

NATIONAL AUTONOMOUS UNIVERSITY OF
MEXICO

FACULTY OF ENGINEERING



Computer Engineering

Compilers

Final Project: Interpreter

Professor: Eng. Rene Adrian Dávila Perez

Semester: 2025-1

Team Members 07:

319050390

319103795

319024045

319240731

319321935

Group: 5

México, CDMX. November 25, 2024

Contents

1	Introduction	3
2	Objective	3
3	Theoretical Framework	3
4	Development	7
4.1	Grammar	7
4.1.1	Context-Free Grammar (CFG)	8
4.2	Tokens	8
4.2.1	Defined Tokens	8
4.3	FIRST() and FOLLOW()	9
4.4	Canonical Collection	10
4.5	Parsing Table	13
4.5.1	Parsing Table Description	13
5	Results	15
5.1	Case Study	15
5.1.1	Grammar and Semantic Rules	15
5.1.2	Step-by-Step Execution	16
5.1.3	Syntax Directed Translation (SDT)	17
5.2	Empire	20
5.2.1	Postfix	22
5.2.2	Case Study	24
6	Conclusions	25

1 Introduction

Throughout the course, the theoretical and practical fundamentals of compiler design have been studied, covering each crucial stage for its development. As a final product of this learning, the development of an interpreter was carried out, a program whose main function is to analyze, process and execute source code in real time, without the need to previously convert it to a binary executable. Unlike a compiler, which translates the complete code into machine language before its execution, an interpreter performs this task line by line, which makes it ideal for interactive environments.

The importance of interpreters lies in their ability to provide immediate feedback during code execution, which makes them a very useful tool in programming, which is why we chose to implement one in Python as a final project. Its development was based on the integration of the previous stages we worked on in the previous deliveries, this with the necessary adjustments, further development and more research.

First, we have the lexical analyzer (Lexer) which is responsible for scanning the source code and classifying it into tokens. Subsequently, the syntactic analyzer (Parser) validated the grammatical structure of the code, ensuring that it complied with the rules of the language and also generated a parse tree. Likewise, the use of Syntax Driven Translation (SDT) techniques allowed verifying the semantic rules and facilitating the transformation of the code into executable instructions, also with more elements that we will continue to see in the development of this work.

In the following work we will explain how was its creation process, organization and we will detail all the elements that we used, its operation and we will also show the results obtained from this practical implementation.

2 Objective

1. Apply all the knowledge acquired throughout the course to implement a functional Interpreter as a final project.
2. Combine the previous stages (Lexer, Parser & SDT) so that the interpreter is able to receive and execute code, translating it into machine code.

3 Theoretical Framework

In this project, we develop the final stage of our project: an interpreter. The interpreter processes source code during runtime, acting as an interface between the code and the hardware where it is executed. To achieve this, previous analyses performed by the lexer and parser are necessary.

The lexical analysis is the first phase in creating the interpreter. It classifies substrings of the input code into a set of tokens defined by regular expressions. The main objective is to generate this set to identify the different elements of the code, ignoring whitespaces

and comments, and to notify the user if any elements or substrings do not match any established category, as this could affect the program's functionality.

For this interpreter, we define the following token categories:

- **ID**: Identifiers, which describe variable or function names determined by the programmer.
- **NUM**: Identifies integers and decimals.
- **COMPLEX**: Identifies complex numbers in polar form.
- **PLUS, MINUS, TIMES, DIVIDE, SQRT**: Symbols for addition, subtraction, multiplication, division, and square root.
- **LPAREN, RPAREN**: Left and right parentheses.
- **ASSIGN**: Symbol = used in assignments.
- **PRINT**: Identifies instructions that produce output on the device.

Key Concepts of Lexical Analysis:

- **Lexemes**: Sequences of characters in the source code that match a token pattern identified by the lexer.
- **Tokenization**: The process of classifying lexemes into tokens based on predefined rules.
- **Regular Expressions**: Patterns that define the structure of token lexemes. For example, keywords match a sequence of characters, whereas identifiers match more complex patterns.
- **Finite Automata**: Mathematical models used to process text in a lexer.
 - **Deterministic Finite Automata (DFA)**: Each state and input combination leads to a single next state, making it predictable and easy to implement.
 - **Nondeterministic Finite Automata (NFA)**: Allows transitions to multiple states, offering flexibility but increasing complexity. Every NFA can be converted into a DFA.
 - **Epsilon-NFA (ϵ -NFA)**: Includes transitions that occur without consuming input, increasing flexibility for pattern recognition.
- **Symbol Table**: A data structure that holds metadata about tokens, such as their type and position.
- **Token**: A pair consisting of a token name and optional attribute value. The token name represents a lexical unit type, such as keywords or identifiers.

After lexical analysis, assuming no lexical errors are present, syntax and semantic analyses are performed. These verify the sequence of tokens and ensure program logic complies with the defined grammar. This process constructs a *parsing tree* to identify ambiguities and avoid recursion.

For this interpreter, an **SLR(1) parser** is used. This *bottom-up parser* builds the parse tree from tokens up to the start symbol.

The next stage, semantic analysis, ensures that programming language rules are followed.

Key Elements of Semantic Analysis:

- **Syntax-Directed Translation (SDT):** Associates semantic rules with grammar productions.
- **Abstract Syntax Tree (AST):** A simplified structure focusing on core program elements, removing redundant syntax.
- **Intermediate Code Generation:** Converts source code into representations closer to machine code.

The intermediate code generation phase translates source code into a more abstract form, preparing it for optimization and target code generation. Common representations include:

- **Linear Form:**
 - **Postfix Notation (RPN):** Uses a stack-based approach to respect operator precedence.
 - **Three-Address Code (TAC):** Divides expressions into sub-expressions using temporary variables and registers.
 - * **Quadruples:** Four fields per instruction: operator, argument1, argument2, and result.
 - * **Triples:** Three fields per instruction: operator, argument1, and argument2.
 - * **Indirect Triples:** Adds pointers to enhance triples for optimization.
- **Tree Form:**
 - **Syntax Tree:** Represents hierarchy and relationships based on grammar.
 - **Directed Acyclic Graph (DAG):** Eliminates redundant computations in the syntax tree.

After intermediate code generation, **code optimization** minimizes resource consumption, such as memory and computation time, while maintaining the program's correctness.

The final stage is **target code generation**, which produces machine-specific code. Key concepts include:

- **Registers:** Fast-access memory locations for storing intermediate values.
- **Instruction Selection:** Translates operations into machine-level instructions, such as:
 - **ADD:** Adds two values, storing the result in the first operand.
 - **SUB:** Subtracts the second operand from the first.
 - **JMP:** Unconditionally jumps to a specified location.

- MOV: Copies data between memory locations or registers.
- INC: Increments a value by 1.
- DEC: Decrements a value by 1.
- LOOP: Decrements a counter and repeats until it reaches zero.
- **Register Allocation:** Assigns program variables to registers to minimize memory accesses. For example:

```

R1 <- a
R2 <- b
R3 <- c
R4 <- d

MOV R3, c
MOV R4, d
MUL R3, R4
MOV R2, b
ADD R2, R3
MOV R1, R2
MOV a, R1

```

The use of assembly language in this stage enables a direct connection between high-level code and hardware, acting as a translator to execute machine-specific operations effectively.

The team, understanding the differences between a compiler and an interpreter, implements the intermediate code in postfix notation, also known as reverse Polish notation, which is characterized by not requiring parentheses to identify the order in which arithmetic operations are performed. In this notation, the operands are placed first, followed by the operators at the end.

Below, we present some examples of converting an expression to postfix notation:

1. Expression: $A + B$
 - Postfix: $AB +$
2. Expression: $A \cdot (B + C)$
 - Postfix: $ABC + \cdot$

From the examples above, we see that the operation transitions from infix to postfix notation by separating operators and operands, considering the order in which operators appear to establish their priority, and eliminating all parentheses.

To solve an expression in postfix notation, the following steps are applied:

1. While there are elements in the expression:
 - 1.1 **Element** = next element in the expression.
 - 1.2 If the element is an operand:
 - Push the element onto the stack.

1.3 If the element is an operator:

- Pop element b from the stack.
- Pop element a from the stack.
- Perform the operation with a and b .
- Push the result onto the stack.

In this project, a stack will be used to analyze each element in the postfix expression. This approach allows handling real and complex numbers, which are pushed onto the stack after conversion. Using the algorithm above, alongside tokenization to identify the desired operation, invalid operators will trigger an error message to prompt the user to adjust the input.

At the end of the interpreter's execution, the stack used to store the operands will contain the final result of the indicated operation.

4 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.

4.1 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.

4.1.1 Context-Free Grammar (CFG)

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

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

$$NT = \{ S, STATEMENT, EXPRESSION \}$$

- **Terminals (T):**

$$T = \{ id, print, lparen, rparen, plus, minus, times, divide, num, complex, sqrt \}$$

- **Production Rules (P):**

```
0. S -> STATEMENT
1. STATEMENT -> id assign EXPRESSION
2. STATEMENT -> print lparen EXPRESSION rparen
3. EXPRESSION -> EXPRESSION plus EXPRESSION
4. EXPRESSION -> EXPRESSION minus EXPRESSION
5. EXPRESSION -> EXPRESSION times EXPRESSION
6. EXPRESSION -> EXPRESSION divide EXPRESSION
7. EXPRESSION -> minus EXPRESSION
8. EXPRESSION -> lparen EXPRESSION rparen
9. EXPRESSION -> num
10. EXPRESSION -> complex
11. EXPRESSION -> id
12. EXPRESSION -> sqrt lparen EXPRESSION rparen
```

- **Initial Symbol (S):**

$$S = S$$

4.2 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:

4.2.1 Defined Tokens

```
# ASSIGN
assign = {=}
```

```
# LPAREN
lparen = {(}
```

```
# RPAREN
```



```

rparen = {}

# PLUS
plus = {+}

# MINUS
minus = {-}

# TIMES
times = {*}

# DIVIDE
divide = {/}

# MODULUS
modulus = {%}

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

# NUMBER
number = {0, 1, 2, ..., infinite +}

# COMPLEX
complex = {-1+2j, -1+3j, ...}

#SQRT
sqrt = {sqrt}

#PRINT
print = {print}

```

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.

4.3 FIRST() and FOLLOW()

The FIRST and FOLLOW functions play a fundamental role in the construction of parsing tables with context-free grammars. These functions help the parser predict which grammar rules to apply during the parsing process.

- **FIRST(X)**: This set contains all the terminal tokens that can appear at the beginning of any string derived from a non-terminal token X. In other words, it indicates which are the possible first tokens that can be derived from X.
- **FOLLOW(X)**: This set contains all terminal symbols that can appear immediately after X in a string derived from the initial symbol of the grammar.

The table below provides the FIRST and FOLLOW sets for each non-terminal in the specified grammar.

No Terminal	FIRST()	FOLLOW()
S	{ id, print }	{ \$ }
STATEMENT	{ id, print }	{ \$ }
EXPRESSION	{ minus, lparen, num, complex, id, sqrt }	{ rparen, \$ }

Table 1: FIRST () and FOLLOW () sets for our grammar.

4.4 Canonical Collection

- **State 0:** Description: starting point of the analysis. The analyzer expects to process a statement, either an assignment or a printout.
- **State 1:** Description: Acceptance status. Indicates that a complete statement has been successfully recognized and there are no more tokens to parse.
- **State 2:** Description: an identifier (id) has been recognized at the beginning of an assignment. It is now waiting for the assign (=) operator.
- **State 3:** Description: The print token has been recognized. It now expects a left parenthesis (to start an expression that will be evaluated and then printed.
- **State 4:** Description: after id assign has been recognized, the parser is ready to process the expression associated with the assignment.
- **State 5:** Description: Within a print instruction. The parser starts processing the expression to be evaluated within the parentheses.
- **State 6:** Description: An identifier (id) has been recognized as part of an expression. The parser may reduce it or continue processing.
- **State 7:** Description: an assignment (id assign expression) has been completed. The parser is ready to reduce this output and conclude it.
- **State 8:** Description: start of a unary expression or a new expression. The parser is waiting for unary operators (-) or valid components of an expression (such as numbers, identifiers, parentheses, etc.).
- **State 9:** Description: within a grouped expression. The parser is waiting for an expression to be completed within parentheses.
- **State 10:** Description: a number has been recognized as a complete expression. The parser can reduce this expression.
- **State 11:** Description: A complex number has been recognized as a complete expression. It can be reduced.
- **State 12:** Description: Beginning of a square root. The parser expects a left parenthesis to process the expression inside the root.
- **State 13-17:** Description: The parser is processing binary operations. Example: After recognizing PLUS expression, it waits for another expression to complete the operation.

- **State 18:** Description: A unary expression (as - expression) has been completed. It can be reduced as a valid output.
- **State 19:** Description: Within a grouped expression. The parser waits for closing parentheses to finish grouping.
- **State 20 and onwards:** Description: These states represent specific variations of expression reduction, binary operations or structures such as square roots and groupings. Example: Processing the closure of a square root or a complete operation as expression PLUS expression.

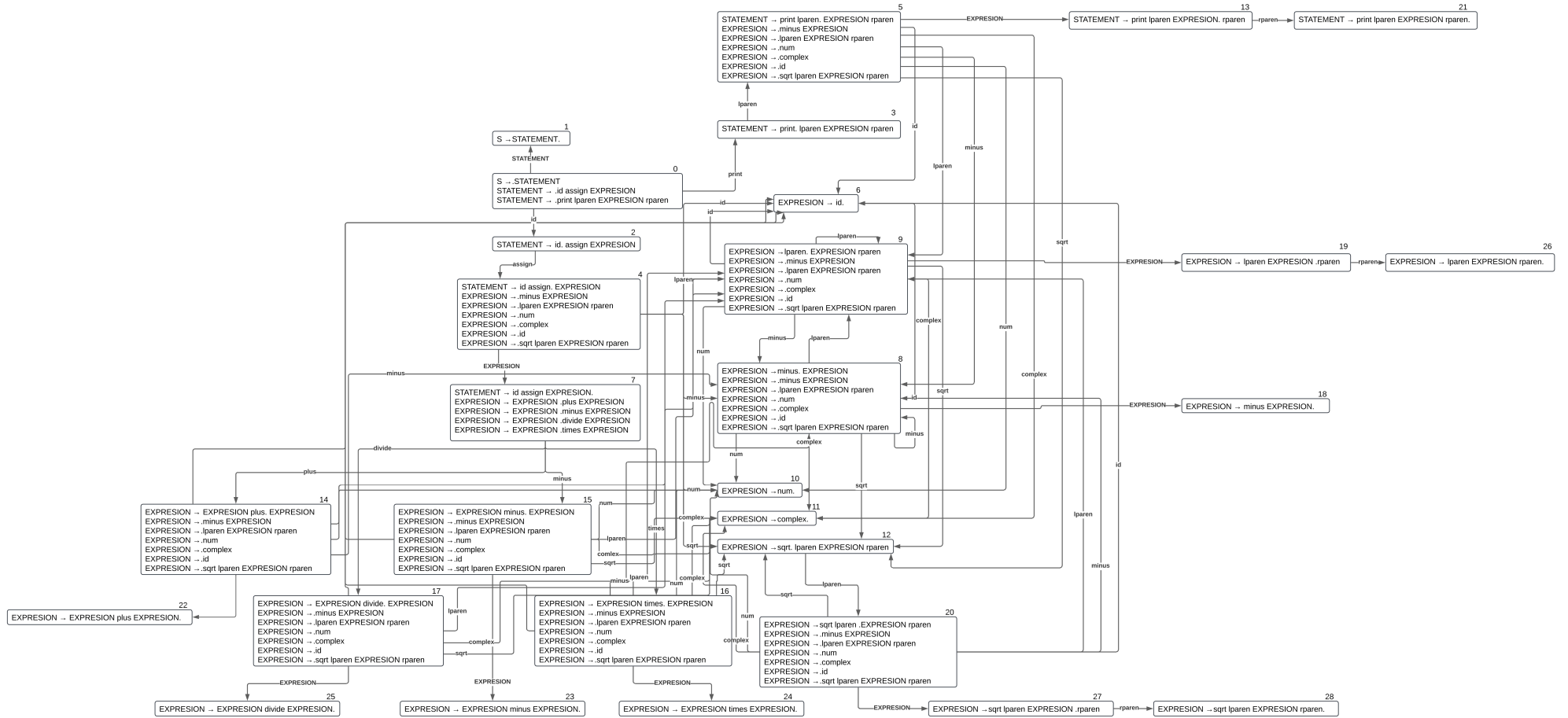


Figure 1: Canonical Collection States

4.5 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.

4.5.1 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.

Some important features for our parsing table:

- The grammar is context-free (CFG) and based on productions.
- The non-terminals are: `S`, `STATEMENT`, `EXPRESSION`.
- Terminals are: `id`, `assign`, `print`, `lparen`, `rparen`, `plus`, `minus`, `times`, `divide`, `num`, `complex`, `sqrt`.

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.

Each row of the table corresponds to a state of the LR automaton. Terminal symbol columns indicate actions based on the input token. And the columns of non-terminal symbols (`EXPRESSION`, `STATEMENT`) indicate transitions to new states after reductions.

Interpretation of actions:

1. Shift actions (s):

- `s#` : Shift to state `#`. The current token is stored in the stack and moved to the new state.
- Example: In state 0, when `id` is found, the action is `S2` (move to state 2).

2. Reduction actions (r):

- `r#` : Reduce using the production `#` of the grammar.
- Example: In state 7, when `plus` is found, `r7` is performed, which corresponds to `EXPRESSION -> minus EXPRESSION`.

3. Acceptance (acc):

- When the action `acc` is reached, the parsing ends correctly.
- Example: In state 1, the symbol `$` leads to the action `acc`.

Analysis of some states

- **State 0:**

- Start of the analysis.
- Possible actions:
 - * `id -> s2`: Move to state 2 to process an identifier.
 - * `print -> s3`: Process a print command.

- **State 7:**

- This state represents a potential reduction with production `r7 (EXPRESSION -> minus EXPRESSION)`.
- It also has a transition for `plus` and `times`, reflecting the hierarchy of operations.

- **State 13:**

- Here you see a shift with `lparen -> S20`, indicating that a sub-expression is expected in parentheses.

The next table is a typical implementation of an LR(1) parser. Each action in the table corresponds to one of the basic parsing operations: Shift to handle terminals, reduction to build derivations and transitions for non-terminals.

Parsing Table:

state	\$	assign	complex	divide	id	lparen	minus	num	plus	print	rparen	sqrt	times	EXPRESSION	STATEMENT
0					S2					S3					1
1	acc														
2		S4													
3						S5									
4			S11		S6	S9	S8	S10				S12		7	
5			S11		S6	S9	S8	S10				S12		13	
6	r11			r11			r11		r11		r11		r11		
7	r1			S17			S15		S14				S16		
8			S11		S6	S9	S8	S10				S12		18	
9			S11		S6	S9	S8	S10				S12		19	
10	r9			r9			r9		r9		r9		r9		
11	r10			r10			r10		r10		r10		r10		
12						S20									
13				S17			S15		S14		S21		S16		
14			S11		S6	S9	S8	S10				S12		22	
15			S11		S6	S9	S8	S10				S12		23	
16			S11		S6	S9	S8	S10				S12		24	
17			S11		S6	S9	S8	S10				S12		25	
18	r7			r7			r7		r7		r7		r7		
19				S17			S15		S14		S26		S16		
4			S11		S6	S9	S8	S10				S12		27	
21	r2														
22	r3			S17			r3		r3		r3		S16		
23	r4			S17			r4		r4		r4		S16		
24	r5			r5			r5		r5		r5		r5		
25	r6			r6			r6		r6		r6		r6		
26	r8			r8			r8		r8		r8		r8		
27				S17			S15		S14		S28		S16		
28	r12			r12			r12		r12		r12		r12		

Table 2: Parsing Table for the parsing part.

5 Results

5.1 Case Study

This section presents a practical example that demonstrates how the interpreter evaluates a mathematical expression using lexical analysis, syntactic analysis, and stack-based evaluation. The case under analysis is the evaluation of the expression $(a + b) \cdot (b + c)$, showing step-by-step how the interpreter processes the input, converts it to postfix notation, and evaluates the final result.

5.1.1 Grammar and Semantic Rules

Productions The grammar used for this case study includes the following productions:

```
STATEMENT → id assign EXPRESSION
STATEMENT → print lparen EXPRESSION rparen
EXPRESSION → EXPRESSION plus EXPRESSION
EXPRESSION → EXPRESSION minus EXPRESSION
EXPRESSION → EXPRESSION times EXPRESSION
EXPRESSION → lparen EXPRESSION rparen
EXPRESSION → id
EXPRESSION → num
```

Semantic Rules The semantic rules associated with each production define how the input is transformed and results are generated:

- **STATEMENT → id assign EXPRESSION**
variables[ID] = evaluate_postfix(expression)
- **EXPRESSION → EXPRESSION plus EXPRESSION**
expression.result = expression1.result + " " +
expression2.result + " +"
- **EXPRESSION → EXPRESSION times EXPRESSION**
expression.result = expression1.result + " " +
expression2.result + " *"
- **EXPRESSION → lparen EXPRESSION rparen**
expression.result = expression.result
- **EXPRESSION → id**
expression.result = variables[ID]
- **EXPRESSION → num**
expression.result = NUM

5.1.2 Step-by-Step Execution

The interpreter processes the following input:

```
a = 2
b = 3
c = 4
x = (a + b) * (b + c)
print(x)
```

Lexer (Lexical Analysis) The lexical analyzer breaks the input into recognizable tokens. The generated tokens are:

```
ID(a), ASSIGN(=), NUM(2)
ID(b), ASSIGN(=), NUM(3)
ID(c), ASSIGN(=), NUM(4)
```

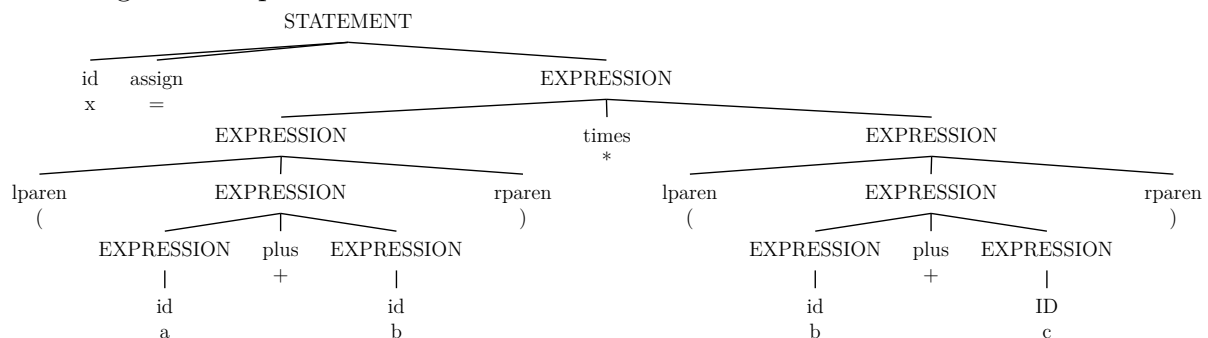
For the complete expression, the following token sequence is generated:

```
ID(x), ASSIGN(=), LPAREN(( ), ID(a), PLUS(+), ID(b), RPAREN( ) ) ,
TIMES(*), LPAREN(( ), ID(b), PLUS(+), ID(c), RPAREN( ) )
```

Parser (Syntactic Analysis) The parser constructs the parse tree for the expression $x = (a + b) \cdot (b + c)$ based on the grammar productions:

```
STATEMENT → id assign EXPRESSION
EXPRESSION → EXPRESSION times EXPRESSION
EXPRESSION → lparen EXPRESSION rparen
EXPRESSION → EXPRESSION plus EXPRESSION
EXPRESSION → id
```

The generated parse tree is as follows:



5.1.3 Syntax Directed Translation (SDT)

Parsing Process The following table demonstrates the step-by-step parsing process as part of the Syntax Directed Translation (SDT). Each reduction corresponds to a grammar rule applied to the input string, showcasing how the parser evaluates expressions and statements incrementally.

STACK	PARSE SYMBOLS	READ SYMBOL	ACTION
0		a	shift to 2
0 2	ID	=	shift to 4
0 2 4	ID =	2	shift to 10
0 2 4 10	ID = NUM		reduce by EXPRESSION
0 7	ID = EXPRESSION	(end of line)	reduce by STATEMENT
0 8	STATEMENT	b	shift to 2
0 8 2	STATEMENT ID	=	shift to 4
0 8 2 4	STATEMENT ID =	3	shift to 10
0 8 2 4 10	STATEMENT ID = NUM		reduce by EXPRESSION
0 8 7	STATEMENT ID = EXPRESSION	(end of line)	reduce by STATEMENT
0 8 8	STATEMENT STATEMENT	c	shift to 2
0 8 8 2	STATEMENT STATEMENT ID	=	shift to 4
0 8 8 2 4	STATEMENT STATEMENT ID =	4	shift to 10
0 8 8 2 4 10	STATEMENT STATEMENT ID = NUM		reduce by EXPRESSION
0 8 8 7	STATEMENT STATEMENT ID = EXPRESSION	(end of line)	reduce by STATEMENT
0 8 8 8	STATEMENT STATEMENT STATEMENT	x	shift to 2
0 8 8 8 2	STATEMENT STATEMENT STATEMENT ID	=	shift to 4
0 8 8 8 2 4	STATEMENT STATEMENT STATEMENT ID =	(shift to 9
0 8 8 8 2 4 9	STATEMENT STATEMENT STATEMENT ID = (a	shift to 2
0 8 8 8 2 4 9 2	STATEMENT STATEMENT STATEMENT ID = (ID	+	shift to 14
0 8 8 8 2 4 9 2 14	STATEMENT STATEMENT STATEMENT ID = (ID +	b	shift to 2
0 8 8 8 2 4 9 2 14 2	STATEMENT STATEMENT STATEMENT ID = (ID + ID)	reduce by EXPRESSION
0 8 8 8 2 4 9 15	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION	*	shift to 16
0 8 8 8 2 4 9 15 16	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION *	(shift to 9
0 8 8 8 2 4 9 15 16 9	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION * (b	shift to 2
0 8 8 8 2 4 9 15 16 9 2	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION * (ID	+	shift to 14
0 8 8 8 2 4 9 15 16 9 2 14	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION * (ID +	c	shift to 2
0 8 8 8 2 4 9 15 16 9 2 14 2	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION * (ID + ID)	reduce by EXPRESSION
0 8 8 8 2 4 9 15 16 9 2 15	STATEMENT STATEMENT STATEMENT ID = (EXPRESSION * (EXPRESSION)	reduce by EXPRESSION	
0 8 8 8 2 4 15	STATEMENT STATEMENT STATEMENT ID = EXPRESSION	(end of line)	reduce by STATEMENT
0 8 8 8 8	STATEMENT STATEMENT STATEMENT STATEMENT	print	shift to 20
0 8 8 8 8 20	STATEMENT STATEMENT STATEMENT STATEMENT PRINT	(shift to 9
0 8 8 8 8 20 9	STATEMENT STATEMENT STATEMENT STATEMENT PRINT (x	shift to 2
0 8 8 8 8 20 9 2	STATEMENT STATEMENT STATEMENT STATEMENT PRINT (ID)	reduce by EXPRESSION
0 8 8 8 8 20 15	STATEMENT STATEMENT STATEMENT STATEMENT PRINT (EXPRESSION)	reduce by STATEMENT	
0 8 8 8 8 8	STATEMENT STATEMENT STATEMENT STATEMENT STATEMENT		reduce by STATEMENT

Table 3: Parsing Process for the Input String $\{a = 2; b = 3; c = 4; x = (a + b) * (b + c); \text{print}(x)\}$

Explanation of the Parsing Table of the Case Study The parsing table above provides a detailed breakdown of the parsing process, structured as follows:

1. Stack (STACK** Column)** This column shows the internal state of the parser:

- The states in the stack correspond to transitions defined in the grammar rules.
- Symbols such as ID, NUM, or EXPRESSION are added as they are processed.
- Reductions replace processed symbols with the corresponding nonterminal, reflecting how grammar rules are applied.

2. Parse Symbols (PARSE SYMBOLS** Column)** This column shows:

- The intermediate derivation of the input string.
- Each step demonstrates how subtrees (e.g., EXPRESSION) are constructed incrementally.

3. Read Symbol (READ SYMBOL Column) The token being processed at each step is shown here:

- ID, NUM, and operators are processed in sequence.
- The parentheses () help define precedence in the expression.

4. Action (ACTION Column) This column describes how the parser handles the current symbol:

- **Shift:** The symbol is added to the stack and transitions to a new state.
- **Reduce:** A grammar rule is applied to replace a sequence of tokens with a nonterminal.
- **Accept:** The input is successfully parsed.

Final Notes The parser reduces all input lines into individual STATEMENTS. At the end:

- The final reduction consolidates all parsed STATEMENTS into a single sequence.
- This reflects the full derivation of the program.

Semantic Actions During the analysis, the SDT generates the following semantic actions:

1. For the production $\text{STATEMENT} \rightarrow \text{id assign EXPRESSION}$:

$\text{variables['a']} = 2$ (from the statement $a = 2$)
 $\text{variables['b']} = 3$ (from the statement $b = 3$)
 $\text{variables['c']} = 4$ (from the statement $c = 4$)

Each variable assignment is parsed, and the right-hand expression (NUM) is evaluated directly.

2. For the production $\text{EXPRESSION} \rightarrow \text{EXPRESSION plus EXPRESSION}$:

$\text{expression1.result} = "2 \ 3 \ +"$ (result of $a + b$)
 $\text{expression2.result} = "3 \ 4 \ +"$ (result of $b + c$)

These reductions occur when parsing the subexpressions inside parentheses: $(a + b)$ and $(b + c)$.

3. For the production $\text{EXPRESSION} \rightarrow \text{EXPRESSION times EXPRESSION}$:

$\text{expression.result} = "2 \ 3 \ + \ 3 \ 4 \ + \ *"$

This reduction combines the results of $(a + b)$ and $(b + c)$ using the multiplication operator.

4. For the production $\text{STATEMENT} \rightarrow \text{id assign EXPRESSION}$:

`variables['x'] = evaluate_postfix("2 3 + 3 4 + *") = 35`

The final assignment stores the evaluated result of the entire expression $(a+b) \cdot (b+c)$ into the variable `x`.

5. For the production $\text{STATEMENT} \rightarrow \text{print lparen EXPRESSION rparen}$:

`output = variables['x']` (prints the value of `x`, which is 35)

The `print(x)` statement accesses the variable `x` and outputs its stored value.

Stack Evaluation The postfix notation generated for the expression $(a+b) \cdot (b+c)$ is:

`"2 3 + 3 4 + *"`

The evaluation process uses a stack and follows these detailed steps:

1. **Read the first token:** 2

- It is a number, perform **push** on the stack.
- Stack state: [2]

2. **Read the next token:** 3

- It is a number, perform **push** on the stack.
- Stack state: [2, 3]

3. **Read the next token:** +

- It is an operator. Perform **pop** on the top two numbers from the stack (3 and 2).
- Evaluate $2 + 3 = 5$.
- Push the result (5) back onto the stack.
- Stack state: [5]

4. **Read the next token:** 3

- It is a number, perform **push** on the stack.
- Stack state: [5, 3]

5. **Read the next token:** 4

- It is a number, perform **push** on the stack.
- Stack state: [5, 3, 4]

6. **Read the next token:** +

- It is an operator. Perform **pop** on the top two numbers from the stack (4 and 3).

- Evaluate $3 + 4 = 7$.
- Push the result (7) back onto the stack.
- Stack state: [5, 7]

7. Read the next token: *

- It is an operator. Perform **pop** on the top two numbers from the stack (7 and 5).
- Evaluate $5 \cdot 7 = 35$.
- Push the result (35) back onto the stack.
- Final stack state: [35]

The final result, stored in the stack as [35], corresponds to the value of the expression $(a + b) \cdot (b + c)$, evaluated with $a = 2$, $b = 3$, and $c = 4$.

Results The final result of the evaluation is stored in the variable `x` and then displayed by the interpreter:

```
Result: 35.0
```

5.2 Empire

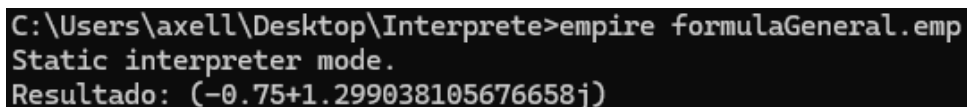
The interpreter "*empire*" delivers robust functionality, operating in two distinct modes: **static execution** and **interactive mode**. Below, we summarize its main results:

1. Processing **.emp** Files

In static mode, the interpreter processes **.emp** files line-by-line, executing instructions such as assignments, mathematical calculations, and print commands. For example:

```
a = 5
b = 4
c = 3
x = (-b + sqrt((b*b) - 4*(a*c))) / (2*a)
print(x)
```

Result: $(-0.44+0.663j)$ (a complex number, depending on the input values).



```
C:\Users\axell\Desktop\Interprete>empire formulaGeneral.emp
Static interpreter mode.
Resultado: (-0.75+1.299038105676658j)
```

Figure 2: Static mode execution of the interpreter "*empire*". The interpreter reads a file and evaluates the formula, outputting the result in complex format.

2. Interactive Mode

In interactive mode, there are two ways to execute the interpreter:

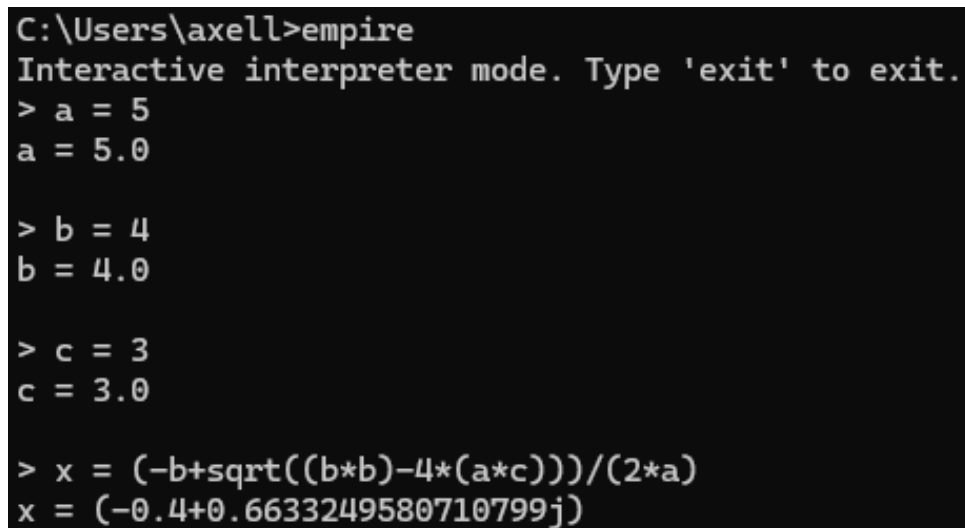
- Run the command `empire` in the terminal (if the environment variable is set up).
- Alternatively, execute it using `./empire` in the terminal or by launching the provided executable file.

Once the interpreter is running, users can input commands in real-time. For example:

```
> a = 5
> b = 4
> c = 3
> x = (-b + sqrt((b*b) - 4*(a*c))) / (2*a)
> print(x)
```

Result: $(-0.44+0.663j)$.

Below is an example of the interactive mode:

A screenshot of a terminal window showing the execution of the 'empire' interpreter. The prompt is 'C:\Users\axell>empire'. The interpreter responds with 'Interactive interpreter mode. Type 'exit' to exit.' The user enters several commands: '> a = 5', '> b = 4', '> c = 3', and '> x = (-b+sqrt((b*b)-4*(a*c)))/(2*a)'. The interpreter echoes each command and shows the resulting values: 'a = 5.0', 'b = 4.0', 'c = 3.0', and 'x = (-0.4+0.6633249580710799j)'.

```
C:\Users\axell>empire
Interactive interpreter mode. Type 'exit' to exit.
> a = 5
a = 5.0

> b = 4
b = 4.0

> c = 3
c = 3.0

> x = (-b+sqrt((b*b)-4*(a*c)))/(2*a)
x = (-0.4+0.6633249580710799j)
```

Figure 3: Interactive mode example with real-time variable assignment and computation.

Additionally, the executable file for launching the interpreter can be represented as follows:

3. Support for Advanced Mathematical Expressions

The interpreter handles complex calculations, including operations with square roots, powers, and algebraic expressions. These are evaluated efficiently using intermediate **postfix notation**.

4. Flexible Variable Assignments

It supports the dynamic assignment of variables (e.g., $a = 5$, $b = 4$), which can be reused in subsequent calculations, making it suitable for iterative testing and simulations.



Figure 4: Icon or executable used to launch the "empire" interpreter.

5. Real-Time Output of Results

When using interactive mode, results of expressions or commands are printed directly to the console, allowing immediate verification of calculations.

6. Implementation with PLY

Using **PLY**, the interpreter leverages lexical and syntactic analysis to process instructions. This includes:

- Tokenizing keywords, operators, variables, and numbers.
- Parsing the input to ensure valid syntax and correctly handle operator precedence.
- Generating postfix notation for efficient evaluation.

5.2.1 Postfix

The conversion to **postfix notation** (or **Reverse Polish Notation, RPN**) is a key feature of our interpreter "*empire*". This process transforms infix expressions (e.g., $a+b\cdot c$) into a format that is easier for a machine to evaluate: postfix notation ($abc \cdot +$).

Conversion Process Using PLY With the help of **PLY** (**P**ython **L**ex-**Y**acc), we defined a **lexer** and a **parser** to analyze and process expressions. The steps are as follows:

1. Lexical Analysis (Lexer) The lexer identifies the fundamental components of expressions, referred to as *tokens*. For instance:

- Identifiers (e.g., a, b, c)
- Operators (e.g., $+, -, *, /, (,)$)
- Numbers (e.g., $1, 2.5$)

These components are defined using lexical rules, often implemented via regular expressions.

2. Syntax Analysis (Parser) The parser uses grammatical rules to interpret how tokens should combine. This includes:

- **Operator Precedence:** For example, $*$ and $/$ have higher precedence than $+$ and $-$.
- **Parentheses Handling:** Expressions enclosed in parentheses have the highest precedence and are processed first.

3. Conversion to Postfix Notation During parsing, we implement a stack-based approach to convert infix expressions to postfix notation:

- Operators are pushed onto the stack.
- Operands (e.g., variables or numbers) are directly added to the output.
- When encountering a closing parenthesis $)$, the stack is popped until the corresponding opening parenthesis is found.

Implementation in PLY In PLY, parsing rules are implemented as functions associated with grammar productions. For example:

```
def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Instead of directly evaluating the expression, we adapt it to generate postfix output:

```
def p_expression_binop_postfix(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
    p[0] = f"{p[1]} {p[3]} {p[2]}"
```

This approach produces a postfix string as the result for each parsed expression.

Advantages of Postfix Notation

- **Ease of Evaluation:** Once in postfix form, expressions can be directly evaluated using a stack without considering precedence or parentheses.
- **Efficiency:** Reduces computational complexity since expressions are processed in a single pass.
- **Modular Implementation:** Separating lexical/syntactic analysis from evaluation results in a clean and maintainable interpreter design.

Example Given the infix expression:

$$a + b \cdot c$$

The postfix conversion results in:

$$a b c \cdot +$$

Evaluation:

1. Push a , b , and c onto the stack.
2. Process the operator $*$, calculating $b \cdot c$.
3. Process the operator $+$, calculating $a + (b \cdot c)$.

This stack-based approach, implemented with PLY, enables the interpreter to handle complex mathematical expressions efficiently and reliably.

5.2.2 Case Study

```
C:\Users\axell>empire
Interactive interpreter mode. Type 'exit' to exit.
> a = 2
a = 2.0

> b = 3
b = 3.0

> c = 4
c = 4.0

> x = (a+b)*(b+c)
x = 35.0

> exit
```

Figure 5: Running the Case study

Summary

"empire" is a versatile interpreter capable of processing both complete scripts and real-time user inputs. It reliably delivers accurate results for mathematical calculations and variable manipulations, supported by a well-structured design built with PLY.

6 Conclusions

An interpreter is a program that translates and executes instructions written in a programming language directly, without the need to convert them to an intermediate language such as machine code. This makes it a powerful and practical tool for high-level languages, and is what led us to choose it as our final implementation.

The construction of this interpreter as the final project of the course was the result of a semester of continuous learning, where we studied and implemented the different stages of a Compiler/Interpreter/Assembler. We started with the implementation of the lexical analyzer, a key stage where we learned to decompose the source code into more basic components called tokens. Subsequently, we worked on the design and construction of the parser using context-free grammars and Parse Tables, which allowed us to validate the language structure, and we used Semantic Directed Translation (SDT) to extend our capabilities towards semantic validation, which was the last implementation we performed. This whole process was fundamental to achieve our main goal: to build a functional interpreter.

Our interpreter (Empire) managed to implement a simple but complete language, with support for arithmetic operations, assignments, control structures and other functionalities specifically designed in the previously defined grammar. From the reading of the instructions to their interpretation and execution, this project reflected the integration of each of the concepts worked on in the previous stages of the semester. In addition, the interpreter was also able to handle lexical as well as syntactic and semantic errors, providing feedback to the user to correct problems in their code.

This project also presented many challenges. During development, we faced problems as a team at different stages, for example in the construction of the Parse Tables, in the implementation of the semantic rules, particularly in cases where the logic of the language required more complex evaluations or when trying to handle recursive structures. However, each of these challenges allowed us to reinforce our understanding of the theoretical principles and to find practical solutions with the help and support of each member of this team. We were able to recognize not only the importance of an interpreter and each stage that composes it, but also the importance of teamwork and the understanding the theoretical bases, which resulted in the fulfillment of our main objective: the construction of a functional interpreter.

References

- [1] Del Estado de Hidalgo, U. A., “Aspirantes - UAEH,” *Universidad Autónoma del Estado de Hidalgo*, [Online]. Available: <http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/autocontenido/autocon/index.html>. [Accessed: 25-Nov-2024].
- [2] A. V. Aho, *Compilers: Principles, Techniques, and Tools*, Addison Wesley Longman, 2000.
- [3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2006.
- [4] M. Navarro, *Modelos Abstractos de Cómputo I Autómatas y Lenguajes Formales*, 1st ed.
- [5] G. A. Alvarez Alvarez, *COMPILADORES*, 1st ed.
- [6] E. R. A. Rodriguez, *HERRAMIENTAS DE SOFTWARE PARA LA FACILITACION DE LA COMPRENSION, CONSTRUCCION Y TESTEO DE COMPILADORES*.
- [7] IA for paraphrasing, “ChatGPT,” [Online]. Available: <https://chatgpt.com/>. [Accessed: 25-Nov-2024].