Universidad Nacional Autónoma de México
Facultad de Ingeniería

Computer Engineering

Subject: Compilers (0434)

Account numbers:
320075669
423066940
423035975
320199866
320061682

Group: 5

Semester: 2026 - I

Mexico, CDMX. November 3rd 2025

Índice

Compilers
Group: 5
UNAM / FI / DIE
Semestre 2026-1

Parser & SDT

1. Introduction

1.1. Problem Statement

The challenge is to construct a **Syntactic Analyzer (Parser)** integrated with a **Syntax-Directed Translation Scheme (SDT)** capable of processing an input sequence of tokens. The system must perform two critical checks: first, verifying if the token sequence is syntactically valid against a defined grammar, and second, simultaneously verifying its semantic compliance using the SDT rules. The core difficulty lies in implementing a robust procedure that builds a parse tree and verifies both the structural (syntactic) and logical (semantic) correctness of the input, yielding a clear pass/fail state for each analysis.

1.2. Justification

The successful completion of this task is essential as it reinforces the comprehension of the theoretical foundations of compiler design, specifically the crucial syntactic and semantic analysis phases. It requires students to apply fundamental concepts, such as choosing a parsing method (*Top-Down* or *Bottom-Up*), defining grammar productions, and creating SDT rules for semantic actions.

1.3. Objectives

Overall Objective

To design and implement an integrated system comprising a Parser and a Syntax-Directed Translation Scheme (SDT) that successfully processes tokenized input to verify both syntactic validity and semantic compliance.

Specific Objectives

1) Define a formal grammar whose productions precisely match the input tokens provided by the lexical analyzer.

2) Select and implement a specific parsing method (*Top-Down* or *Bottom-Up*) and define the corresponding SDT rules for semantic analysis.

3) Implement a procedure to read the token input, construct a parse tree, and verify the satisfaction of all defined SDT rules.

4) Ensure the application produces a clear, distinguishable output for all possible scenarios: successful parsing and verified SDT, parsing success but SDT failure, or parsing error.

2. Theoretical Framework

2.1. YACC (Yet Another Compiler Compiler)

YACC is a classic tool that generates a parser from a formal grammar specification, developed at the beginning of the 1970s by Stephen C. Johnson for the Unix operating system. It takes a CFG as input

and produces a C program that implements a push-down automaton to parse the language described by that grammar. A concise definition of this is:

> "Yacc provides a general tool for describing the input to a computer program, together with code to be invoked as each such structure is recognized, and turns such a specification into a subroutine that handles the input process." (Johnson, 1975).

In this project, we use **PLY**, which is a Python implementation of both LEX and YACC. The `ply.yacc` module provides the same core functionality as YACC but within the Python programming environment.

2.2. Key Concepts and Features of YACC

Core ideas and characteristics of YACC:

- YACC takes a context-free grammar as input, usually in Backus-Naur Form, which describes the syntax rules of the language it will parse.
- It translates this grammar into a C function that efficiently parses input text according to the predefined rules.
- YACC uses LALR(1) parsing, which is a bottom-up method that utilizes a single token of lookahead to determine the next parsing action.
- Semantic actions are code associated with grammar productions. This code is executed when a rule is recognized, enabling the construction of abstract syntax trees, generation of intermediate representations, or error handling.
- It supports attribute grammars, where non-terminal grammar symbols have attributes that are manipulated by semantic actions, often used in constructing parse trees or outputting code.
- YACC is often used together with Lex, a tool that generates lexical analyzers. Lex breaks the input into tokens that are then processed by the YACC parser.

2.3. Structure of a YACC Program

A YACC program is traditionally split into three sections, separated by `%%`:

```
/* definitions */
....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

1: Structure of a YACC Program

Declarations

This section contains C declarations, delimited by `%{` and `%}`, and YACC declarations for tokens, operator precedence, and associativity. This initial block allows specifying all the terminal symbols that will be utilized in the syntax structure:

```
%token NUMBER
  %token ID
```

While YACC automatically manages the internal integer values assigned to each token, it is possible to explicitly define a unique numerical ID for a token if necessary, for example: `%token NUMBER 621`.

YACC inherently treats literals as tokens based on their respective ASCII codes. Therefore, any custom numerical values manually assigned to tokens must be chosen so that they do not clash with standard character codes.

Programmers can insert external C source code, global definitions, and variable declarations directly into this part of the specification by encompassing them within the special delimiters % and % positioned at the first column. This section is also used to explicitly indicate the nonterminal that serves as the root or main entry point for the grammar analysis:

```
%start nonterminal
```

In the PLY implementation (*PARSER_C.py*), this corresponds to importing the `tokens` from the lexer and defining the `precedence` tuple.

Rules

This is the core of the parser, containing the grammar productions. Each rule consists of a non-terminal followed by a colon, a sequence of terminals and non-terminals, and an optional action. The fundamental format for defining a structural component of the language is based on a modified Backus-Naur Form:

```
nonterminal : sequence_of_symbols { semantic_action };
```

Where:

- **Nonterminal**: This represents the general concept or structure being defined (e.g., expression, statement, list).
- **Sequence of Symbols**: This is the definition of the structure, consisting of a series of terminal symbols and other nonterminal symbols.
- **Punctuation**: The definition is separated from the nonterminal by a colon (:) and the entire rule or group of alternatives are terminated by a semicolon (;).

When a single nonterminal can be constructed in multiple ways, the pipe symbol (|) can be used to list the alternatives without repeating the nonterminal symbol:

```
expression : expression '+' term | term;
```

The C code enclosed in curly braces is the semantic action. This code is executed immediately when the YACC parser recognizes the complete sequence of symbols on the right-hand side of the rule. Semantic actions are crucial for processing data and building the final output. They use special pseudovariables to access and assign attribute values:

- **$$**: Represents the semantic value of the nonterminal on the left-hand side. The action must assign a value to $$ to pass the result of the reduction up the parse tree.
- **$1, $2, $3, ...**: Represent the semantic values associated with the symbols in the right-hand sequence, counted sequentially from left to right.

In PLY, rules are defined as Python functions with the grammar rule in the function's docstring. The "action" is the Python code within the function body.

Auxiliary Functions

This third and final section contains supporting C code that YACC does not generate itself but is required for the functional operation of the synthesized parser, such as `main()`, `yylex()`, and `yyerror()`.

The most critical routines that must be defined are:

Function	Purpose
`main()`	The primary entry point for the entire executable program. It must typically call the parser function: `return yyparse();`.
`yylex()`	The **Lexical Analyzer** function. This function is called by the YACC parser (`yyparse()`) whenever it needs the next input token. It must return an integer value representing the type of token found.
`yyerror(const char *s)`	The **Error Handling** routine. This function is invoked by the parser when it encounters a syntax error (i.e., when the input does not match any grammar rule). It usually prints an error message.

Table 1. Main YACC functions and their purpose.

Beyond the essential functions, this section can contain any other programmer-defined C functions, global variables, or libraries that are referenced by the semantic actions. In this project, this corresponds to the additional Python functions defined, such as `generate_syntax_tree_image`, `evaluate_expression`, and the main `parse_code` function.

2.4. Shift-Reduce Parsing

YACC generates a shift-reduce parser. This parser operates by maintaining a stack and reading tokens from the input buffer. It repeatedly performs one of four actions:

- **Shift:** Move the next input token onto the top of the stack. "During the shift operation, the parser reads the next input token and pushes it onto a stack. The parser maintains a buffer to hold the remaining input tokens. The shift operation moves the parser's current position to the right in the input stream."(Ahmed H, 2023).
- **Reduce:** When the symbols on top of the stack match the right-hand side of a grammar rule, they are "reduced" to the rule's left-hand side non-terminal. The semantic action associated with that rule is executed at this time.
- **Accept:** When the input is exhausted and the stack contains only the start symbol, the program is syntactically valid. Ïf the stack contains the start symbol only and the input buffer is empty at the same time then that action is called accept."(Ahmed H, 2023).
- **Error:** If the parser reaches a state where none of the other actions are possible, a syntax error is reported. .[A] situation in which the parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called an error action."(Ahmed H, 2023).

2.5. Conflict Resolution

Ambiguous grammars can lead to conflicts where the parser does not know which action to take.

- **Shift/Reduce Conflict:** The parser cannot decide whether to shift the next token or reduce a rule already on the stack. This is most common with expressions.
- **Reduce/Reduce Conflict:** The parser cannot decide between two or more possible reductions.

YACC and PLY allow us to resolve shift/reduce conflicts by declaring operator precedence and associativity. In *PARSER_C.py*, this is defined by the `precedence` tuple. By listing operators from lowest to highest precedence, we instruct the parser on how to resolve these ambiguities.

2.6. Semantic-Directed Translation (SDT)

SDT is the process of executing semantic actions as a side-effect of parsing. In YACC and PLY, the action associated with a rule is executed when that rule is reduced. This allows the parser to do more than just validate syntax. In our project, we use these actions to:

A) **Build the AST:** The action for a rule typically creates a new node for the AST (in our case, a tuple) and assigns it to `p[0]`, which represents the value of the rule's head.

B) **Perform Semantic Checks:** We use actions to interact with our `SymbolTable` class, checking if a variable is declared before use (`get_symbol` function) or adding a new variable upon declaration (`add_symbol` function).

3. Body of Work (Development)

Our compiler's parsing and semantic analysis logic is implemented in *PARSER_C.py*, which works in conjunction with the lexer defined in *LEX_C.py*.

3.1. Lexer-Parser Integration

The parser (*PARSER_C.py*) is tightly coupled with the lexer (*LEX_C.py*). The general functioning is:

A) The parser imports the `tokens` list directly from the lexer module. This ensures both components agree on the set of valid terminals.

B) The `yacc.yacc()` function automatically looks for the `yylex()` function, which is provided by the `ply.lex` object created in *LEX_C.py*.

C) The main `parse_code` function first calls `analyze_code` from the lexer to get the token list and check for lexical errors before passing the code to the `parser.parse()` method.

3.2. Grammar

The parser implements a CFG for a simplified C language. The rules are defined by the `'p_'` functions. The key structures of our grammar are:

- `program`: The start symbol. It must be a `main` function, defined as:

  ```
  'program : INT ID LPAREN RPAREN LBRACE statements RBRACE'
  ```

  The action for this rule semantically checks if the `ID` (p[2]) is "main".

- `statements`: A list of one or more `statement` non-terminals.

- `statement`: A `statement` can be one of the following: `declaration`, `assignment`, `if_statement`, `printf_statement`, or `return_statement`.

- `declaration`: A type followed by an ID and a semicolon:

  ```
  'declaration : tipo ID SEMICOLON'
  ```

- `assignment`: An ID, an equals sign, an expression, and a semicolon:

  ```
  'assignment : ID IGUALS expression SEMICOLON'
  ```

- `if_statement`: The grammar supports both `if` and `if-else` structures:

```
'if_statement : IF LPAREN expression RPAREN LBRACE statements RBRACE
              | IF LPAREN expression RPAREN LBRACE statements RBRACE
              ELSE LBRACE statements RBRACE'
```

- `expression`: Defined recursively to handle binary operations respecting precedence, parentheses, numbers, and variable IDs.

3.3. Semantic Analysis and Symbol Table

A core feature of our parser is its ability to perform semantic analysis using SDT. This is achieved with the `SymbolTable` class.

- `SymbolTable` **Class:** This class maintains a dictionary of `symbols`. It provides three key methods:
  - `add_symbol(name, type, value=None)`: Adds a new variable to the table. It raises a `SemanticError` if the variable `name` is already declared.
  - `get_symbol(name)`: Retrieves a variable's information. It raises a `SemanticError` if the `name` has not been declared.
  - `update_symbol(name, value)`: Updates the value of an existing variable.

SDT in Practice:

- **Declarations:** When `p_declaration` is reduced, its action calls `symbol_table.add_symbol(p[2], p[1])` inside a `try...except` block. If the variable is re-declared, the exception is caught and a semantic error is appended to the `semantic_errors` list.
- **Variable Use:** When `p_assignment` or `p_expression_id` are reduced, their actions first call `symbol_table.get_symbol(p[1])` (where `p[1]` is the ID) to ensure the variable exists. If it doesn't, the `SemanticError` is caught and reported.
- **Value Tracking:** The `p_assignment` action also calls `evaluate_expression` on the right-hand side (`p[3]`) and then uses `symbol_table.update_symbol` to store the new value.

3.4. AST Generation and Visualization

As the parser reduces rules, it builds an Abstract Syntax Tree (AST). The AST is represented as a nested set of tuples.

- **AST Construction:** Each rule's action assigns a tuple to `p[0]`. For example:
  - `p_assignment`: `p[0] = ('assignment', p[1], p[3])`
  - `p_expression_binop`: `p[0] = ('binop', p[2], p[1], p[3])`
  - `p_program`: `p[0] = ('program', p[6])`

  . This creates a single, large tuple that represents the entire program structure.
- **Visualization:** The `generate_syntax_tree_image` function is called after a successful parse. This function:

  A) Checks if `GRAPHVIZ_AVAILABLE` is true.

  B) Creates a `Digraph` object.

C) Recursively traverses the AST tuple (`result`).

D) For each node type (e.g., 'program', 'declaration', 'binop'), it adds a formatted node to the graph with Spanish labels (e.g., 'programa', 'declaracion', 'asignacion').

E) It renders the final graph as a JPG file named *arbol_sintactico.jpg*.

3.5. Error Handling

The parser handles three types of errors:

A) **Lexical Errors:** Detected by `t_error` in *LEX_C.py* and collected by `get_lexical_errors()`.

B) **Syntax Errors:** Handled by the `p_error(p)` function in the parser. When PLY enters an error state, it calls this function, which appends a formatted error message including the token value and line number to the `semantic_errors` list.

C) **Semantic Errors:** Handled by the `try...except SemanticError` blocks in the grammar rule actions, as described in section 3.3.

The final output of the `parse_code` function reports on all three error types, clearly distinguishing between syntactic failure and semantic failure.

4. Results

This section presents the results obtained from executing the developed compiler, which processes the source code through its Lexical, Syntactic, and Semantic analysis phases. The graphical interface (GUI), built with Tkinter and ttkbootstrap, facilitates the testing and visualization of results.

4.1. Source Code Input

The following test program was used to validate the full functionality of the compiler, featuring variable declaration, arithmetic assignment, and a conditional if-else structure.

```
int main() {
    int x;
    x = 10 * 2 + 1;
    if (x > 10) {
        printf("x es mayor que 10");
    } else {
        printf("x es menor que 10");
    }
}
```

Test Source Code Analyzed

4.2. Analysis Procedure and Lexical Results

The analysis procedure began successfully, with the Lexical Analyzer processing the input file and categorizing all components into recognized tokens.

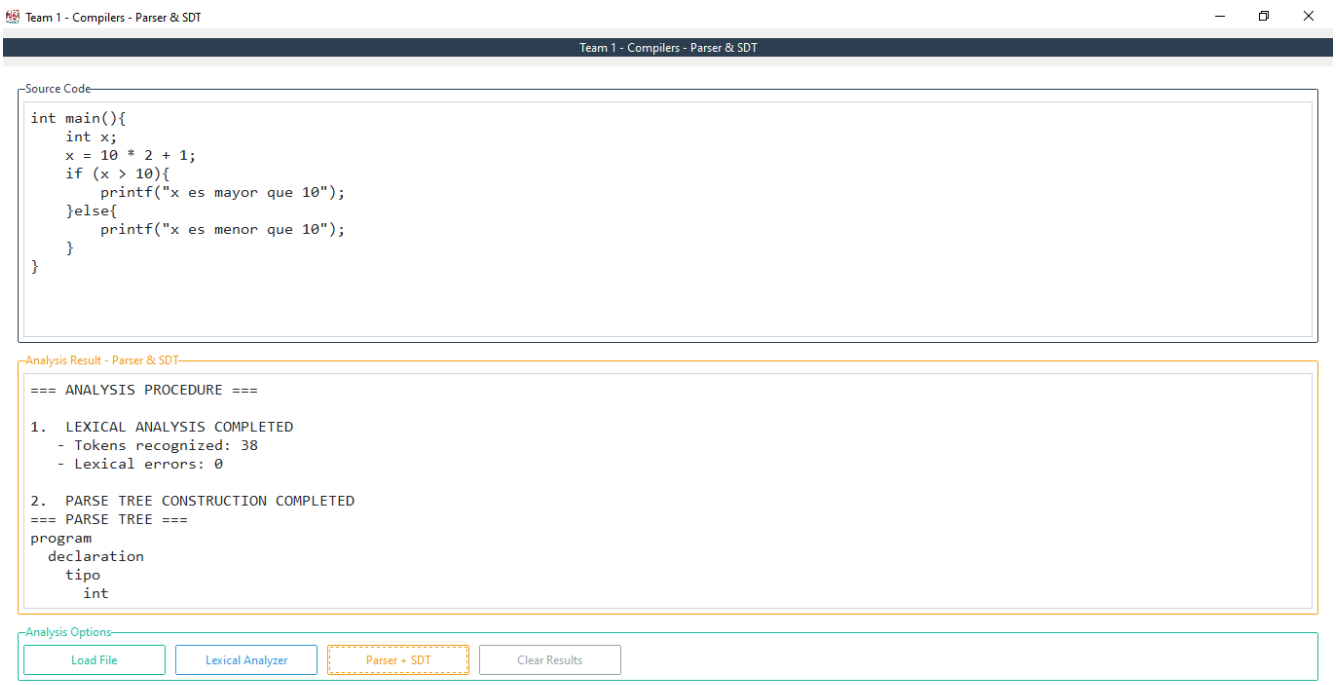

Figure 1. Analysis Results for Lexical Analysis

Furthermore, to verify this, we can go to the Lexical Analyzer option for a more detailed description, which will output:

```
=== RECOGNIZED TOKENS ===
Total tokens: 38
1: INT = 'int'
2: ID = 'main'
3: LPAREN = '('
4: RPAREN = ')'
5: LBRACE = '{'
6: INT = 'int'
7: ID = 'x'
8: SEMICOLON = ';'
9: ID = 'x'
10: IGUALS = '='
11: NUMBER = '10'
12: TIMES = '*'
13: NUMBER = '2'
14: PLUS = '+'
15: NUMBER = '1'
16: SEMICOLON = ';'
17: IF = 'if'
18: LPAREN = '('
19: ID = 'x'
20: GT = '>'
21: NUMBER = '10'
22: RPAREN = ')'
23: LBRACE = '{'
24: PRINTF = 'printf'
25: LPAREN = '('
26: STRING = 'x es mayor que 10'
27: RPAREN = ')'
28: SEMICOLON = ';'
29: RBRACE = '}'
30: ELSE = 'else'
31: LBRACE = '{'
32: PRINTF = 'printf'
33: LPAREN = '('
34: STRING = 'x es menor que 10'
35: RPAREN = ')'
36: SEMICOLON = ';'
37: RBRACE = '}'
38: RBRACE = '}'
```

Recognized tokens full output

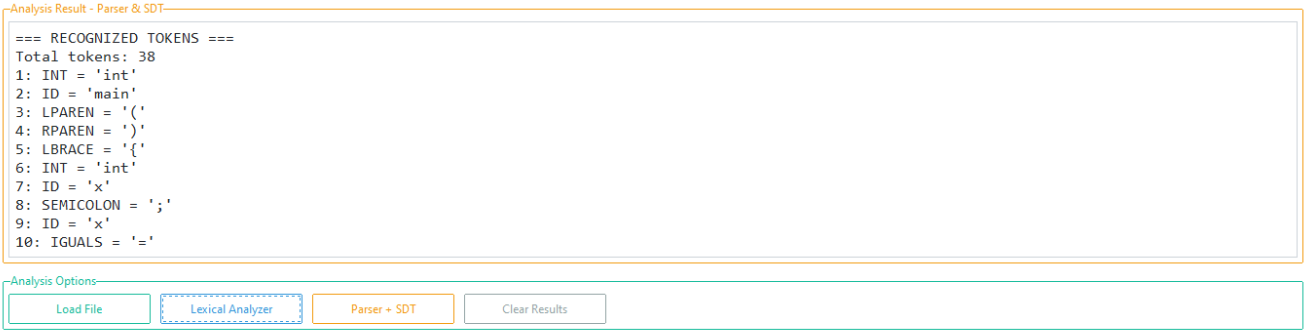

Figure 2. Recognized Tokens GUI output

This confirms the proper functioning of the tokenizer in identifying keywords, identifiers, operators, and literals according to the defined grammar.

4.3. Syntactic and Semantic Verification

Parse Tree Construction

The Syntactic Analyzer received the 38 tokens and successfully constructed the Parse Tree, verifying the grammatical structure of the program. The console output showed the successful hierarchical decomposition of the code structure:

```
=== ANALYSIS PROCEDURE ===
2.  PARSE TREE CONSTRUCTION COMPLETED

=== PARSE TREE ===
program
  declaration
    tipo
      int
    x
  assignment
    x
    binop
      +
      binop
        *
        number
          10
          number
            2
        number
          1
  if
    binop
      >
      id
        x
      number
        10
    printf
      x es mayor que 10
    else
      printf
        x es menor que 10
```

Hierarchical decomposition of the code structure

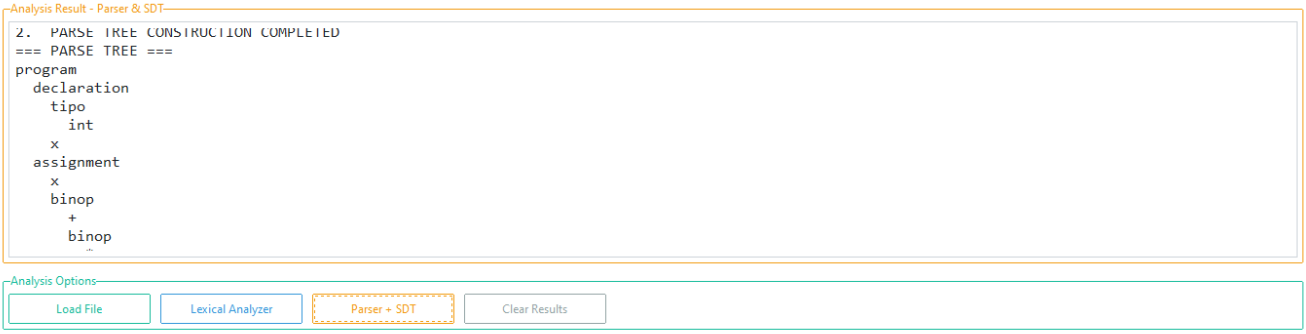

Figure 3. Parse Tree Construction

SDT Rules Verification

The Semantic Analysis stage applied the Syntax Directed Translation (SDT) rules to validate the code's logical correctness.

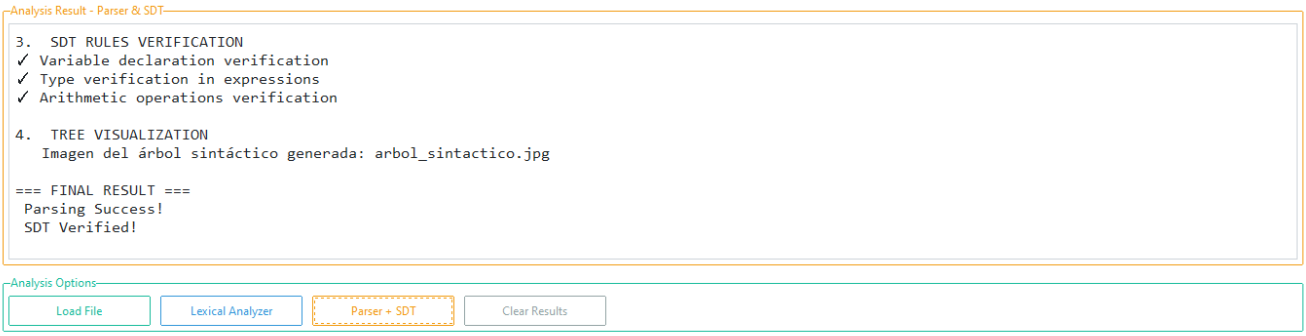

Figure 4. STD Rules Verification

The process successfully finished, confirming semantic validity: `"SDT Verified!"`. The Symbol Table information shows the successful registration of the variable x:

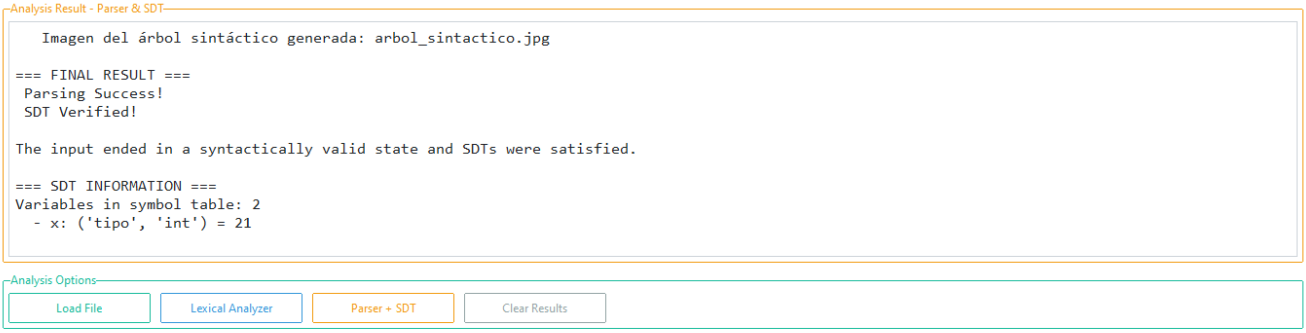

Figure 5. STD Information

Tree Visualization

The system generated a visual representation of the tree, confirming the correct hierarchy. The visualization clearly separates the nodes for declaration, assignment, and the if conditional block.

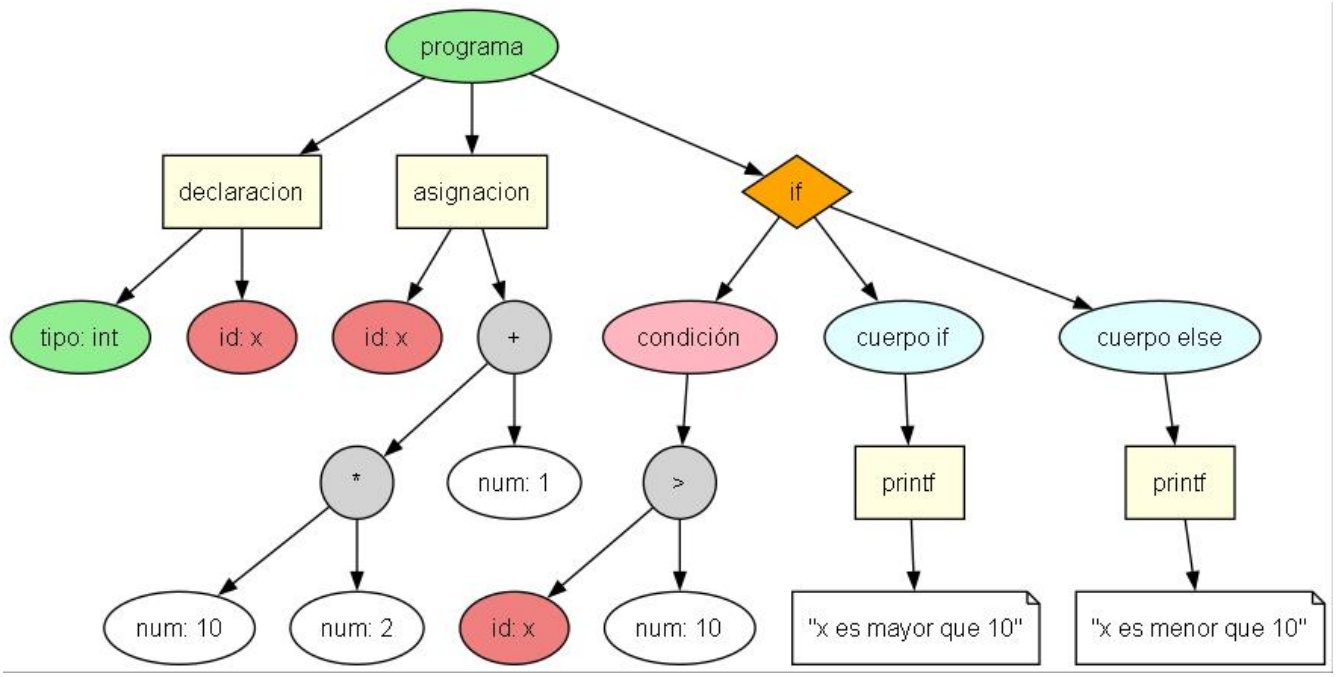

Figure 6. Generated tree: `arbol_sintactico.jpg`

4.3.1. Final message

Finally, a message is obtained indicating a successful analysis and a verified STD.

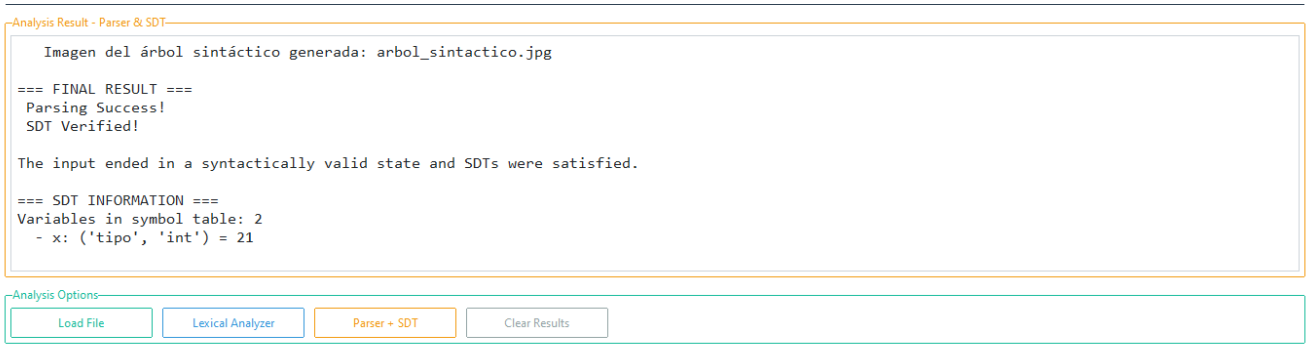

Figure 7. Final message

4.4. Error handling

Now we introduce another source code, but this one will have errors.

```
1 int main() {
2     int x,a=2,b=3,c=5;
3     x = a+b*c;
4     printf("The value of x is %d", x);
5     return 0;
6 }
```

Test Source Code Analyzed

The failure observed in the compiler output is a Syntactic Error (Parsing Error), meaning the input code does not conform to the established grammar rules of the language being analyzed. The analysis procedure halts at the second stage:

```
=== ANALYSIS PROCEDURE ===

1.  LEXICAL ANALYSIS COMPLETED
   - Tokens recognized: 35
   - Lexical errors: 8

2.  PARSE TREE CONSTRUCTION FAILED

=== FINAL RESULT ===
 Parsing error...
 SDT error...

The input did NOT end in a syntactically valid state.

=== SDT INFORMATION ===
Variables in symbol table: 0
```

Parsing error messages

The key message is:

`The input did NOT end in a syntactically valid state.`

The parser failed because the line declaring and initializing the variables contains a syntax error according to most standard C or C-like language grammars:

`Problematic Line: int x,a=2,b=3,c=5;`

The grammar likely expects a list of identifiers for declaration (e.g., int x, a, b, c;) OR a single declaration and initialization (e.g., int x; int a = 2;). The combination of declaring a variable without initialization (x) alongside multiple initializations separated by commas (a=2, b=3, c=5) is not supported by the language's syntax rules.

In conclusion, the compiler successfully identified all tokens (Lexical Analysis Completed), but the Parser failed because the statement for declaring and initializing multiple variables (int x,a=2,b=3,c=5;) violates the defined grammar rules. This demonstrates the correct functionality of the Syntactic Analyzer in rejecting invalid programs.

5. Conclusions

The successful design and implementation of the integrated Parser and Syntax-Directed Translation (SDT) system confirm that all core project objectives were met. The evidence presented in the results section demonstrates the system's capability to correctly process a tokenized input, construct a valid parse tree, and simultaneously validate the code's logical and semantic rules.

This project was a practical application of fundamental compiler theory. The specific objectives were achieved by:

A) Defining a formal grammar that precisely matched the input tokens from the lexer.

B) Implementing a bottom-up LALR(1) parsing method (via `PLY`), which proved effective for the defined grammar.

C) Integrating SDT rules directly into the grammar productions. This was the most critical step, as it bridged the gap between mere syntactic validation and true semantic analysis.

The implementation of the `SymbolTable` was central to the SDT's success, allowing the parser to move beyond simple pattern matching to perform context-sensitive checks, such as verifying variable declarations before their use.

Furthermore, the system's ability to clearly differentiate between a successful parse, a semantic failure, and a syntax error as shown in the results section, fulfills the final objective of creating a robust tool with clear, distinguishable outputs. Ultimately, this work reinforces a comprehensive understanding of the compiler's core analysis phases and the intricate relationship between syntactic structure and semantic meaning.

References

[1] GeeksforGeeks. (2025, July 11). *Introduction to YACC*. Retrieved November 2, 2025, from `https://www.geeksforgeeks.org/compiler-design/introduction-to-yacc/`

[2] S. C. Johnson. (1975). *Yacc: Yet Another Compiler-Compiler*. Computing Science Technical Report No. 32, Bell Laboratories.

[3] Ramele, R. [Rodrigo Ramele]. (2021, December 30). *Compiladores. Intro a Yacc* [Video]. YouTube. `https://www.youtube.com/watch?v=UMr9vmxKuc8`

[4] Levine, J. R., Mason, T., & Brown, D. (1992). *lex & yacc* (2nd ed.). O'Reilly Media.

[5] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compiladores: Principios, técnicas y herramientas* (2nd ed.). Pearson Education, Inc.

[6] Beazley, D. M. (n.d.). *PLY (Python Lex-Yacc) documentation*. Retrieved october 30, 2025, from: `https://www.dabeaz.com/ply/ply.html`

[7] Ahmed H, I. (2023, July 3). *Shift Reduce Parser |Compiler Design*. Medium. `https://medium.com/@izhan_a1/shift-reduce-parser-compiler-design-41dbf88779ba`