DFA structure. Once the parse tree is successfully built, the SDT actions verify that the variable `x` is compatible with the declared type `int`. The symbol table is updated, and the console confirms both syntactic and semantic correctness with the messages `Parsing Success!` and `SDT Verified!`. This demonstrates full alignment between the lexical, syntactic, and semantic phases of the compiler.

```
>> int x = 10;
Parsing Success!
SDT Verified!
```

Figure 4: Successful parsing and semantic validation of the statement `int x = 10;`.

Execution — Syntactic error: missing semicolon. The input `int x = 10` omits the terminal `;` required by the production $D \rightarrow T \ id = E ;$. The token stream ends after `CONSTANT` and violates the expected FOLLOW of the declaration. The LL(1)/SLR driver therefore reports `Parsing error...` before any SDT actions are executed.

```
>> int x = 10
Parsing error...
```

Figure 5: Parsing error caused by the missing semicolon in the input `int x = 10`.

Execution — Semantic error: type mismatch. In this case, the parser successfully recognizes the full declaration `int x = 10.5;`, so syntactic validation passes. However, the SDT actions detect an inconsistency between the declared type `int` and the assigned expression's evaluated type `float`. According to the semantic rule $D \rightarrow T \ id = E ; \{if (T = int) \wedge (E.type = int) \text{ then ok else error}\}$, the mismatch triggers a semantic violation. As a result, the console reports `Parsing Success!` followed by `SDT error... (Type mismatch)`, indicating a semantic fault despite a successful parse.

```
>> int x = 10.5;
Parsing Success!
SDT error... (Type mismatch)
```

Figure 6: Parsing success but SDT error due to type mismatch in the declaration `int x = 10.5;`.

Execution — Declaration with float and consistent assignment. The input `float i6 = 7.2;` is tokenized as KEYWORD-IDENT-OP-CONST-PUNCT. The parser recognizes the production $D \rightarrow T id = E$; without conflicts. SDT actions evaluate E as `float` and check that it matches the declared type `float`. The symbol table records the entry (`i6`, `float`, `7.2`), so the console reports `Parsing Success!` followed by `SDT Verified!`.

```
>> float i6=7.2;
Parsing Success!
SDT Verified!
```

Figure 7: Successful parsing and SDT verification for `float i6 = 7.2;`.

Execution — Printing a literal via SDT action. For the input `print("Example");` the token stream is KEYWORD-PUNCT(()-STRING-PUNCT())-PUNCT();. The parser matches the production $P \rightarrow \text{print}(\text{STRING})$;. Its attached SDT action emits the literal to the console, hence the line `Output: Example`. Because the structure is syntactically valid and the argument type is appropriate for `print`, the run ends with `Parsing Success!` and `SDT Verified!`.

```
>> print("Example");
Output: Example
Parsing Success!
SDT Verified!
```

Figure 8: Print statement: SDT emits the string literal and confirms syntactic and semantic validity.

Execution — Exiting the REPL. Typing `exit` ends the interactive loop. The driver closes the input stream, releases parser resources (tables cached on disk remain available as `parsetab.py` and `parser.out`), and returns control to the operating system prompt. No parsing or SDT actions are performed for this command; it is processed as a shell instruction to gracefully terminate the session.

```
>> exit
PS C:\Users\echav\OneDrive\Documentos\Compiladores\unam.fi.compilers.g5.06-main\parser> █
```

Figure 9: Session termination: the REPL ends on `exit` and returns to the shell.

5 Conclusions

The developed Syntax and Semantic Analyzer successfully integrates grammar-based parsing with immediate semantic verification. By building upon the lexer output, it bridges the lexical and syntactic phases, ensuring that each recognized statement satisfies both grammatical and type rules. Deterministic parsing and embedded SDT actions provide auditable results and stable behavior.

The error management mechanism proved to be effective in isolating and reporting each category of error precisely. This modular design ensured that illegal characters, grammar violations, and semantic inconsistencies such as type mismatches or divisions by zero were all detected and reported clearly to the user.

This project validates the educational and practical value of PLY as a tool for understanding the principles of syntax analysis, semantic validation, and compiler construction.

The implementation of the syntax analyzer using PLY successfully demonstrated the integration of lexical, syntactic, and semantic analysis under a unified architecture. Through the definition of grammar rules and SDT actions, the system was able to validate the syntactic structure of input statements and perform type checking and arithmetic verification dynamically.

References

- [1] David Beazley. *PLY (Python Lex-Yacc) Documentation, Version 3.11*, 2011. Accessed: November 1, 2025.
- [2] GeeksforGeeks. Bottom-up parser. <https://www.geeksforgeeks.org/compiler-design/bottom-up-or-shift-reduce-parsers-set-2/>. Accessed: November 2, 2025.
- [3] GeeksforGeeks. Phases of a compiler. <https://www.geeksforgeeks.org/compiler-design/phases-of-a-compiler/>. Accessed: November 2, 2025.
- [4] GeeksforGeeks. Semantic analysis in compiler design. <https://www.geeksforgeeks.org/compiler-design/semantic-analysis-in-compiler-design/>. Accessed: November 2, 2025.
- [5] GeeksforGeeks. Syntax directed translation in compiler design. <https://www.geeksforgeeks.org/compiler-design/syntax-directed-translation-in-compiler-design/>. Accessed: November 2, 2025.
- [6] GeeksforGeeks. Types of parsers in compiler design. <https://www.geeksforgeeks.org/compiler-design/types-of-parsers-in-compiler-design/>. Accessed: November 2, 2025.
- [7] GeeksforGeeks. What is context-free grammar? <https://www.geeksforgeeks.org/theory-of-computation/what-is-context-free-grammar/>. Accessed: November 2, 2025.