National Autonomous University of Mexico

Faculty of Engineering

Computer Engineering

Compilers

Project 2: Syntax and Semantic Analyzer

Professor: Eng. Rene Adrian Davila Perez

Semester: 2025-1

Team Members 07:

319050390

319103795

319024045

319240731

319321935

September 10, 2024

Contents

1 Introduction

What is the importance of a syntax analyzer and semantic analyzer?
The development of a parser and a Syntax-Directed Translation (SDT) system is an experience that combines the precision of mathematics, the logic of programming, and the creativity required to solve complex problems. This work documents not only the technical aspects of the project but also the learnings, challenges, and solutions encountered during its construction, aiming to create a functional and efficient tool for syntactic and semantic analysis.
The project began with an ambitious goal: to transform a set of abstract grammatical rules into an operational system capable of processing and validating code, translating it into intermediate structures ready for compilation. This challenge involved the careful selection of strategies such as eliminating left recursion, handling grammatical ambiguities, and implementing parsing methods like LL(1) and LR(1), each with its own advantages and limitations.
Beyond the technical aspects, this project represents a bridge between theory and practice. Context-Free Grammars (CFGs), *FIRST* and *FOLLOW* functions, and techniques to resolve shift/reduce conflicts transitioned from abstract concepts to practical tools that brought the parser to life. The integration of the SDT expanded the system's capabilities, enabling not only syntax validation but also adding meaning to each grammatical production.
The objective of this work is not just to describe how the system was built but also to inspire readers to understand the importance of parsers and SDTs in the programming world. These components are not mere technical mechanisms; they are the key to transforming human ideas into instructions that machines can understand. By reading this document, the audience is expected to appreciate not only the challenges and solutions addressed but also the relevance of this project in compiler design and its impact on creating robust and efficient software.
This work invites the reader to embark on a journey through the fundamentals of programming language construction, where logic, creativity, and precision converge to solve one of the most fascinating problems in software development: language analysis. Are you ready to uncover the secrets behind computational language?

2 Theoretical Framework

2.1 Parsing

Parsing refers to the process of analyzing a sequence of tokens to determine its grammatical structure based on a predefined grammar. It involves constructing a parse tree or an abstract syntax tree that represents the hierarchical organization of the code.

2.1.1 Key Concepts in CFGs

Classification of CFGs: Context-Free Grammars (CFGs) can be classified based on various properties. Below are visual representations of these classifications:

Ambiguous vs. Unambiguous CFGs

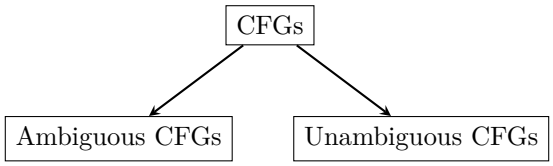

Figure 1: Classification of CFGs based on ambiguity.

Ambiguity Example: Consider the grammar:

```
E → E + E | E * E | id
```

The string `id + id * id` can have multiple parse trees, indicating ambiguity.

Left Recursive vs. Right Recursive CFGs

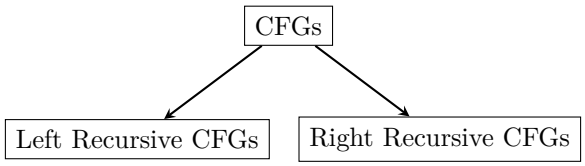

Figure 2: Classification of CFGs based on recursion.

Eliminating Left Recursion: Left recursion (e.g., $E \to E + T$) can lead to infinite loops in parsers. It can be eliminated as follows:

```
E → T E'
E' → + T E' | \epsilon
```

Deterministic vs. Non-Deterministic CFGs

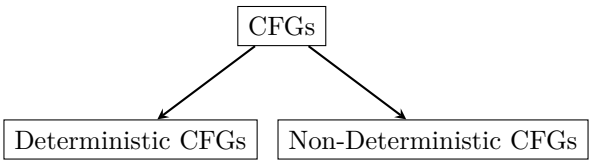

Figure 3: Classification of CFGs based on determinism.

Summary of CFG Classifications: These classifications of CFGs—based on ambiguity, recursion, and determinism—help in designing grammars that are compatible with different types of parsers and avoid common pitfalls in parsing.

2.1.2 Types of Parsers

Top-Down Parsers:

- **Recursive Descent:** Uses recursive functions to process production rules.
- **LL(1):** Utilizes predictive tables based on `FIRST` and `FOLLOW` functions to decide the next step.

Bottom-Up Parsers:

- Builds the parse tree from tokens to the start symbol.
- **Types:**
  - Operator Precedence Parsers: Handle operator precedence.
  - LR Parsers (`LR(0), SLR(1), CLR(1), LALR(1)`): Resolve conflicts efficiently using canonical item collections.

2.2 Semantic Analysis

Definition: Semantic analysis validates the language rules that cannot be checked during syntax analysis, such as type consistency, identifier usage, and operation coherence. It ensures that the code has valid meaning before execution or compilation.

2.2.1 Key Elements

Syntax-Directed Translation (SDT):

- **Definition:** A technique that associates semantic rules with grammar productions to evaluate or transform input programs.
- **Attributes:**
  - Synthesized: Computed using values from child nodes of the parse tree.
  - Inherited: Derived from parent or sibling nodes.
- **Evaluation Strategies:**
  - **Postorder Evaluation:** Attributes are computed after visiting all child nodes.
  - **One-Pass Evaluation:** Attributes are computed during parsing for efficiency.
- **Example:**

```
Production: E → E1 + T { E.val = E1.val + T.val }
Example Input: 4 + 2
Evaluation:
E1.val = 4
T.val = 2
E.val = 4 + 2 = 6
```

- **Practical Use Case:**

```
Grammar Production:
D → T id { addSymbol(id.name, T.type); }
Example Input: int x;
Semantic Action:
- Adds "x" to the symbol table with type "int".
```

Abstract Syntax Tree (AST): A simplified representation that removes redundant syntactic details to focus on the core structure of the program.

Type Checking: Ensures that operations between types are valid.

- **Example:**

```
int x = "hello"; // Error: type mismatch
```

Scope and Symbol Management:

- Symbol Tables: Associate identifiers with their information (type, value, etc.).
- **Scope Checking:** Verifies whether an identifier is correctly defined within its context.

Intermediate Code Generation: Translates the source program into a representation closer to machine code.

- **Example:**

```
x = y + z;
```

Intermediate Code (Three-Address Code):

```
t1 = y + z
x = t1
```

Advantages of SDT:

- Modular and structured definition of semantic rules.
- Efficient attribute evaluation during parsing.
- Enables the generation of Abstract Syntax Trees (ASTs) and intermediate representations.

Applications of SDT:

- **Type Checking:** Validates type consistency in expressions and assignments.
- **Scope Management:** Ensures identifiers are declared and used within valid scopes.
- **Intermediate Code Generation:** Facilitates translation to low-level representations like three-address code.

Parsing establishes the syntactic structure, while Semantic Analysis adds meaning to that structure. Together, they ensure that syntactically correct programs adhere to language rules, allowing reliable execution or compilation

3 Development

For the implementation of the syntax analyzer, it is essential to define three key elements: grammar, tokens, and the parsing method. Below, we provide a detailed explanation of each.

3.1 Grammar

In our case, we will be using **context-free grammars (CFGs)**. CFGs are a type of grammar widely used for defining the syntax of programming languages and are particularly suited for syntactic analysis. They consist of the following components:

- NT: A set of non-terminal symbols, which represent syntactic categories or abstractions.
- T: A set of terminal symbols, representing the basic symbols or tokens of the language.
- P: A set of production rules, where each rule specifies how a non-terminal can be expanded into a sequence of terminals and/or non-terminals.
- S: A designated start symbol, which is a special non-terminal that represents the entire structure of the language.

The formal definition of a CFG is represented as a tuple:

$$G = (NT, T, P, S)$$

The syntax analyzer for this project will utilize specific sets of non-terminal and terminal symbols, which are described below.

3.1.1 Context-Free Grammar (CFG)

The syntax analyzer is based on a Context-Free Grammar (CFG), which consists of the following elements:

- **Non-Terminals (NT):**

```
NT = { S', PROGRAM, FUNCTION, BLOCK, DECLARATIONS,
       DECLARATION, STATEMENTS, STATEMENT,
       ASSIGNMENT_STATEMENT, ASSIGNMENT_OPERATION,
       OPERATION, RETURN_STATEMENT }
```

- **Terminals (T):**

```
T = { include, keyword, identifier, punctuation,
      operator, literal, number, plus, minus,
      times, divide, modulus, exponent }
```

- **Production Rules (P):**

```
1. S' -> PROGRAM
2. PROGRAM -> include FUNCTION
3. FUNCTION -> keyword identifier punctuation punctuation BLOCK
4. BLOCK -> punctuation DECLARATIONS STATEMENTS RETURN_STATEMENT
            punctuation
5. DECLARATIONS -> DECLARATION punctuation
                 | DECLARATIONS DECLARATION punctuation
6. DECLARATION -> keyword identifier
7. STATEMENTS -> STATEMENT punctuation
               | STATEMENTS STATEMENT punctuation
8. STATEMENT -> ASSIGNMENT_STATEMENT
              | ASSIGNMENT_OPERATION
              | identifier punctuation literal punctuation
9. ASSIGNMENT_STATEMENT -> identifier operator number
                         | identifier operator literal
10. ASSIGNMENT_OPERATION -> identifier operator OPERATION
11. OPERATION -> number plus number
               | number minus number
               | number times number
               | number divide number
               | number modulus number
               | number exponent number
12. RETURN_STATEMENT -> keyword number punctuation
```

- **Initial Symbol (S):**

$$S = S'$$

3.2 Tokens

Tokens are the smallest meaningful units of a program, serving as the building blocks for lexical analysis during compilation. The lexer scans the source code and categorizes each character sequence into a specific token type. Using tokens ensures that the syntax analyzer can focus on higher-level structures without worrying about the detailed syntax of individual components. For this project, we define the following tokens as sets:

3.2.1 Defined Tokens

```
# KEYWORD
keywords = {int, return, char, float, double, void}

# PUNCTUATION
punctuation = {;, {, }, (, )}

# OPERATOR
operator = {=, +, -, *, /, %, ^}

# PLUS
plus = {+}

# MINUS
minus = {-}
```

```
# TIMES
times = {*}

# DIVIDE
divide = {/}

# MODULUS
modulus = {%}

# EXPONENT
exponent = {^}

# INCLUDE
include = {#include<stdio.h>}

# LITERAL
literals = {"string", 'character'}

# IDENTIFIER
identifier = {variable_name, function_name, ...}

# numbers
numbers = {0, 1, 2, ..., infinite +}
```

The sets above are designed to efficiently capture and categorize all elements of the source code, ensuring that both the lexer and parser can work together seamlessly.

3.3 FIRST and FOLLOW Functions

The FIRST and FOLLOW functions are crucial for constructing parsing tables in LALR(1) parsers. These functions determine how the parser predicts which grammar rules to apply during parsing.

- **FIRST:** The FIRST function of a non-terminal is the set of terminal symbols that can appear at the beginning of a derivation from that non-terminal.
- **FOLLOW:** The FOLLOW function of a non-terminal is the set of terminal symbols that can appear immediately to the right of that non-terminal in some derivation.

The following table summarizes the FIRST and FOLLOW sets for each non-terminal in the given grammar:

Non-Terminal	**FIRST**	**FOLLOW**
S'	{include}	{$}
$PROGRAM$	{include}	{$}
$FUNCTION$	{keyword}	{$}
$BLOCK$	{punctuation}	{$}
$DECLARATIONS$	{keyword}	{identifier, keyword}
$DECLARATION$	{keyword}	{punctuation}
$STATEMENTS$	{identifier}	{identifier, punctuation}
$STATEMENT$	{identifier}	{punctuation}
$ASSIGNMENT\_STATEMENT$	{identifier}	{punctuation}
$ASSIGNMENT\_OPERATION$	{identifier}	{punctuation}
$OPERATION$	{number}	{punctuation}
$RETURN\_STATEMENT$	{keyword}	{punctuation}

Table 1: FIRST and FOLLOW Sets for the Grammar

3.4 Canonical Collection of LR(1) Items

The canonical collection of LR(1) items is a set of states used to construct the parsing table in an LALR(1) parser. Each state represents a unique set of LR(1) items, which include the parser's current position within grammar rules and a lookahead token to guide parsing decisions. These items ensure efficient and accurate parsing by predicting the next action (shift, reduce, or accept) based on both the current state and the lookahead symbol.

- **Significance:** The canonical collection helps construct the parsing table by defining all possible states and transitions of the parser, resolving ambiguities in grammar, and ensuring correctness during parsing.
- **F**or this project, we will use the canonical collection of LR(1) items as shown in the image below.

3.4.1 Program Structure and Initial States

- **Initial States:**
  - **State 0:** Starts the parsing process, waiting for the production $S' \rightarrow PROGRAM$.
  - **State 1-3:** Handles the initial structure of the program, including the 'include' directive and preparing for function definitions.
- **Function Definition:**
  - **State 4-9:** Processes function declarations, including keywords, identifiers, parentheses, and setup for the function block.

3.4.2 Declaration States

- **Declaration Handling:**
  - **State 10:** Entry point for declarations.
  - **State 11-20:** Manages individual and multiple declarations, keywords, identifiers, and assignments, ensuring proper punctuation and completion.
  - **State 28:** Handles the entire declaration block.

3.4.3 Statement States

- **Statement Structure:**
  - **State 21:** Handles the block of statements.
  - **State 22-24:** Processes individual statements and ensures punctuation.
- **Types of Statements:**
  - **State 25-29:** Processes punctuation, identifiers, literals, and return statements.
  - **State 35-36:** Finalizes return statements and completes literals.

3.4.4 Operation States

- **Building Operations:**
  - **State 27:** Handles operators.
  - **State 30-34:** Processes numbers, literals, and more complex operations, ensuring correct precedence and grouping.
- **Operators:**
  - **State 37-42:** Processes arithmetic operators: addition, subtraction, multiplication, division, modulus, and exponentiation.
  - **State 43-48:** Completes operations for each operator type.

3.4.5 Special Features and Observations

- **Expression Handling:**
  - Supports arithmetic expressions with operator precedence and parentheses for grouping.
- **Program Structure:**
  - Allows multiple declarations, well-defined code blocks, and mandatory return statements.
- **Type System:**

  - Supports typed identifiers, numbers, and literals with type validation in operations.

- **Control Flow:**
  - Maintains a hierarchical structure with clear handling of nested blocks and punctuation for delimiting structures.

- **Design Principles:**
  - The grammar is deterministic, with each state serving a specific purpose.
  - Transitions between states are consistent, resolving conflicts like shift/reduce effectively.

3.4.6 Implementation Notes

The organization of states ensures a logical and progressive flow:

- Symbols are handled coherently, and transitions are clear and unambiguous.
- The grammar avoids ambiguities, facilitating extensions for additional features.
- Parsing decisions are deterministic, ensuring efficient and reliable operation.

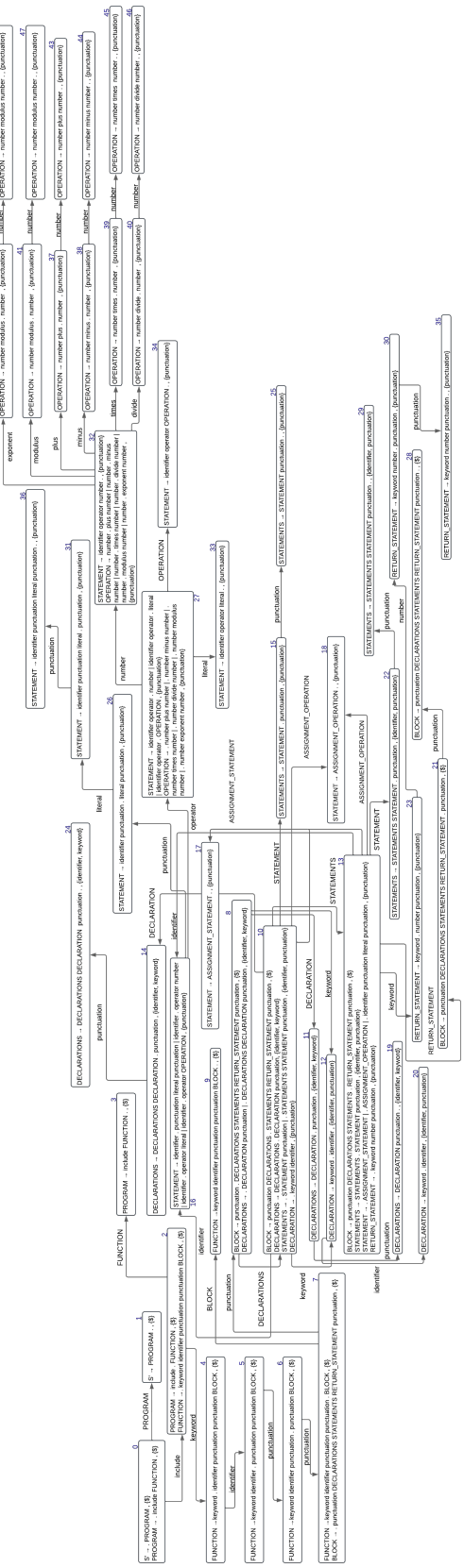

Figure 4: Canonical Collection of LR(1) Items for the Project Grammar

3.5 Parsing Table

The parsing table is used to guide the LALR(1) parser in making decisions about shifting, reducing, or accepting input. It is generated from the canonical collection of LR(1) items and is essential for deterministic and efficient parsing. The table ensures that the parser can handle grammar constructs accurately and resolve conflicts effectively.

3.5.1 Parsing Table Description

The parsing table divides its states into different columns that represent the possible terminal and non-terminal symbols that may be encountered. These include shift actions, reduce actions, accept actions, and the states to which the parser transitions.

Each entry in the table shows the action to take:

- "S" followed by a number represents a shift action to a state.
- "r" followed by a number indicates a reduction using a specific grammar rule.
- "acc" represents acceptance of the input, indicating the successful end of parsing.

Part 1: Transitions for Terminal Symbols This first section of the table contains the actions for terminal symbols, which include `$end`, `DIVIDE`, `EXPONENT`, `INCLUDE`, `KEYWORD`, `LITERAL`, `MINUS`, `MODULUS`, `NUMBER`, `OPERATOR`, `PLUS`, and `PUNCTUATION`.

Key State Transition Examples:

- **State 0:**
  - `INCLUDE` triggers a shift (`S2`), allowing the production `program -> INCLUDE function` to start.
- **State 1:**
  - This state shows `acc` (accept) when encountering `$end`, indicating the successful end of parsing.
- **State 2:**
  - Here, `KEYWORD` triggers `S4`, leading to the next step in the definition of the function.

These transitions ensure that upon encountering certain symbols, the parser knows the next state or whether to apply a reduction to resolve a specific production.

Part 2: Actions and States with Specific Terminal and Non-Terminal Symbols This section includes both terminal symbols (`PUNCTUATION`, `TIMES`, etc.) and non-terminal symbols (`assignment_operation`, `assignment_statement`, `block`, etc.).

Notable Transition Details:

- **State 5:**
  - PUNCTUATION triggers S6, a step in the production `function -> KEYWORD IDENTIFIER PUNCTUATION PUNCTUATION block`.
- **State 17:**
  - Here, r15 (reduction) is applied using the rule `statement -> assignment_statement`. This indicates that a complete assignment statement is considered a `statement`.
- **State 21:**
  - In this state, `return_statement` and other non-terminal symbols such as `statement` are managed, allowing the parser to progress through code blocks and statements.

These states and their transitions play a fundamental role in processing code blocks, handling both statements and assignments.

Part 3: States and Complex Productions This section shows the relationship between states and major non-terminal productions such as `program`, `function`, `statements`, and `operation`.

Key Transition Examples in States:

- **State 0:**
  - Allows the initial transition to `program`, indicating the start of the main production.
- **State 10:**
  - In this state, both `statements` and `declarations` are evaluated. The state allows the construction of complete code blocks.
- **State 16:**
  - IDENTIFIER can trigger PUNCTUATION and OPERATOR, triggering productions of `assignment_statement` or `assignment_operation`, facilitating handling of operations and assignments within the code.

These states allow the parser to proceed to additional productions within the main function grammar.

Table 2: Parsing Table for the parser (ACTIONS)

state	$	divide	exponent	identifier	include	keyword	literal	minus	modulus	number	operator	plus	punctuation	times
0					S2									
1	acc													
2						S4								
3	R1													
4				S5										
5													S6	
6													S7	
7													S8	
8						S12								
9	R2													
10				S16		S12								
11													S19	
12				S20										
13				S16		S23								
14													S24	
15													S25	
16											S27	S26		
17													R15	
18													R17	
19				R4		R4								
20													R6	
21													S28	
22													S29	
23										S30				
24				R5		R5								
25				R7		R7								
26							S31							
27							S33			S32				
28	R3													
29				R8		R8								
30													S35	
31													S36	
32		S40	S42					S38	S41			S37	R18	S39
33													R19	
34													R20	
35													R21	
36													R16	
37										S43				
38										S44				
39										S45				
40										S46				

Continued on next page

Table 2: Parsing Table for the Project Grammar (ACTIONS - continued)

state	$	divide	exponent	identifier	include	keyword	literal	minus	modulus	number	operator	plus	punctuation	times
41										S47				
42										S48				
43													R9	
44													R10	
45													R11	
46													R12	
47													R13	
48													R14	

Table 3: Parsing Table for the parser (GOTO)

state	ASSIGN_OP	ASSIG_STATE	BLOCK	DECLA	DECLARS	FUNCT	OPER	PROGRAM	RET_STATE	STATE	STATES
0								1			
1											
2						3					
3											
4											
5											
6											
7			9								
8				11	10						
9											
10	18	17			14					15	13
11											21
12			22								
13	18	17							21	22	
14											
15											
16											
17											
18											
19											
20											
21											
22											
23											
24											
25											
26											
27							34				
28											
29											
30											
31											
32											
33											
34											
35											
36											
37											
38											
39											
40											

Continued on next page

Table 3: Parsing Table for the Project Grammar (GOTO - continued)

state	ASSIGN_OP	ASSIG_STATE	BLOCK	DECLA	DECLARS	FUNCT	OPER	PROGRAM	RET_STATE	STATE	STATES
41											
42											
43											
44											
45											
46											
47											
48											

3.6 Case Study

In this case study, we analyze the input string {`int var1; var1 = 4; return 0;`} using the implemented parser. The following table illustrates the step-by-step parsing process:

Table 4: Parsing Process for the Input String {`int var1; var1 = 4; return 0;`}

STACK	PARSE SYMBOLS	READ SYMBOL	ACTION
7		{	shift to 8
7 8	{	int	shift to 12
7 8 12	{ int	var1	shift to 20
7 8 12 20	{ int var1	DECLARATION	reduce by DECLARATION
7 8 11	{ DECLARATION	;	shift to 19
7 8 11 19	{ DECLARATION ;	DECLARATIONS	reduce by DECLARATIONS
7 8 10	{ DECLARATIONS	var1	shift to 16
7 8 10 16	{ DECLARATIONS var1	=	shift to 27
7 8 10 16 27	{ DECLARATIONS var1 =	4	shift to 32
7 8 10 16 27 32	{ DECLARATIONS var1 = 4	STATEMENT	reduce by STATEMENT
7 8 10 15	{ DECLARATIONS STATEMENT	;	shift to 25
7 8 10 15 25	{ DECLARATIONS STATEMENT ;	STATEMENTS	reduce by STATEMENTS
7 8 10 13	{ DECLARATIONS STATEMENTS	return	shift to 23
7 8 10 13 23	{ DECLARATIONS STATEMENTS return	0	shift to 30
7 8 10 13 23 30	{ DECLARATIONS STATEMENTS return 0	;	shift to 35
7 8 10 13 23 30 35	{ DECLARATIONS STATEMENTS return 0 ;	RETURN_...	reduce by RETURN_...
7 8 10 13 21	{ DECLARATIONS STATEMENTS RETURN_...	}	shift to 28
7 8 10 13 21 28	{ DECLARATIONS STATEMENTS RETURN_... }	BLOCK	reduce by BLOCK
7 9	BLOCK		

3.6.1 Analysis and Interpretation

The parser processes the input string by applying the following actions:

- **Shifts:** Represent movements to a new state upon recognizing an input symbol. For instance, in step 1, upon reading the symbol {, the parser performs a shift to state 8.
- **Reduces:** Apply a grammar rule by replacing a sequence of symbols with a non-terminal. For example, in step 4, the sequence `int var1` is reduced to `DECLARATION`.

The analysis correctly follows the grammar structure:

- The variable declaration (`int var1;`) is processed and reduced to `DECLARATION`.
- The assignment (`var1 = 4;`) is interpreted as a `STATEMENT`.
- The return statement (`return 0;`) is reduced to `RETURN_STATEMENT`.

The parser concludes with the non-terminal `BLOCK`, indicating that the entire input is valid according to the grammar. The use of `shift` and `reduce` in the parsing table ensures syntactic consistency.

This case study demonstrates how the implemented LALR(1) parser handles block structures, declarations, and statements, ensuring both syntactic and semantic correctness.

3.7 Implementation of the Parser

The parser for this project was built using Python's PLY (Python Lex-Yacc) library, a powerful tool for constructing lexers and parsers. PLY is designed to follow the traditional structure of Lex and Yacc but is implemented entirely in Python, providing a straightforward interface for defining tokens, grammar rules, and parsing behavior.

One of PLY's key features is its use of LALR(1) parsing, a deterministic parsing technique that efficiently resolves grammar rules by combining Look-Ahead and Left-to-Right parsing strategies. This makes it well-suited for handling complex grammars while maintaining performance.

3.7.1 Defining Tokens and Grammar Rules

In PLY, tokens are defined using regular expressions, while grammar rules are implemented as Python functions. The lexer generates tokens from input strings, which the parser processes according to the defined grammar. For example:

```
tokens = ['NUMBER', 'PLUS']

t_PLUS = r'\+'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

This defines tokens for numbers and the plus operator, allowing the lexer to recognize arithmetic expressions.

3.7.2 Defining Grammar Rules

Grammar rules in PLY are represented as Python functions, with each rule corresponding to a specific production. The parser applies these rules to process the input tokens and construct a valid parse tree or perform semantic analysis. Here's an example of a grammar rule for a single declaration:

```
# Grammar rule for a single declaration
def p_declaration(p):
    '''declaration : KEYWORD IDENTIFIER'''
```

Semantic Error Handling This rule matches a type keyword (e.g., `int`, `float`) followed by an identifier. It checks for semantic errors, such as redeclaration of variables, and updates the symbol table accordingly. Semantic error handling ensures that the input adheres to contextual rules of the language. For example, the following grammar rule checks for variable redeclaration:

```
def p_declaration(p):
    '''declaration : KEYWORD IDENTIFIER'''
    var_type = p[1]
    var_name = p[2]
    if var_name in symbol_table:
        print(f"Semantic error: the variable '{var_name}' is
        already declared.")
        global flag_semantic_error
        flag_semantic_error = True
    else:
        symbol_table[var_name] = var_type
```

This rule processes a variable declaration (‘KEYWORD IDENTIFIER‘), extracts its type and name, and checks if the variable already exists in the `symbol_table`. If redeclared, an error message is printed, and a global error flag is set.

Semantic error handling ensures:

- Variables are declared uniquely within the same scope.
- Clear error messages are provided for redeclaration issues.
- The parser halts or takes corrective actions when the global error flag is set.

Syntax Error Handling Syntax error handling ensures that invalid input is identified and reported during parsing. The following function defines error handling in the parser:

```
def p_error(p):
    global flag_sintax_error, flag_semantic_error
    flag_sintax_error = True
    flag_semantic_error = True
    if p:
        print(f"Syntax error in '{p.value}'")
    else:
        print("Syntax error at the end of the file")
```

This function sets global error flags for syntax and semantic errors. It distinguishes between:

- Errors in specific tokens, reporting the token value causing the issue.
- Errors at the end of the input, indicating incomplete input or unexpected termination.

This mechanism allows the parser to halt gracefully and provide meaningful feedback to the developer, aiding in debugging and ensuring program correctness.

3.8 Parsing Workflow

PLY integrates the lexer and parser seamlessly:

- The lexer converts the input into a stream of tokens.
- The parser processes these tokens using LALR(1) logic, applying grammar rules like the one above to perform syntax and semantic analysis.

PLY's API provides clear error messages and debugging tools, making it easier to identify and resolve issues in the grammar or parsing logic.

This approach ensures the implementation is both efficient and extensible, adhering to the project's grammar and parsing requirements.

4 Results

4.1 Syntax-Directed Translation (SDT)

The Syntax-Directed Translation (SDT) defines semantic rules associated with grammar productions. These rules enable the parser to perform specific semantic actions during the parsing process, ensuring proper interpretation of the input. Below are the SDT rules generated from our grammar:

```
S' → PROGRAM {printf("$");}
PROGRAM → include FUNCTION {printf("P");}
FUNCTION → keyword identifier punctuation BLOCK {printf("F");}
BLOCK → punctuation DECLARATIONS STATEMENTS RETURN_STATEMENT punctuation
        {printf("B");}
DECLARATIONS → DECLARATION punctuation | DECLARATIONS DECLARATION
               punctuation {printf("DS");}
DECLARATION → keyword identifier {printf("D");}
STATEMENTS → STATEMENT punctuation | STATEMENTS STATEMENT punctuation
{printf("ES");}
STATEMENT → ASSIGNMENT_STATEMENT
          | ASSIGNMENT_OPERATION
          | identifier punctuation literal punctuation {printf("E");}
ASSIGNMENT_STATEMENT → identifier operator number
                     | identifier operator literal {printf("AS");}
ASSIGNMENT_OPERATION → identifier operator OPERATION {printf("AO");}
OPERATION → number plus number
          | number minus number
          | number times number
          | number divide number
          | number modulus number
          | number exponent number {printf("OP");}
RETURN_STATEMENT → keyword number punctuation {printf("R");}
```

4.1.1 Input Example and SDT Execution

For the input string:

```
#include<stdio.h> int main ( ) {int var1; var1 = 2+3; return 0;}
```

The parser applies the semantic actions during parsing. The following sequence of semantic actions is generated:

```
OUTPUT : DOPRDSAOEESBFPS
```

4.1.2 Interpretation of the SDT Output

The output represents the sequence of semantic actions executed during parsing:

- `D`: Variable declaration.
- `OP`: Arithmetic operation.
- `R`: Return statement.
- `DS`: Multiple declarations.
- `AO`: Assignment operation.
- `E`: Processing a statement.
- `ES`: Multiple statements.
- `B`: Block structure processed.
- `F`: Function defined.
- `P`: Program structure completed.
- `$`: End of parsing.

This output demonstrates how the SDT rules translate the input string into a meaningful sequence of semantic actions, ensuring that the program structure adheres to both syntactic and semantic correctness.

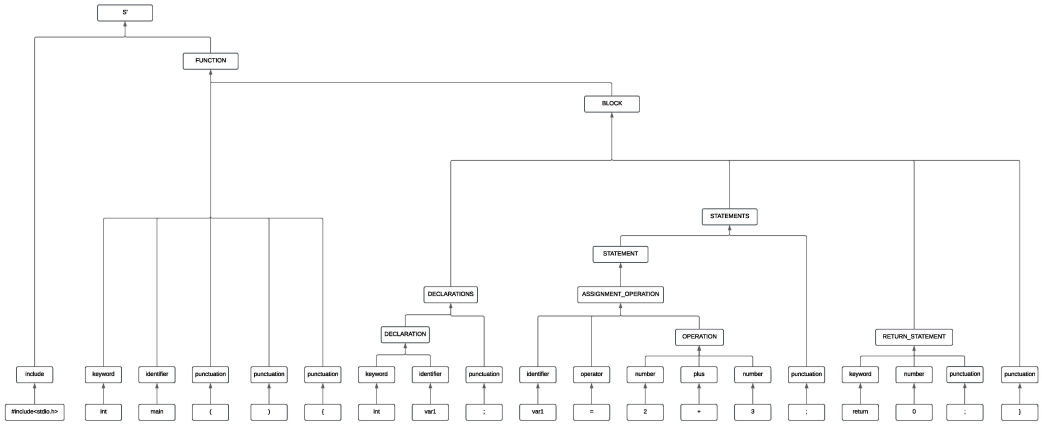

Figure 5: SDT for input string

4.2 Error Analysis in Executions

The following details the errors found in each execution, along with an explanation of the types of errors present.

Execution 1:

```
#include <stdio.h>
int main() {
    int b;
    t = 5+3;
    b = 3+2;
    return 0;
}
```

- **Execution result:**
  - Semantic error: The variable `t` has not been declared.
  - Syntactic analysis successful.
  - SDT error.
- **Explanation:**
  - **Semantic error**: This error occurs because the variable `t` is used without being previously declared. During semantic analysis, the compiler verifies that all variables used are declared; if any is not, a semantic error is generated.
  - **SDT error**: The SDT (Syntax-Directed Translation) error could be due to the semantic error, preventing the correct translation of the code.

```
PS C:\Users\axell\desktop\compiladores\python_parser> python parser.py test1.c
Syntactic analysis result:
Semantic error: the variable 't' has not been declared yet.
Parsing success
SDT error
```

Figure 6: Execution 1 Result

Execution 2:

```
#include <stdio.h>
int main() {
    int a;
    int b;
    printf("Hello, world!");
    a = 5^2;
    b = 3+2;
    return 0;
}
```

- **Execution result:**
  - Syntactic analysis successful.
  - SDT verified.
- **Explanation:**
  - **No errors**: Both syntactic and semantic analysis were successful. Although the ^ operator in `a = 5^2;` may not behave as expected (as it is a bitwise operator in C), it does not generate a semantic error.
  - **SDT verified**: This indicates that the translation process was carried out correctly, as there are no errors in the code.

```
PS C:\Users\axell\desktop\compiladores\python_parser> python parser.py main.c
Syntactic analysis result:
Parsing success
SDT Verified
```

Figure 7: Execution 2 Result

Execution 3:

```
#include <stdio.h>
int main() {
    int a;
    int b;
    c int;
    printf("Hello, world!");
    a = 5^2;
    b = 3+2;
    return 0;
}
```

- **Execution result:**
  - Syntax error: Syntax error in `int`.
  - Parsing error.
  - SDT error.
- **Explanation:**
  - **Syntax error**: The error is in `c int;`, which is not a valid declaration in C. This incorrect syntax causes a parsing error, as the compiler expects variable declarations to follow a specific format.
  - **SDT error**: Due to the syntax error, SDT could not be completed correctly.

```
PS C:\Users\axell\desktop\compiladores\python_parser> python parser.py test2.c
Syntactic analysis result:
Syntax error in 'int'
Parsing error
SDT error
```

Figure 8: Execution 3 Result

4.2.1 Summary of Error Types

- **Syntax errors**: These occur when the code does not follow the grammatical rules of the language. Such an error is seen in Execution 3 due to an incorrect declaration.
- **Semantic errors**: These arise when the code is syntactically correct but uses variables or functions incorrectly (as in Execution 1, where `t` is undeclared).
- **Both errors**: Both types of errors can occur simultaneously if the code has structural (syntactic) and logical (semantic) issues.
- **No errors**: When the code is correct in both syntax and semantics, no errors occur, as in Execution 2.

5 Conclusions

This project focused on the design and implementation of a syntax and semantic analyzer, integrating theoretical concepts such as context-free grammars (CFGs), FIRST and FOLLOW functions, and canonical collections of LR(1) items. It was an experience that combined mathematical precision, programming logic, and creativity to solve complex challenges. The project translated abstract grammar rules into a functional system capable of validating syntax and performing Syntax-Directed Translations (SDT). Implementation using Python and the PLY library enabled an efficient workflow, supporting complex grammars with deterministic LALR(1) analysis. Representative examples validated the analyzer's functionality, including syntax and semantic error detection, demonstrating system robustness. Challenges included resolving shift/reduce conflicts in ambiguous grammars through strategies such as eliminating left recursion, addressing semantic issues like undeclared variables and incorrect type usage with robust symbol table management, and validating less common operators like exponentiation ($\hat{()}$) for proper implementation. Highlights of the project included developing rules to translate syntactic inputs into semantic actions, creating Abstract Syntax Trees (ASTs) and intermediate code for subsequent compilation processes, and transitioning theoretical concepts such as LR(1) canonical collections into practical tools for constructing parsing tables. This project not only reinforced fundamental knowledge in programming language design but also demonstrated the relevance of analyzers in creating robust software. Its implementation highlighted how well-designed tools can transform abstract ideas into concrete, functional solutions.

References

[1] S. S. S. P. Rao, K. S. R. Anjaneyulu, and K. S. R. Anjaneyulu, "English Algorithm Translation to C Program using Syntax Directed Translation," in *2022 6th International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 1-5, 2022. [Online]. Available: `https://ieeexplore.ieee.org/document/9996044`. [Accessed: 14-Nov-2024].

[2] J. Boyland and G. Hedin, "Implementing Attribute Grammars Using Conventional Compiler-Compiler Tools," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. 1-10, 1992. [Online]. Available: `https://ieeexplore.ieee.org/document/6078313`. [Accessed: 14-Nov-2024].

[3] J. E. Hopcroft and J. D. Ullman, "Syntax-Directed Transduction," in *1969 Spring Joint Computer Conference*, pp. 1-15, 1969. [Online]. Available: `https://ieeexplore.ieee.org/document/4569518`. [Accessed: 14-Nov-2024].