University of Isfahan
Computer Engineering Department

Trust Compiler Report

Students:
Zahra Masoumi - Matin Azami

Students ID:
4003623034 - 4003623003

Professor:
Dr. Shafiee

Contents

Chapter 1

Lexical Analyzer

1-1- Introduction

The lexical analyzer is the first phase of the compiler that processes the source code. It reads the input characters and groups them into meaningful sequences called tokens using formal grammar rules. This chapter describes the key functions of our lexical analyzer implementation for the Trust language compiler, including their formal grammar definitions.

The extract function in a lexical analyzer is responsible for processing each line of source code and extracting tokens from it. It does so by iterating through each character of the line and using various helper functions (like is_space, is_comment, is_operator, etc.) to identify different types of tokens such as whitespace, comments, operators, keywords, numbers, strings, and identifiers.

Breakdown of the Function: Line Number Tracking:

The function starts by incrementing the line_number variable every time it processes a new line.

Tokenization Process:

The function uses a loop to process each character of the input line.

It checks each character to determine what type of token it represents by calling several helper functions, such as:

is_space: Checks for spaces or whitespace.

is_comment: Checks for comments.

is_operator: Checks for operators (+, -, *, etc.).

is_keyword: Checks for keywords (e.g., if, return, loop).

is_hexadecimal: Checks for hexadecimal numbers.

is_decimal: Checks for decimal numbers.

is_id: Checks for identifiers (variable names, function names).

is_string: Checks for string literals.

After each check, if a valid token is found (i.e., the get_type() method does not return Invalid), the token is added to a collection (e.g., tokens).

If no valid token is found, an Invalid token is generated for the character, indicating an error, and the num_errors counter is incremented.

Explanation of the Importance of Order in Token Extraction: The order of token extraction is important because certain types of tokens might overlap or resemble other types of tokens. Specifically:

Keyword vs. Identifier:

Keywords (such as i32, if, loop) and identifiers (like variable names) may look similar, but they have different meanings.

For example, i32 is a keyword, while integer is an identifier. If we first check for keywords, we ensure that reserved words are treated correctly as keywords rather than as identifiers.

In the order in extract(), the function checks for keywords before identifiers. This means if a word matches both a keyword and an identifier (like int), it will be categorized as a keyword first.

Why it's important: If we checked identifiers first, we could mistakenly treat keywords as just regular identifiers, which would cause issues in interpreting the code. Identifier names should not be the same as any keyword.

Comment vs. Other Tokens:

Comments (e.g., // comment) should be ignored during tokenization, as they are not part of the executable code. By checking for comments early in the process (before other tokens), we ensure that they are excluded from further tokenization.

If we processed comments after identifying other tokens, we might accidentally treat comment text as part of the code, which would be incorrect.

Whitespace and Other Tokens:

Whitespace is often used for separation between tokens, but it's not a token that we need to process further. Checking for whitespace early in the process allows us to skip over unnecessary spaces, keeping the token stream clean and efficient.

If we processed whitespace later, we might end up with unnecessary whitespace tokens or miss tokens that should be grouped together.

Escaped Characters in Strings:

String literals can contain escape sequences (like ör

). Checking for strings and escape sequences properly ensures that strings are parsed correctly without treating the escape sequences as individual tokens.

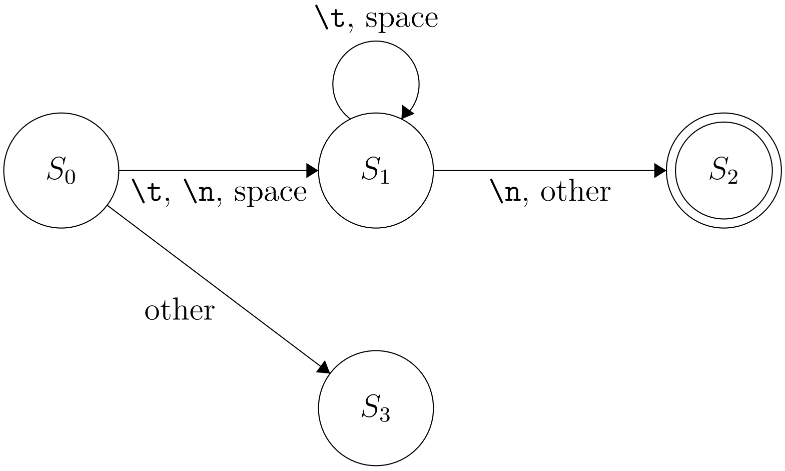
Why the Order Matters: Efficiency: Checking for the most specific tokens first (e.g., keywords, comments) helps the lexer efficiently identify and discard irrelevant characters or words early on. This reduces unnecessary processing later.

Correct Token Classification: If the lexer were to check for a more general token (like an identifier) before a specific one (like a keyword), it could misclassify code. For example, if would be classified as an identifier if we checked for identifiers first, rather than recognizing it as a keyword.

Preventing Conflicts: By checking for comments and spaces early, we prevent the analyzer from mistakenly treating non-code content as part of the program logic.

1-2- White Spaces

1-2-1- Automata

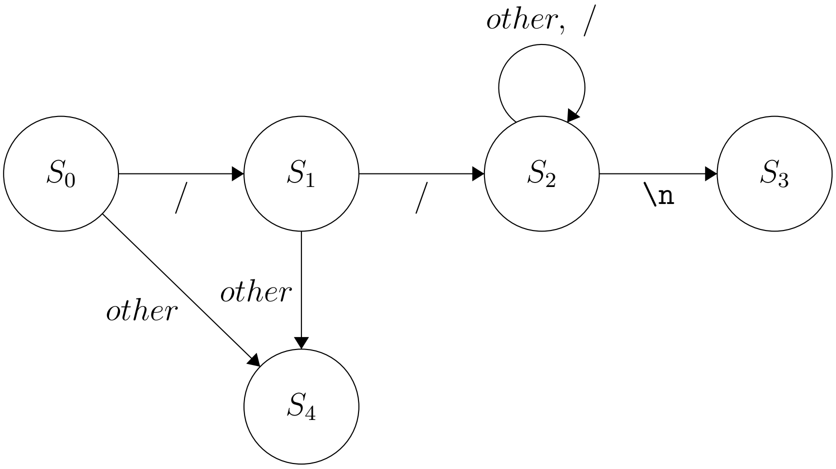


1-2-2- Detailed Explanation

The is_space function in a lexical analyzer checks whether a part of the source code consists of whitespace characters like spaces, tabs, or newlines. It scans the input line character by character using a state machine approach. If it encounters whitespace, it transitions through states, eventually returning a token indicating that whitespace has been found. If the characters are not whitespace, it flags them as invalid. This process is crucial in a lexical analyzer for properly separating tokens in the source code.

1-3- Comments

1-3-1- Automata

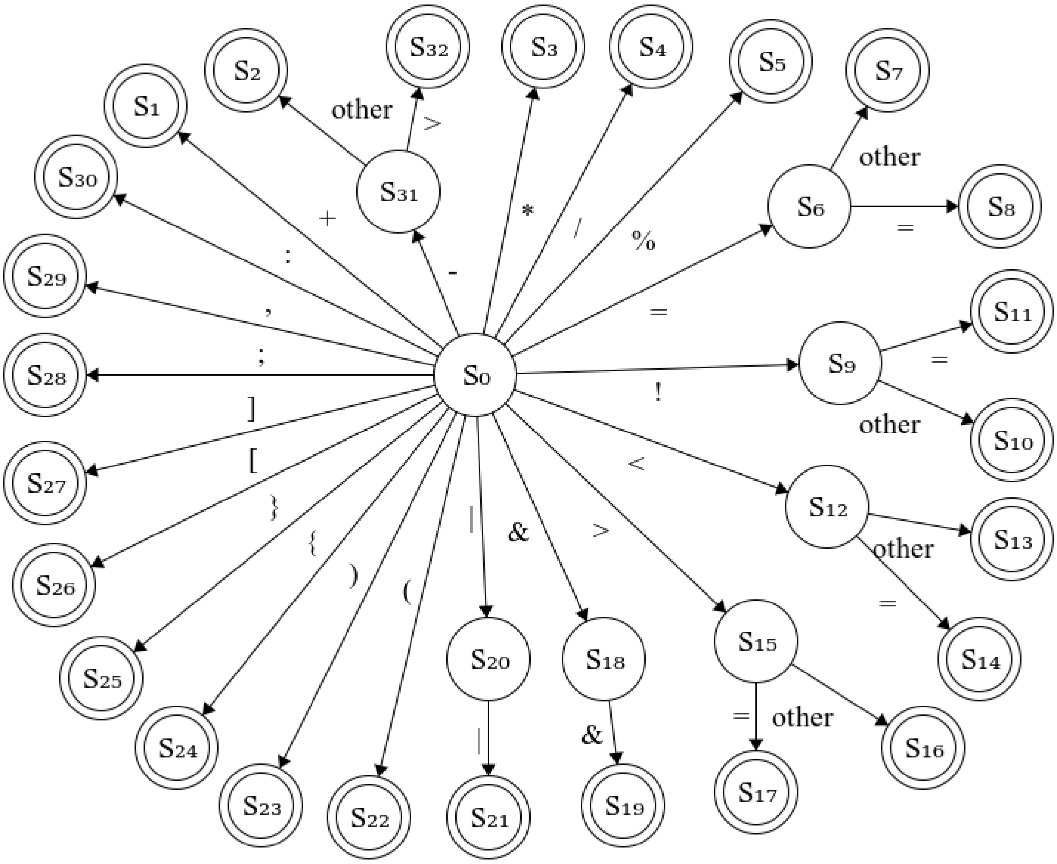


1-3-2- Detailed Explanation

The is_comment function in a lexical analyzer is responsible for detecting comments in a line of code, typically for single-line comments. It checks if a part of the source code starts with a comment indicator (//), and if so, captures the content of the comment until the end of the line or an invalid sequence is encountered. The function uses a state machine approach to handle different stages of comment detection: it starts by checking for the initial comment markers (//), then captures the content inside the comment, and finally returns a token representing the comment. If an invalid sequence is encountered (e.g., characters not following the expected comment format), the function returns an ”invalid” token. This is an essential part of the lexical analysis process for ignoring comments while parsing the code.

1-4- Operators

1-4-1- Automata

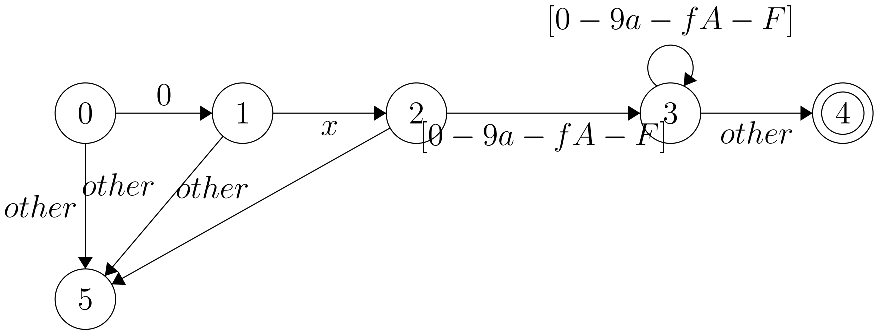


1-4-2- Detailed Explanation

The `is_operator` function in a lexical analyzer is responsible for detecting and categorizing operators and punctuation marks in a line of code. This function uses a state machine approach to handle the detection of various operators, including arithmetic operators (e.g., +, −, ∗), relational operators (e.g., ==, <, ≥), logical operators (e.g., &&, ||), and other symbols (e.g., parentheses, semicolons, and commas). The function scans the line character by character, transitioning between states as it encounters known operator symbols. If it identifies a valid operator or symbol, it returns a corresponding token (such as `T_AOp_AD` for addition, `T_ROp_E` for equality check, or `T_Semicolon` for semicolon). If an unrecognized character is encountered, the function returns an “Invalid” token, signaling an error. This function plays a critical role in the lexical analysis phase of a compiler, where operators and punctuation are identified and classified for further processing.

1-5- HexaDecimal

1-5-1- Automata



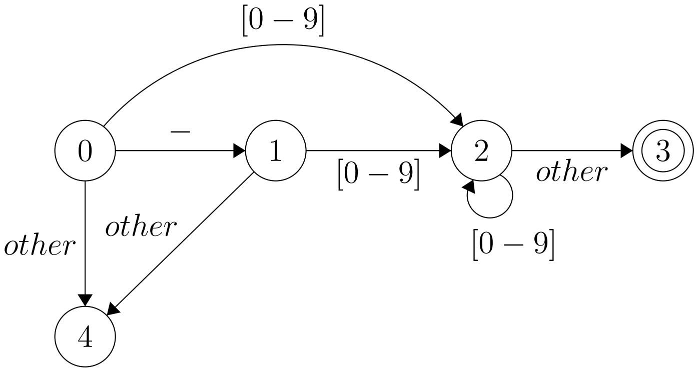
1-5-2- Detailed Explanation

The `is_hexadecimal` function implements a six-state DFA for strict hexadecimal number validation. State 0 requires a leading ’0’, transitioning to state 1 which demands either ’x’ or ’X’. State 2 then requires at least one hexadecimal digit (0-9, a-f, A-F), with state 3 consuming subsequent hexadecimal digits. The final state (state 4) validates that proper hexadecimal digits follow the prefix before returning a `T_Hexadecimal` token.

This implementation enforces the language specification that hexadecimal literals must have at least one digit after the prefix. The function uses `isxdigit()` for proper character classification and preserves the original casing of hexadecimal digits in the token content. Invalid sequences like ”0x” without subsequent digits are properly rejected with `Invalid` tokens.

1-6- Decimal

1-6-1- Automata



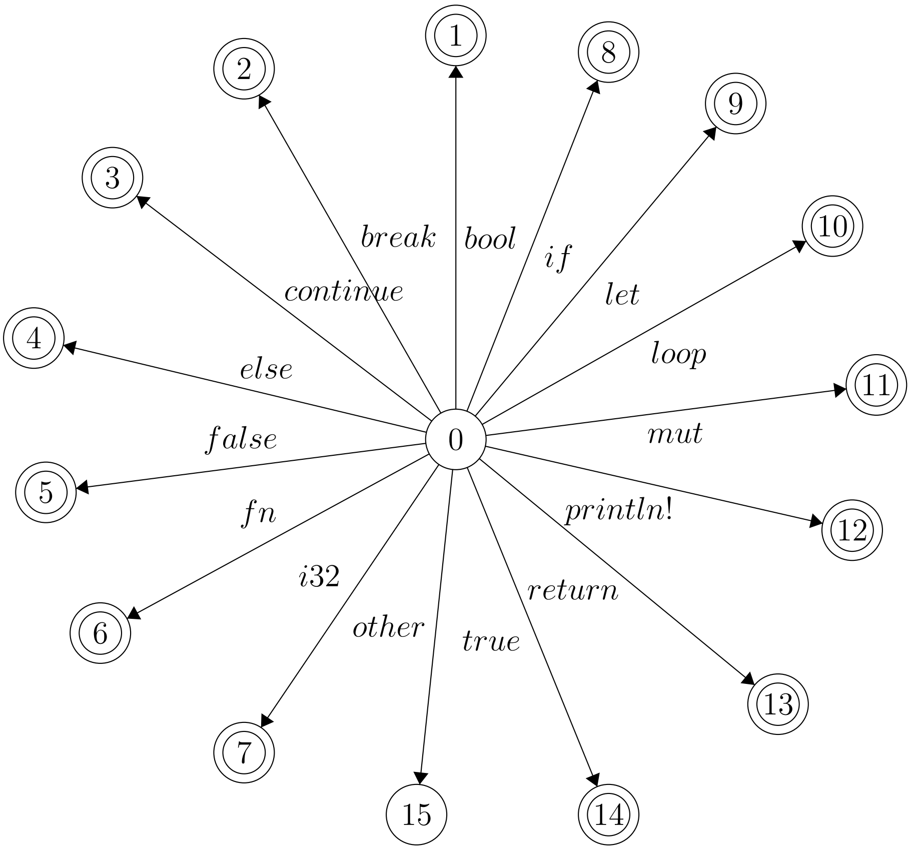
1-6-2- Detailed Explanation

The `is_decimal` function implements a five-state DFA for recognizing both positive and negative decimal integers. State 0 handles the optional minus sign, transitioning to state 1 if present. State 2 requires at least one digit (0-9) and continues consuming digits until a non-digit character is encountered. The final state (state 3) validates that at least one digit was collected before returning a `T_Decimal` token.

This implementation properly handles edge cases like standalone minus signs (invalid) and maximum number length constraints. The function also preserves the original string representation of the number (including leading zeros) in the token content for accurate source representation. Number validation occurs without conversion to numeric values, maintaining precision during lexical analysis.

1-7- Keywords

1-7-1- Automata



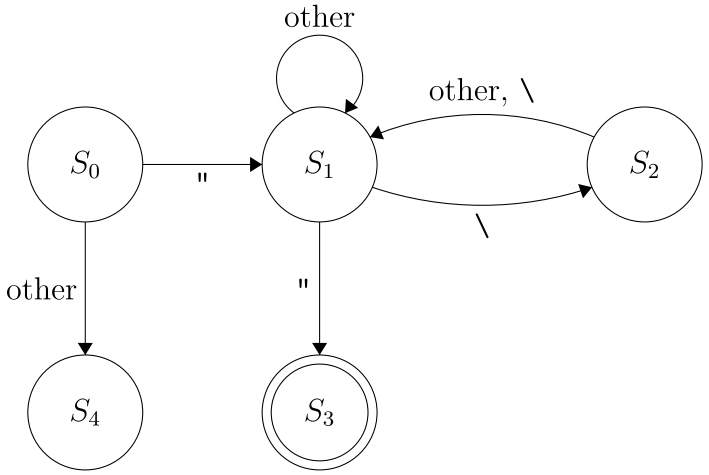
1-7-2- Detailed Explanation

The `is_keyword` function implements a deterministic finite automaton (DFA) that recognizes the 14 reserved keywords of the Trust language. Each keyword is treated as a fixed literal pattern that must match exactly, with case sensitivity. The function first checks for the longest possible keyword match (”println!”) before proceeding to shorter ones to ensure correct tokenization. After finding a potential match, it verifies that the match isn’t part of a larger identifier by checking the following character isn’t a letter, digit, or underscore. This strict validation prevents keywords from being misidentified within identifiers while maintaining language semantics.

The implementation uses a state machine that transitions through each character of potential keywords, with failure paths that efficiently fall back to identifier recognition when no keyword matches. This approach provides O(n) time complexity where n is the length of the longest keyword, making it highly efficient for lexical analysis.

1-8- Strings

1-8-1- Automata

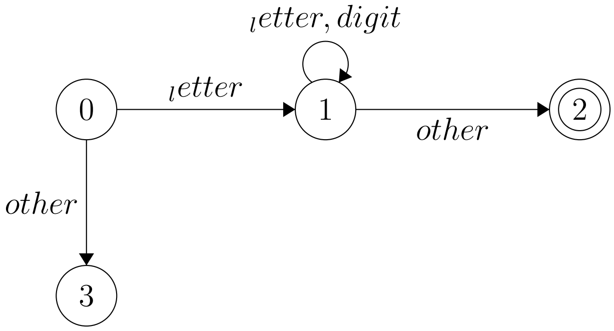


1-8-2- Detailed Explanation

The is_string function in a lexical analyzer is designed to detect and process string literals in the source code. It scans a line character by character, looking for strings enclosed in double quotes ("). The function handles different scenarios, such as escape sequences (̀) and the correct closing of the string literal. Using a state machine, it transitions through various states: it starts by detecting the opening quote, then handles escape characters, and finally detects the closing quote to complete the string. If the string is valid, it returns a token of type T_String containing the string content. If an invalid string is encountered (e.g., the string is incomplete or malformed), it returns an Invalid token. This function is a key part of the lexical analysis process, ensuring strings are correctly identified and tokenized for further processing.

1-9- IDs

1-9-1- Automata



1-9-2- Detailed Explanation

The is_id function implements the identifier recognition grammar through a four-state DFA. The initial state requires the first character to be either an underscore or letter (state 1). Subsequent characters transition to state 2 where letters, digits, or underscores are accepted. The function continues consuming characters until encountering a non-matching character, at which point it verifies the collected identifier doesn't match any keyword (using is_keyword) before returning a T_Id token.

This implementation carefully handles Unicode characters by using isalpha() and isalnum() functions for proper character classification. The function also maintains position information to support precise error reporting when invalid identifiers are encountered. The lookahead verification ensures identifiers don't conflict with language keywords while allowing maximum flexibility in naming conventions.

Chapter 2

Syntax Analyzer

2-1- Introduction

The syntax analyzer, or parser, is responsible for checking the grammatical structure of the token sequence produced by the lexical analyzer. In this project, we implemented a Predictive Parser, a type of top-down parser that uses an LL(1) parsing table to decide which grammar rule to apply based on the current input token and the top of the parsing stack.

2-2- Method

We used a predictive parsing algorithm which:

Reads the sequence of tokens generated by the lexer.

Uses a parse table constructed from a context-free grammar.

Builds the corresponding parse tree without backtracking.

The grammar was first transformed to eliminate left recursion and perform left factoring to ensure compatibility with LL(1) parsing.

2-2-1- Workflow

The parser initializes with the start symbol on the stack.

At each step, it consults the parse table using the top of the stack and current input token.

Based on the table entry, it replaces the non-terminal or matches a terminal.

This continues until the input is consumed and the stack is empty, resulting in a valid parse tree.

2-2-2- Error Handling

To manage syntax errors during parsing, we implemented panic-mode error recovery using synchronizing tokens (sync tokens).

When an error is detected (i.e., no valid entry in the parse table), the parser:

Discards input tokens until a sync token is found ; , }, or a specific keyword.

Pops non-terminals from the stack until a state with a valid continuation is reached.

This approach prevents the parser from getting stuck and allows it to continue parsing the rest of the input, enabling multiple error detections in a single run.

Using sync tokens improves the robustness of the parser and helps provide meaningful error messages for incomplete or incorrect code.

2-2-3- Output

The final output is a parse tree that represents the syntactic structure of the source code. If the input does not match the grammar, the parser reports a syntax error with its position.

2-3- Grammar Overview

The grammar defines the valid structure of our language, covering functions, statements, expressions, and types. It has been transformed (removing left recursion and applying left factoring) to make it suitable for LL(1) parsing.

The full grammar is divided into the following main parts:

2-3-1- Program Structure

```
<program>      → <func_ls>
<func_ls>      → <func> <func_ls> @ <stmt_ls>
```

This defines a program as a sequence of functions or statements.

2-3-2- Function Structure

```
<func>         → T_Fn T_Id T_LP <func_args> T_RP <func_type> T_LC <stmt_ls> <return_stmt> T_
<program>      → <func_ls>
<func_ls>      → <func> <func_ls> @ <stmt_ls>
<func>         → T_Fn T_Id T_LP <func_args> T_RP <func_type> T_LC <stmt_ls> <return_stmt> T_
```

This section specifies how function, arguments and optional return types are declared for functions.

2-3-3- Statements

```
<stmt_ls>          → <stmt> <stmt_ls> @
<stmt>             → <var_declaration> T_Semicolon @ T_Id <stmt_after_id> T_Semicolon @ ...
<loop_stmt>        → T_Loop T_LC <stmt_ls> T_RC
<if_stmt>          → T_If <exp> T_LC <stmt_ls> T_RC <else_part_opt>
<break_stmt>       → T_Break T_Semicolon
<continue_stmt>    → T_Continue T_Semicolon
<return_stmt>      → T_Return <exp> T_Semicolon @
```

This part of the grammar handles declarations, control structures, and function returns.

2-3-4- Variable Declarations

```
<var_declaration>  → T_Let <mut_opt> <pattern> <type_opt> <assign_opt>
<mut_opt>          → T_Mut @
<type_opt>         → T_Colon <type> @
<assign_opt>       → T_Assign <exp> @
```

Supports mutable (mut) declarations, optional type annotations, and optional assignments.

2-3-5- Expressions

```
<exp>              → <log_exp>
<log_exp>          → <rel_exp> <log_exp_tail>
<rel_exp>          → <eq_exp> <rel_exp_tail>
<eq_exp>           → <cmp_exp> <eq_exp_tail>
<cmp_exp>          → <arith_exp> <cmp_exp_suf>
<arith_exp>        → <arith_term> <arith_exp_tail>
<arith_term>       → <arith_factor> <arith_term_tail>
<arith_factor>     → literals | variables | function calls | array indexing | logical NOT
```

Expressions support logical operations, comparisons, arithmetic, function calls, and indexing.

2-3-6- Types and Arrays

```
<type>             → T_Int @ T_Bool @ T_LP <type_ls> T_RP @ T_LB <opt_type> T_Semicolon <opt
<opt_type>         → (same structure) @
<type_ls>          → <type> <type_ls_tail> @
```

This supports basic types (int, bool) and compound types (tuples, arrays).

2-3-7- Function Calls and Indexing

```
<fac_id_opt>       → T_LP <exp_ls_call> T_RP @ T_LB <exp> T_RB @
<exp_ls_call>      → <exp_ls> @
<exp_ls>           → <arg_item> <exp_ls_tail> @
<arg_item>         → <exp> <arg_item_suffix>
```

This handles function calls with arguments and optional named arguments.

2-3-8- Print Statements

```
<println_stmt>     → T_Print T_LP <println_args> T_RP T_Semicolon
<println_args>     → T_String <println_format_args_opt> @ <exp_non_string>
<println_format_args_opt> → T_Comma <println_format_args_list> @
```

This allows for print() statements with string formatting and expressions.

Chapter 3

Semantic Analyzer

Chapter 4

Code Generator

Chapter 5

Reference

PouyaRahimpour/Compiler: A compiler implementation available at https://github.com/PouyaRah This project was used as a reference for understanding compiler design, parsing techniques, and implementation details.

OpenAI ChatGPT: An AI language model developed by OpenAI, used to assist in generating documentation, explanations, and clarifications. Available at https://openai.com/chatgpt.