Phase 2- The Lexical Analyzer

CSE439-Design of Compilers
19 April 2025

Prepared For:

Dr. Waffa Samy

Eng. Mohamed Abdelmegeed

Prepared by:

Philopater Guirgis 22P0250

Hams Hassan 22P0253

Ahmed Lotfy 22P0251

Jana Sameh 22P0237

Yousif Salah 22P0232

Ahmed Al-amin`22P0137

Table of Contents

Table of Figures

1. Introduction:

A lexical analyzer, or lexer, is the first phase of a compiler or interpreter. Its main role is to read the source code and convert it into a stream of tokens, which are the meaningful building blocks of a programming language. These tokens typically include identifiers, keywords, operators, literals, and punctuation symbols.

In this project, we focus on developing a lexical analyzer for Python using C++. Python, being a dynamically typed and indentation-sensitive language, poses unique challenges for lexical analysis. While it's usually interpreted using its native parser, building a lexer in C++ provides deep insight into the internal workings of interpreters and how language syntax is processed at a lower level, this lexer scans Python source code, identifies valid tokens, and stores them in a structured format such as a symbol table. The symbol table keeps track of identifiers along with their types and values, which is essential for later phases like parsing and semantic analysis.

By implementing the lexical analyzer in C++, we leverage its strong performance and memory control capabilities, making it ideal for building fast and efficient analysis tools.

2. Project Structure and Files:

The Lexer implementation is organized across four key files:

- **Token.hpp**: Defines the Token structure, TokenType and TokenCategory enumerations, and utility functions for token categorization and string conversion.
- **Lexer.hpp**: Declares the Lexer class, including its public and private members for token generation, symbol table management, and type inference.
- **Lexer.cpp**: Implements the Lexer class, handling tokenization logic, indentation, and type inference for identifiers.
- **main .cpp**: Provides the entry point, reading input from a file, invoking the Lexer, and displaying the token stream and symbol table.

3. Token Design and Categorization:

The Token.hpp file defines the foundation of the Lexer's output:

- **Token Structure**: Each Token contains a type (from TokenType), lexeme (the source text), line number, and category (from TokenCategory).
- **Token Types**: The TokenType enum includes Python-inspired tokens such as keywords (TK_IF, TK_DEF, TK_CLASS), types (TK_INT, TK_STR, TK_LIST), operators (TK_PLUS, TK_ASSIGN), punctuation (TK_LPAREN, TK_COLON), and special tokens (TK_IDENTIFIER, TK_NUMBER, TK_STRING, TK_EOF).
- **Token Categories**: The TokenCategory enum groups tokens into KEYWORD, IDENTIFIER, NUMBER, STRING, PUNCTUATION, OPERATOR, EOFILE, and UNKNOWN, aiding in processing and analysis.
- **Utility Functions**: tokenTypeToString converts TokenType to human-readable strings, and getTokenCategory maps TokenType to TokenCategory.

4. Lexer Class Design:

The Lexer class, declared in Lexer.hpp and implemented in Lexer.cpp, is the core of the tokenization process:

- **Constructor**: Initializes the Lexer with the input source code, sets up a keyword map, and prepares indentation tracking.
- **Public Interface**:
  - nextToken(): Generates the next token, handling indentation and storing tokens in a public tokens vector.
  - getSymbolTable(): Returns a symbol table mapping identifiers to inferred types.
  - processIdentifierTypes(): Infers types for identifiers based on assignments and type hints.
- **Private Members**: Include the input string, current position (pos), line number, keyword map, symbol table, indentation stack, and pending tokens for indentation handling.
- **Helper Methods**: Functions like skipWhitespaceAndComments, handleIdentifierOrKeyword, handleNumeric, handleString, and handleSymbol modularize token recognition logic.

```
class Lexer {
public:
    explicit Lexer(string input);
    Token nextToken(); // Generates tokens one by one
    const unordered_map<string, string>& getSymbolTable(); // Gets the table *after* processing
    void processIdentifierTypes(); // Processes the generated tokens list
    vector<Token> tokens; // Public vector to store generated token

    // getter for the symbol table
    const vector<Lexer_error>& getErrors() const;
    string panicRecovery();

    static bool isKnownSymbol(char c);

    void reportError(const string &message, const string &lexeme);

private:
    string input;
    size_t pos;
    int line;
    unordered_map<string, TokenType> keywords;
    unordered_map<string, string> symbolTable; // Internal symbol table: <name, inferred_type_string>

    // Indentation tracking
    vector<int> indentStack;
    int currentIndent;
    bool atLineStart;
    vector<Token> pendingTokens; // For storing DEDENT tokens
```

Figure 1

```
    // Helper methods
    bool isAtEnd() const;

    char getCurrentCharacter() const;

    char advanceToNextCharacter();

    bool matchAndAdvance(char expected);

    bool skipMultilineComment();

    void skipWhitespaceAndComments();

    void skipComment();
    void processIndentation();
    Token createToken(TokenType type, const string &text) const;

    // Token handling methods
    Token handleIdentifierOrKeyword();

    Token handleNumeric();

    Token handleString();

    Token handleSymbol();

    Token operatorToken(TokenType simpleType, TokenType assignType, char opChar);
```

Figure 2

```
    // Type inference methods (called by processIdentifierTypes)
    string inferType(size_t& index); // Main inference function
    string inferListType(size_t& index);
    string inferTupleType(size_t& index);
    string inferDictOrSetType(size_t& index);
    string combineTypes(const vector<string>& types); // Helper to combine element types
    // Removed internal tokenTypeToString, use the one from Token.hpp
    // Removed inferComplexTypeHint
};

#endif // LEXER_HPP
```

Figure 3

5. Tokenization Process:

The tokenization process, implemented in Lexer.cpp, is a sophisticated single-pass algorithm that transforms Python-like source code into a sequence of tokens. It handles a variety of token types, whitespace, comments, and edge cases, ensuring robust and accurate token generation. Below is a detailed breakdown:

- **Input Processing**:
  - The Lexer reads the entire source file into a string, allowing character-by-character analysis.
  - The pos variable tracks the current position, and line tracks the line number for accurate error reporting.
  - The process is driven by nextToken(), which advances through the input until the end (TK_EOF) or a token is produced.
  -

```
Token Lexer::nextToken() {
    // For debugging
    static int tokenCount = 0;
    tokenCount++;

    // If we have pending indentation tokens, return them first
    if (!pendingTokens.empty()) {
        Token token = pendingTokens.front();
        pendingTokens.erase(pendingTokens.begin());
        tokens.push_back(token);
        return token;
    }

    skipWhitespaceAndComments();

    // Re-check for pending tokens after processing indentation
    if (!pendingTokens.empty()) {
        Token token = pendingTokens.front();
        pendingTokens.erase(pendingTokens.begin());
        tokens.push_back(token);
        return token;
    }
```

Figure 4

```
    if (isAtEnd()) {
        // Before returning EOF, check if we need to emit DEDENT tokens
        if (!indentStack.empty()) {
            while (!indentStack.empty()) {
                currentIndent = indentStack.back();
                indentStack.pop_back();
                pendingTokens.push_back(createToken(TokenType::TK_DEDENT, "DEDENT"));
            }

            Token token = pendingTokens.front();
            pendingTokens.erase(pendingTokens.begin());
            tokens.push_back(token);
            return token;
        }

        // Add a newline before EOF if we're not already at the start of a line
        if (!atLineStart && currentIndent > 0) {
            atLineStart = true;

            // Generate DEDENT tokens to get back to level 0
            while (currentIndent > 0) {
                if (!indentStack.empty()) {
                    currentIndent = indentStack.back();
                    indentStack.pop_back();
                } else {
                    currentIndent = 0;
                }
                pendingTokens.push_back(createToken(TokenType::TK_DEDENT, "DEDENT"));
            }
```

Figure 5

```
        if (!pendingTokens.empty()) {
            Token token = pendingTokens.front();
            pendingTokens.erase(pendingTokens.begin());
            tokens.push_back(token);
            return token;
        }
    }
    if (tokens.empty() || tokens.back().type != TokenType::TK_EOF) {
        Token eofToken = createToken(TokenType::TK_EOF, "");
        tokens.push_back(eofToken);
        return eofToken;
    }
    return tokens.back(); // Return existing EOF
}

const char currentCharacter = getCurrentCharacter();
Token token;

// Check for comments again, in case skipWhitespaceAndComments missed it
// TODO: Re-check logic of skipWhitespaceAndComments, we might have to return next token every time
if (currentCharacter == '#') {
    skipComment();
    return nextToken();
}

if (isalpha(currentCharacter) || currentCharacter == '_') {
    token = handleIdentifierOrKeyword();
} else if (isdigit(currentCharacter)) {
    token = handleNumeric();
```

Figure 6

```
    } else if (currentCharacter == '"' || currentCharacter == '\'') {
        token = handleString();
    } else {
        token = handleSymbol();
    }

    if (token.type != TokenType::TK_EOF) {
        tokens.push_back(token);
    }
    return token;
}

// --- Helper functions (isAtEnd, getCurrentCharacter, etc.) ---
bool Lexer::isAtEnd() const {
    return pos >= input.size();
}

char Lexer::getCurrentCharacter() const {
    return isAtEnd() ? '\0' : input[pos];
}

char Lexer::advanceToNextCharacter() {
    if (!isAtEnd()) {
        const char c = input[pos];
        pos++;
        return c;
    }
    return '\0';
}
```

Figure 7

- **Token Recognition Strategies**:
  - **Identifiers and Keywords**:
    - Handled by handleIdentifierOrKeyword, which collects alphanumeric characters and underscores starting with a letter or underscore.
    - Checks against a keywords map (populated in the constructor) to distinguish keywords (e.g., if, def, str) from identifiers (e.g., my_variable).
    - Example: def produces TK_DEF, while my_func produces TK_IDENTIFIER.
    - Edge Case: Keywords like False, True, and None are treated as literals with specific token types (TK_FALSE, TK_TRUE, TK_NONE).

```
Token Lexer::handleIdentifierOrKeyword() {
    const size_t start = pos;
    while (!isAtEnd() && (isalnum(getCurrentCharacter()) || getCurrentCharacter() == '_')) {
        advanceToNextCharacter();
    }
    const string text = input.substr(start, pos - start);

    // Check if it's a keyword (including type keywords)
    auto keyword_it = keywords.find(text);
    if (keyword_it != keywords.end()) {
        return createToken(keyword_it->second, text); // Return specific keyword/type token
    } else {
        // It's an identifier
        // Add it to the symbol table
        if (text.size() > 79) {
            reportError("Identifier name is too long", text);
            return createToken(TokenType::TK_UNKNOWN, text);
        }
        // Create an identifier token
        return createToken(TokenType::TK_IDENTIFIER, text);
    }
}
```

Figure 8

- **Numbers**:
  - Managed by handleNumeric, which recognizes integers (e.g., 42), floats (e.g., 3.14, 1e-10), and complex numbers (e.g., 2+3j).
  - Uses a state-machine-like approach to parse digits, decimal points, exponents (e/E), and the j suffix for complex numbers.
  - Distinguishes types within TK_NUMBER for integers and floats, and uses TK_COMPLEX for numbers ending in j.
  - Edge Case: Ensures a dot followed by a non-digit (e.g., 42.) is not consumed as a float, leaving the dot for subsequent tokenization.

```
Token Lexer::handleNumeric() {
    const size_t start = pos;
    bool isFloat = false;
    while (!isAtEnd() && isdigit(getCurrentCharacter())) {
        advanceToNextCharacter();
    }

    // Handle floating point
    if (!isAtEnd() && getCurrentCharacter() == '.') {
        if (pos + 1 < input.size() && isdigit(input[pos + 1])) {
            isFloat = true;
            advanceToNextCharacter(); // Consume '.'
            while (!isAtEnd() && isdigit(getCurrentCharacter())) {
                advanceToNextCharacter();
            }
        }
        // Else: it's an integer followed by '.', don't consume '.'
    }

    // Handle scientific notation
    if (!isAtEnd() && (getCurrentCharacter() == 'e' || getCurrentCharacter() == 'E')) {
        if (pos + 1 < input.size()) {
            char nextChar = input[pos+1];
            if (isdigit(nextChar) || ((nextChar == '+' || nextChar == '-') && pos + 2 < input.size() && isdigit(input[pos+2]))) {
                isFloat = true; // Scientific notation implies float
                advanceToNextCharacter(); // Consume 'e' or 'E'
                if (input[pos] == '+' || input[pos] == '-') {
                    advanceToNextCharacter(); // Consume sign
                }
                while (!isAtEnd() && isdigit(getCurrentCharacter())) {
                    advanceToNextCharacter();
```

Figure 9

```
                while (!isAtEnd() && isdigit(getCurrentCharacter())) {
                    advanceToNextCharacter();
                }
            }
        }
    }

    // Handle complex numbers AFTER potential float part
    if (!isAtEnd() && getCurrentCharacter() == 'j') {
        advanceToNextCharacter(); // Consume 'j'
        const string text = input.substr(start, pos - start);
        return createToken(TokenType::TK_COMPLEX, text); // Return specific complex token
    }

    // If not complex, return TK_NUMBER for both int and float
    const string text = input.substr(start, pos - start);
    // Although we detected float, the required TokenType is TK_NUMBER
    return createToken(TokenType::TK_NUMBER, text);
}
```

Figure 10

- **Strings**:
  - Processed by handleString, which handles single-quoted (') and double-quoted (") strings, including byte literals (e.g., b"data").
  - Supports escape sequences (e.g., \", \n) by skipping the escaped character.
  - Returns TK_STRING for regular strings and TK_BYTES for byte literals.

- Edge Case: Detects unterminated strings, logging an error and marking them as TK_UNKNOWN.

```
Token Lexer::handleString() {
    bool isBytes = false;
    size_t prefix_len = 0;
    if (!isAtEnd() && (getCurrentCharacter() == 'b' || getCurrentCharacter() == 'B')) {
        if (pos + 1 < input.size() && (input[pos+1] == '\'' || input[pos+1] == '"')) {
            isBytes = true;
            advanceToNextCharacter(); // Consume 'b' or 'B'
            prefix_len = 1;
        }
    }
    // Could add 'r', 'f', 'u' handling here if needed, but they usually affect parsing/value, not base type

    const char quote = getCurrentCharacter();
    advanceToNextCharacter();
    const size_t start = pos;
```

Figure 11

```
    while (!isAtEnd()) {
        const char c = getCurrentCharacter();

        if (c == '\n') {
            reportError("Unterminated string literal", input.substr(start - 1, pos - start + 1));
            return createToken(TokenType::TK_UNKNOWN, input.substr(start - 1, pos - start + 1));
        }

        if (c == quote) {
            advanceToNextCharacter(); // consume closing quote
            return createToken(isBytes ? TokenType::TK_BYTES : TokenType::TK_STRING, input.substr(start, pos - start - 1));
        }

        if (c == '\\' && pos + 1 < input.size()) {
            advanceToNextCharacter(); // skip the backslash
        }

        advanceToNextCharacter();
    }
    reportError("Unterminated string literal", input.substr(start - 1, pos - start + 1));
    return createToken(TokenType::TK_UNKNOWN, input.substr(start - 1, pos - start + 1));
}
```

Figure 12

- **Symbols (Operators and Punctuation)**:
  - Handled by handleSymbol, which processes single- and multi-character operators (e.g., +, +=, **=) and punctuation (e.g., (, :).

```
Token Lexer::handleSymbol() {
    const char currentCharacter = getCurrentCharacter();
    switch (currentCharacter) {
        // Single character punctuation
        case '(': advanceToNextCharacter(); return createToken(TokenType::TK_LPAREN, "(");
        case ')': advanceToNextCharacter(); return createToken(TokenType::TK_RPAREN, ")");
        case '[': advanceToNextCharacter(); return createToken(TokenType::TK_LBRACKET, "[");
        case ']': advanceToNextCharacter(); return createToken(TokenType::TK_RBRACKET, "]");
        case '{': advanceToNextCharacter(); return createToken(TokenType::TK_LBRACE, "{");
        case '}': advanceToNextCharacter(); return createToken(TokenType::TK_RBRACE, "}");
        case ',': advanceToNextCharacter(); return createToken(TokenType::TK_COMMA, ",");
        case ';': advanceToNextCharacter(); return createToken(TokenType::TK_SEMICOLON, ";");
        case '.': advanceToNextCharacter(); return createToken(TokenType::TK_PERIOD, ".");
        case '~': advanceToNextCharacter(); return createToken(TokenType::TK_BIT_NOT, "~");

            // Potential multi-char operators/punctuation
        case ':':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_WALNUT, ":="); // :=
            }
            return createToken(TokenType::TK_COLON, ":");
        case '-':
            advanceToNextCharacter();
            if (matchAndAdvance('>')) {
                return createToken(TokenType::TK_FUNC_RETURN_TYPE, "->"); // ->
            }
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_MINUS_ASSIGN, "-="); // -=
            }
            return createToken(TokenType::TK_MINUS, "-");
```

Figure 13

```
        case '+': return operatorToken(TokenType::TK_PLUS, TokenType::TK_PLUS_ASSIGN, '+'); // + or +=
        case '*':
            advanceToNextCharacter();
            if (matchAndAdvance('*')) {
                if (matchAndAdvance('=')) {
                    return createToken(TokenType::TK_POWER_ASSIGN, "**="); // **=
                }
                return createToken(TokenType::TK_POWER, "**"); // **
            }
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_MULTIPLY_ASSIGN, "*="); // *=
            }
            return createToken(TokenType::TK_MULTIPLY, "*"); // *
        case '/':
            advanceToNextCharacter();
            if (matchAndAdvance('/')) {
                if (matchAndAdvance('=')) {
                    return createToken(TokenType::TK_FLOORDIV_ASSIGN, "//="); // //=
                }
                return createToken(TokenType::TK_FLOORDIV, "//"); // //
            }
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_DIVIDE_ASSIGN, "/="); // /=
            }
            return createToken(TokenType::TK_DIVIDE, "/"); // /
        case '%': return operatorToken(TokenType::TK_MOD, TokenType::TK_MOD_ASSIGN, '%'); // % or %=
        case '@':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                // Use TK_IMATMUL for @= as defined in the provided Token.hpp
```

-

Figure 14

```
                return createToken(TokenType::TK_IMATMUL, "@=");
            }
            return createToken(TokenType::TK_MATMUL, "@"); // @
        case '&': return operatorToken(TokenType::TK_BIT_AND, TokenType::TK_BIT_AND_ASSIGN, '&'); // & or &=
        case '|': return operatorToken(TokenType::TK_BIT_OR, TokenType::TK_BIT_OR_ASSIGN, '|'); // | or |=
        case '^': return operatorToken(TokenType::TK_BIT_XOR, TokenType::TK_BIT_XOR_ASSIGN, '^'); // ^ or ^=
        case '=':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_EQUAL, "=="); // ==
            }
            return createToken(TokenType::TK_ASSIGN, "="); // =
        case '!':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_NOT_EQUAL, "!="); // !=
            }
            // '!' alone is not a standard Python operator
            return createToken(TokenType::TK_UNKNOWN, "!");
        case '>':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_GREATER_EQUAL, ">="); // >=
            }
            if (matchAndAdvance('>')) {
                if (matchAndAdvance('=')) {
                    return createToken(TokenType::TK_BIT_RIGHT_SHIFT_ASSIGN, ">>="); // >>=
                }
                return createToken(TokenType::TK_BIT_RIGHT_SHIFT, ">>"); // >>
            }
```

-

Figure 15

```
            return createToken(TokenType::TK_GREATER, ">"); // >
        case '<':
            advanceToNextCharacter();
            if (matchAndAdvance('=')) {
                return createToken(TokenType::TK_LESS_EQUAL, "<="); // <=
            }
            if (matchAndAdvance('<')) {
                if (matchAndAdvance('=')) {
                    return createToken(TokenType::TK_BIT_LEFT_SHIFT_ASSIGN, "<<="); // <<=
                }
                return createToken(TokenType::TK_BIT_LEFT_SHIFT, "<<"); // <<
            }
            return createToken(TokenType::TK_LESS, "<"); // <

        default:
            // Unknown single character
            advanceToNextCharacter();
            string unknown  = panicRecovery();
            return createToken(TokenType::TK_UNKNOWN, unknown);
    }
}
```

Figure 16

- Uses matchAndAdvance to check for subsequent characters in multi-character operators (e.g., := for TK_WALNUT, -> for TK_FUNC_RETURN_TYPE).

```
bool Lexer::matchAndAdvance(const char expected) {
    if (isAtEnd() || input[pos] != expected)
        return false;
    pos++;
    return true;
}
```

Figure 17

- Example: >= produces TK_GREATER_EQUAL, while > alone produces TK_GREATER.
- Edge Case: Non-standard characters (e.g., ! alone) are marked as TK_UNKNOWN.

- **Whitespace and Comment Handling**:
    - The skipWhitespaceAndComments function skips spaces, tabs, newlines, and carriage returns, updating the line count for newlines.

```
void Lexer::skipWhitespaceAndComments() {
    while (!isAtEnd()) {
        char c = getCurrentCharacter();

        if (c == ' ' || c == '\t') {
            // Check if we're at the start of a line (for indentation)
            if (atLineStart) {
                processIndentation();
                break;
            }
            // Else skip whitespace in the middle of a line
            advanceToNextCharacter();
        } else if (c == '\n') {
            line++;
            advanceToNextCharacter();
            atLineStart = true; // Mark that we're at the start of a new line
        } else if (c == '\r') {
            advanceToNextCharacter();
        }
        else if (c == '#') {
            skipComment();
        }
        else if (c == '"' || c == '\'') {
            if (!skipMultilineComment())
                break;
        } else {
            // If we're at the start of a line with non-whitespace, process for indentation
            if (atLineStart) {
                processIndentation();
            }
            break;
```

Figure 18

- Comments (starting with #) are skipped until the end of the line using skipComment.

```
void Lexer::skipComment() {
    while (!isAtEnd() && getCurrentCharacter() != '\n') {
        advanceToNextCharacter();
    }
    // After a comment, check for updating new line
    if (!isAtEnd() && getCurrentCharacter() == '\n') {
        line++;
        advanceToNextCharacter();
        atLineStart = true;
    }
}
```

Figure 19

- Whitespace at the start of a line triggers processIndentation to handle Python-style indentation (detailed in Section 6).
- Edge Case: Ensures comments at the end of the file don't cause premature termination by checking for newlines.

- **Error Handling**:
  - Unknown characters are assigned TK_UNKNOWN with their lexeme, allowing the Lexer to continue processing.
  - Unterminated strings trigger a console error with the starting line and partial lexeme.
  - Future Improvement: Could add detailed error tokens or throw exceptions for stricter error handling.
- **Efficiency Considerations**:
  - Uses string substrings sparingly, leveraging pos to track boundaries.
  - Minimizes memory allocation by reusing pendingTokens for indentation and storing tokens in a vector.
  - Avoids redundant checks by processing whitespace and comments before token recognition.

6. Indentation Handling:

A standout feature is the Lexer's support for Python-style indentation:

- **Indentation Stack**: Tracks indentation levels using a vector<int> (indentStack) and a currentIndent counter.
- **Process Indentation**: At the start of each line, counts spaces or tabs (treating tabs as 8 spaces) and compares with the current indentation level.
- **Indent/Dedent Tokens**: Generates TK_INDENT for increased indentation and TK_DEDENT for decreased indentation, stored in pendingTokens for correct sequencing.
- **Consistency Checks**: Ensures indentation aligns with previous levels, with potential for future error handling on inconsistencies.

```
void Lexer::processIndentation() {
    int spaces = 0;

    // Count spaces or tabs at the beginning of the line
    while (!isAtEnd() && (getCurrentCharacter() == ' ' || getCurrentCharacter() == '\t')) {
        const char c = getCurrentCharacter();
        spaces += (c == '\t') ? 8 : 1;  // A tab is equivalent to 8 spaces in Python
        advanceToNextCharacter();
    }

    // If a line is empty or a comment, ignore indentation
    if (isAtEnd() || getCurrentCharacter() == '\n' || getCurrentCharacter() == '#') {
        return;
    }

    atLineStart = false;
```

-

```
    atLineStart = false;

    // Compare with the current indentation level
    if (spaces > currentIndent) {
        // Indent
        indentStack.push_back(currentIndent);
        currentIndent = spaces;
        pendingTokens.push_back(createToken(TokenType::TK_INDENT, "INDENT"));
    } else if (spaces < currentIndent) {
        // Dedent
        while (!indentStack.empty() && spaces < currentIndent) {
            currentIndent = indentStack.back();
            indentStack.pop_back();
            pendingTokens.push_back(createToken(TokenType::TK_DEDENT, "DEDENT"));
        }

        // Ensure indentation is consistent
        if (spaces != currentIndent) {
            // TODO: handle inconsistent indentation (error handling)
            // For now we just adjust to the current indentation
            currentIndent = spaces;
        }
    }
}
```

7. Symbol Table and Type Inference:

The Lexer's type inference system, implemented in processIdentifierTypes and related functions, builds a symbol table mapping identifiers to their inferred types. This feature enhances the Lexer's utility for static analysis, IDE support, or compiler front-ends. Below is a detailed explanation:

- **Symbol Table Structure**:
    - An unordered_map<string, string> (symbolTable) stores identifiers (e.g., x, my_list) and their types (e.g., int, list[str]).
    - Types can be primitive (str, int), complex (list[int], dict[str, float]), custom (e.g., MyClass), or special (unknown, Any, function).
    - The table is populated after tokenization by processIdentifierTypes, ensuring all tokens are available for context.
- **ProcessIdentifierTypes() implementation**

```
void Lexer::processIdentifierTypes() {
    symbolTable.clear(); // Start fresh
    string currentClass; // Track current class context for 'self'

    size_t i = 0;
    while (i < tokens.size() && tokens[i].type != TokenType::TK_EOF) {
        Token currentToken = tokens[i];

        // --- Class Definition ---
        if (currentToken.type == TokenType::TK_CLASS && i + 1 < tokens.size() && tokens[i + 1].type == TokenType::TK_IDENTIFIER) {
            currentClass = tokens[i + 1].lexeme;
            symbolTable[currentClass] = "type"; // Class name represents a type
            i += 2; // Skip 'class' and identifier
            // Basic skipping of potential inheritance (...) and ':'
            if (i < tokens.size() && tokens[i].type == TokenType::TK_LPAREN) {
                int paren_depth = 1; i++;
                while(i < tokens.size() && paren_depth > 0) {
                    if (tokens[i].type == TokenType::TK_LPAREN) paren_depth++;
                    else if (tokens[i].type == TokenType::TK_RPAREN) paren_depth--;
                    i++;
                }
            }
            if (i < tokens.size() && tokens[i].type == TokenType::TK_COLON) i++;
            continue;
        }
```

- ```
  if (currentToken.type == TokenType::TK_DEF && i + 1 < tokens.size() && tokens[i + 1].type == TokenType::TK_IDENTIFIER) {
      string funcName = tokens[i+1].lexeme;
      symbolTable[funcName] = "function"; // Mark the function name
      i += 2; // Skip 'def' and funcName
      if (i < tokens.size() && tokens[i].type == TokenType::TK_LPAREN) {
          i++; // Skip '('
          bool firstParam = true;
          while(i < tokens.size() && tokens[i].type != TokenType::TK_RPAREN) {
              if (tokens[i].type == TokenType::TK_IDENTIFIER) {
                  string paramName = tokens[i].lexeme;
                  if (firstParam && !currentClass.empty() && paramName == "self") {
                      symbolTable["self"] = currentClass; // Infer 'self' type
                  } else if (!symbolTable.count(paramName)) { // Don't overwrite 'self'
                      symbolTable[paramName] = "unknown"; // Default param type
                  }
                  i++; // Skip identifier
                  // Basic type hint check (simple type keyword or identifier)
                  if (i < tokens.size() && tokens[i].type == TokenType::TK_COLON) {
                      i++; // Skip ':'
                      if(i < tokens.size()) {
                          TokenType hintType = tokens[i].type;
                          // Check if it's a type keyword defined in Token.hpp
                          if (hintType >= TokenType::TK_STR && hintType <= TokenType::TK_NONETYPE) {
                              symbolTable[paramName] = tokens[i].lexeme; // Use 'int', 'str', etc.
                              i++;
                          } else if (hintType == TokenType::TK_IDENTIFIER) { // Could be custom class
                              symbolTable[paramName] = tokens[i].lexeme;
                              i++;
  ```
  ```

```
                // Basic default value check (just to advance index)
                if (i < tokens.size() && tokens[i].type == TokenType::TK_ASSIGN) {
                    i++; // Skip '='
                    if (i < tokens.size()) {
                        size_t valIdx = i;
                        string inferredDefault = inferType(valIdx); // Infer type of default
                        i = valIdx; // Advance main index past default value
                        // Optionally update type if it was unknown
                        if (symbolTable[paramName] == "unknown" && inferredDefault != "unknown") {
                            symbolTable[paramName] = inferredDefault;
                        }
                    }
                }
            } else { i++; } // Skip other tokens like ',', '*', etc.

            firstParam = false;
            if (i < tokens.size() && tokens[i].type == TokenType::TK_COMMA) { i++; firstParam = true; }
        }
        if (i < tokens.size() && tokens[i].type == TokenType::TK_RPAREN) i++; // Skip ')'
        // Skip return type hint '->' and the type itself
        if (i < tokens.size() && tokens[i].type == TokenType::TK_FUNC_RETURN_TYPE) {
            i++; // Skip '->'
            while(i < tokens.size() && tokens[i].type != TokenType::TK_COLON) { i++; } // Skip until ':'
        }
        if (i < tokens.size() && tokens[i].type == TokenType::TK_COLON) i++; // Skip ':'
    }
    continue;
}
```

```
// --- Assignment: identifier = value ---
if (currentToken.type == TokenType::TK_IDENTIFIER && i + 1 < tokens.size() && tokens[i + 1].type == TokenType::TK_ASSIGN)
{
    string identifier = currentToken.lexeme;
    // Avoid overwriting 'self' type if already set
    if (identifier == "self" && symbolTable.count("self") && symbolTable["self"] != "unknown") {
        i = i + 2; // Skip identifier and '='
        if (i < tokens.size()) { inferType(i); } // Skip value by inferring its type to advance index
        continue;
    }

    size_t valueIndex = i + 2; // Index of token after '='
    if (valueIndex < tokens.size()) {
        string inferred = inferType(valueIndex); // Infer type, advances valueIndex
        symbolTable[identifier] = inferred;
        i = valueIndex; // Update main loop index
        continue; // Skip normal i++
    } else {
        i += 2; // Skip identifier and '=', value missing
        continue;
    }
}
```

```
    if (currentToken.type == TokenType::TK_IDENTIFIER && i + 1 < tokens.size() && tokens[i + 1].type == TokenType::TK_COLON)
    {
        string identifier = currentToken.lexeme;
        size_t typeIndex = i + 2;
        string typeName = "unknown";

        if (typeIndex < tokens.size()) {
            Token typeToken = tokens[typeIndex];
            // Check if it's a type keyword or identifier hint
            if (typeToken.type >= TokenType::TK_STR && typeToken.type <= TokenType::TK_NONETYPE) {
                typeName = typeToken.lexeme;
                typeIndex++;
            } else if (typeToken.type == TokenType::TK_IDENTIFIER) {
                typeName = typeToken.lexeme; // Custom type
                typeIndex++;
            } else {
                // Skip complex hints like list[int] - just advance past the type part
                // Basic skip: assume type hint ends before '=' or newline (simplification)
                while (typeIndex < tokens.size() && tokens[typeIndex].type != TokenType::TK_ASSIGN && tokens[typeIndex].type != TokenType::TK_SEMICOLON /* add oth
                    if (tokens[typeIndex].line != currentToken.line) break; // Stop at newline
                    typeIndex++;
                }
                typeName = "complex_hint"; // Mark as complex/unparsed
            }
```

```
            // Update symbol table if not already known
            if (!symbolTable.count(identifier) || symbolTable[identifier] == "unknown") {
                symbolTable[identifier] = typeName;
            }

            i = typeIndex; // Update main loop index past the type hint

            // Check for optional assignment after hint
            if (i < tokens.size() && tokens[i].type == TokenType::TK_ASSIGN) {
                i++; // Skip '='
                if (i < tokens.size()) {
                    inferType(i); // Infer value type mainly to advance index correctly
                    continue; // Skip normal i++
                }
            } else {
                // No assignment after hint, just continue
                continue; // Skip normal i++
            }
        } else {
            i += 2; // Skip identifier and ':', type missing
            continue;
        }
    }

    // If none of the above patterns matched, just move to the next token
    i++;
}
```

- **Type Inference Mechanisms**:
    - **Assignments (**identifier = value**)**:
        - Detects patterns like x = 42, inferring int based on the value's token type (TK_NUMBER).
        - Uses inferType to analyze the value, advancing the token index to skip the entire expression.
        - Example: y = [1, 2] infers list[int] by analyzing the list literal.

        - Edge Case: Skips assignments to self if its type is already known (e.g., in class methods).
    - **Type Hints (**identifier: type [= value]**):**
        - Recognizes explicit type annotations, such as x: str = "hello" → str.
        - Supports primitive type keywords (TK_STR, TK_INT, etc.) and identifiers (e.g., x: MyClass → MyClass).
        - For complex hints (e.g., list[int]), currently marks as complex_hint but could be extended with a type parser.
        - Example: z: dict infers dict unless overridden by an assignment.
    - **Class Definitions**:
        - Identifies class MyClass: and marks MyClass as type in the symbol table.
        - Tracks the current class context to infer self as the class type in methods (e.g., self → MyClass).
        - Skips inheritance clauses by advancing past parentheses, ensuring correct token progression.
    - **Function Definitions**:
        - Marks function names as function (e.g., def my_func(): → my_func: function).
        - Infers parameter types from type hints (e.g., def func(x: int) → x: int) or default values (e.g., x = 42 → int).
        - Handles self as the first parameter in methods, tying it to the enclosing class type.
        - Skips return type hints (-> type) to focus on parameter types, advancing past the function header.
    - **Literal Analysis**:
        - The inferType function maps tokens to types:
            - TK_NUMBER: Checks lexeme for . or e/E to distinguish float from int.
            - TK_STRING: Infers str; TK_BYTES: Infers bytes.
            - TK_TRUE/TK_FALSE: Infers bool; TK_NONE: Infers NoneType.
            - Collections (TK_LBRACKET, TK_LPAREN, TK_LBRACE) trigger specialized inference.
        - Edge Case: Identifiers without known types (e.g., function calls) default to unknown.
    - **InferType() implementation**

```
std::string Lexer::inferType(size_t& index) {
    if (index >= tokens.size() || tokens[index].type == TokenType::TK_EOF) {
        return "unknown";
    }

    Token& token = tokens[index];
    std::string inferred_type = "unknown";

    switch (token.type) {
        case TokenType::TK_NUMBER:
            // Check lexeme to differentiate int/float for TK_NUMBER
            if (token.lexeme.find('.') != std::string::npos ||
                token.lexeme.find('e') != std::string::npos ||
                token.lexeme.find('E') != std::string::npos) {
                inferred_type = "float";
            } else {
                inferred_type = "int";
            }
            index++;
            break;
        case TokenType::TK_COMPLEX: // Specific token for complex literals
            inferred_type = "complex";
            index++;
            break;
        case TokenType::TK_STRING: // Normal string literal
            inferred_type = "str";
            index++;
            break;
        case TokenType::TK_BYTES: // Bytes literal
            inferred_type = "bytes";
            index++;
```

```
            break;
        case TokenType::TK_TRUE:
        case TokenType::TK_FALSE:
            inferred_type = "bool";
            index++;
            break;
        case TokenType::TK_NONE:
            inferred_type = "NoneType";
            index++;
            break;
        case TokenType::TK_LBRACKET: // Start of list literal [...]
            inferred_type = inferListType(index); // Advances index past ']'
            break;
        case TokenType::TK_LPAREN: // Start of tuple literal (...)
            inferred_type = inferTupleType(index); // Advances index past ')'
            break;
        case TokenType::TK_LBRACE: // Start of dict/set literal {...}
            inferred_type = inferDictOrSetType(index); // Advances index past '}'
            break;
        case TokenType::TK_IDENTIFIER:
            // If it's a known variable, use its type. Otherwise, unknown.
            // Could be a function call too - difficult to know return type here.
            if (symbolTable.count(token.lexeme)) {
                inferred_type = symbolTable[token.lexeme];
            } else {
                inferred_type = "unknown"; // Treat as unknown or potential function call
            }
            index++; // Consume identifier
            // Basic handling for function call: skip (...)
```

```
            // Basic handling for function call: skip (...)
            if (index < tokens.size() && tokens[index].type == TokenType::TK_LPAREN) {
                int depth = 1; index++;
                while(index < tokens.size() && depth > 0) {
                    if(tokens[index].type == TokenType::TK_LPAREN) depth++;
                    else if(tokens[index].type == TokenType::TK_RPAREN) depth--;
                    index++;
                }
                // Type remains as initially inferred (e.g., 'function' or 'unknown')
            }
            break;

            // Type keywords used as values (e.g., x = int)
        case TokenType::TK_INT: case TokenType::TK_STR: case TokenType::TK_FLOAT: case TokenType::TK_BOOL:
        case TokenType::TK_LIST: case TokenType::TK_TUPLE: case TokenType::TK_DICT: case TokenType::TK_SET:
            // ... other type keywords
            inferred_type = "type"; // The value is a type object itself
            index++;
            break;

        default:
            // Other tokens (operators, punctuation, non-type keywords) don't represent simple data types
            inferred_type = "unknown";
            index++; // Consume the token
            break;
    }
    return inferred_type;
}
```

- **Complex Type Inference**:
    - **Lists (**inferListType**)**:
        - Processes [elem1, elem2, ...] by inferring each element's type and combining them.
        - Example: [1, 2, 3] → list[int]; [1, "a"] → list[Any].
        - Handles empty lists ([]) as list[Any] and validates commas for syntax.

```
// Infer list type: list[...]
std::string Lexer::inferListType(size_t& index) {
    index++; // Consume '['
    std::vector<std::string> elementTypes;
    bool firstElement = true;

    while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACKET) {
        if (!firstElement) {
            if (index < tokens.size() && tokens[index].type == TokenType::TK_COMMA) {
                index++; // Consume ','
                if (index >= tokens.size() || tokens[index].type == TokenType::TK_RBRACKET) break; // Trailing comma
            } else {
                cerr << "Syntax Error: Expected ',' or ']' in list at line " << (index < tokens.size() ? tokens[index].line : -1) << endl;
                while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACKET) { index++; }
                break;
            }
        }
        firstElement = false;

        if (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACKET) {
            elementTypes.push_back(inferType(index)); // Advances index past element
        } else if (index >= tokens.size()) {
            cerr << "Syntax Error: Unexpected end of input within list literal." << endl;
            break;
        }
    }
```

```
    if (index < tokens.size() && tokens[index].type == TokenType::TK_RBRACKET) {
        index++; // Consume ']'
    } else if (index >= tokens.size()){
        cerr << "Syntax Error: Unexpected end of input, expected ']' for list literal." << endl;
    } else {
        cerr << "Syntax Error: Expected ']' to close list at line " << tokens[index].line << endl;
    }

    return "list[" + combineTypes(elementTypes) + "]";
}
```

- **Tuples (**inferTupleType**)**:
  - Analyzes (elem1, elem2, ...) similarly, supporting single-element tuples with trailing commas (e.g., (42,) → tuple[int]).
  - Empty tuples (()) become tuple[].
  - Edge Case: Distinguishes (x) (parentheses) from (x,) (tuple) using comma detection.

```
// Infer tuple type: tuple(...)
std::string Lexer::inferTupleType(size_t& index) {
    index++; // Consume '('
    std::vector<std::string> elementTypes;
    bool firstElement = true;
    bool trailingComma = false; // Needed to distinguish (elem) from (elem,)

    while (index < tokens.size() && tokens[index].type != TokenType::TK_RPAREN) {
        trailingComma = false; // Reset before processing element or comma
        if (!firstElement) {
            if (index < tokens.size() && tokens[index].type == TokenType::TK_COMMA) {
                index++; // Consume ','
                trailingComma = true;
                if (index >= tokens.size() || tokens[index].type == TokenType::TK_RPAREN) break; // Trailing comma case
            } else {
                cerr << "Syntax Error: Expected ',' or ')' in tuple at line " << (index < tokens.size() ? tokens[index].line : -1) << endl;
                while (index < tokens.size() && tokens[index].type != TokenType::TK_RPAREN) { index++; }
                break;
            }
        }
        firstElement = false;

        if (index < tokens.size() && tokens[index].type != TokenType::TK_RPAREN) {
            elementTypes.push_back(inferType(index)); // Advances index past element
        } else if (index >= tokens.size()) {
            cerr << "Syntax Error: Unexpected end of input within tuple literal." << endl;
            break;
        }
    }
```

```
    if (index < tokens.size() && tokens[index].type == TokenType::TK_RPAREN) {
        index++; // Consume ')'
    } else if (index >= tokens.size()){
        cerr << "Syntax Error: Unexpected end of input, expected ')' for tuple literal." << endl;
    } else {
        cerr << "Syntax Error: Expected ')' to close tuple at line " << tokens[index].line << endl;
    }

    // Special case: single element tuple `(elem,)` needs the comma
    // If only one element was found AND a trailing comma was consumed right before ')', it's a tuple.
    // If only one element and NO trailing comma, it's just parentheses for precedence, not a tuple.
    // However, since we are called ONLY when a '(' literal is found, we treat `(x)` as `tuple[type(x)]` here.
    // Python's type system might disagree, but for literal inference, this seems reasonable.
    // Let `combineTypes` handle the actual content representation.
    if (elementTypes.empty()) return "tuple[]"; // Handle () empty tuple
    return "tuple[" + combineTypes(elementTypes) + "]";


}
```

- **Dictionaries and Sets (**inferDictOrSetType**)**:
  - Determines {key: value, ...} (dict) vs {elem, ...} (set) by checking for TK_COLON.
  - For dictionaries, infers key and value types (e.g., {1: "a"} → dict[int, str]).
  - For sets, infers element types (e.g., {1, 2} → set[int]).
  - Empty {} defaults to dict[Any, Any], per Python's convention.
  - Edge Case: Detects syntax errors like missing colons or mixed dict/set syntax.

```
// Infer dict or set type: {...}
std::string Lexer::inferDictOrSetType(size_t& index) {
    index++; // Consume '{'
    std::vector<std::string> keyTypes, valueTypes, elementTypes;
    bool isDict = false, isSet = false, first = true, determined = false;

    // Handle empty literal {} -> dict
    if (index < tokens.size() && tokens[index].type == TokenType::TK_RBRACE) {
        index++; // Consume '}'
        return "dict[Any, Any]"; // Python defaults {} to empty dict
    }

    while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACE) {
        if (!first) {
            if (index < tokens.size() && tokens[index].type == TokenType::TK_COMMA) {
                index++; // Consume ','
                if (index >= tokens.size() || tokens[index].type == TokenType::TK_RBRACE) break; // Trailing comma
            } else {
                cerr << "Syntax Error: Expected ',' or '}' in dict/set at line " << (index < tokens.size() ? tokens[index].line : -1) << endl;
                while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACE) { index++; }
                break;
            }
        }
        first = false;
```

```
    // Peek ahead for ':' after the first element/key to determine dict vs set
    size_t peekIndex = index;
    if (peekIndex >= tokens.size() || tokens[peekIndex].type == TokenType::TK_RBRACE) break; // Empty after comma

    string tempType = inferType(peekIndex); // Infer type without advancing main index
    bool colonFollows = (peekIndex < tokens.size() && tokens[peekIndex].type == TokenType::TK_COLON);

    if (!determined) {
        isDict = colonFollows;
        isSet = !colonFollows;
        determined = true;
    } else { // Check consistency
        if ((isDict && !colonFollows) || (isSet && colonFollows)) {
            cerr << "Syntax Error: Mixing dict key-value pairs and set elements at line " << tokens[index].line << endl;
            while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACE) { index++; }
            break;
        }
    }
```

```
        // Process based on determined type
        if (isDict) {
            keyTypes.push_back(inferType(index)); // Consume key, advance main index
            if (index < tokens.size() && tokens[index].type == TokenType::TK_COLON) {
                index++; // Consume ':'
                if (index >= tokens.size() || tokens[index].type == TokenType::TK_RBRACE || tokens[index].type == TokenType::TK_COMMA ) {
                    cerr << "Syntax Error: Expected value after ':' in dict at line " << (index > 0 ? tokens[index-1].line : 0) << endl;
                    while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACE) { index++; }
                    break;
                }
                valueTypes.push_back(inferType(index)); // Consume value, advance main index
            } else {
                cerr << "Syntax Error: Expected ':' after key in dict at line " << (index > 0 ? tokens[index-1].line : 0) << endl;
                while (index < tokens.size() && tokens[index].type != TokenType::TK_RBRACE) { index++; }
                break;
            }
        } else { // isSet
            elementTypes.push_back(inferType(index)); // Consume element, advance main index
        }
    }

    if (index < tokens.size() && tokens[index].type == TokenType::TK_RBRACE) {
        index++; // Consume '}'
    } else if (index >= tokens.size()){
        cerr << "Syntax Error: Unexpected end of input, expected '}' for dict/set literal." << endl;
    } else {
        cerr << "Syntax Error: Expected '}' to close dict/set at line " << tokens[index].line << endl;
    }
```

```
    if (isDict) {
        return "dict[" + combineTypes(keyTypes) + ", " + combineTypes(valueTypes) + "]";
    } else if (isSet) {
        return "set[" + combineTypes(elementTypes) + "]";
    } else {
        // This case should only be hit for empty {} which is handled above, or after errors.
        return "dict[Any, Any]"; // Default to dict if unsure/error
    }
}
```

- **Type Combination (**combineTypes**):**
  - Unifies multiple element types in collections (e.g., int, str → Any).
  - Returns a single type if all elements match (e.g., [int, int] → int).
  - Uses a set to deduplicate types and handles special cases like unknown or function by returning Any.
  - Future Improvement: Could support Union[int, str] for precise typing.
  -

```
// Combine types found within a collection (list, tuple, set, dict key/value)
std::string Lexer::combineTypes(const std::vector<std::string>& types) {
    if (types.empty()) return "Any"; // Represent empty collection or unknown element type

    std::set<std::string> unique_types(types.begin(), types.end());

    // If any element's type is unknown or complex, the combined type is uncertain
    if (unique_types.count("unknown") || unique_types.count("complex_hint") || unique_types.count("function")) {
        return "Any"; // Or "unknown" depending on desired strictness
    }
    if (unique_types.count("Any")) {
        return "Any"; // Propagate Any if present
    }

    if (unique_types.size() == 1) {
        return *unique_types.begin(); // All elements have the same single type
    } else {
        // Multiple distinct known types found. Represent as "Any" for simplicity.
        // A more advanced system might use "Union[type1, type2]".
        return "Any";
    }
}
```

- **Error Handling and Robustness**:
    - Logs syntax errors for malformed collections (e.g., missing],), or }) to cerr, advancing the index to recover.
    - Defaults to unknown for unresolvable types, ensuring the Lexer continues processing.
    - Avoids overwriting known types (e.g., self) unless explicitly updated.
- **Context Awareness**:
    - Maintains context for class and function scopes, improving type accuracy for self and parameters.
    - Tracks token sequences to skip complex constructs (e.g., function call arguments) without full parsing.
    - Edge Case: Conservative inference for identifiers in expressions (e.g., function calls) avoids speculative typing.

8. Error Handling:

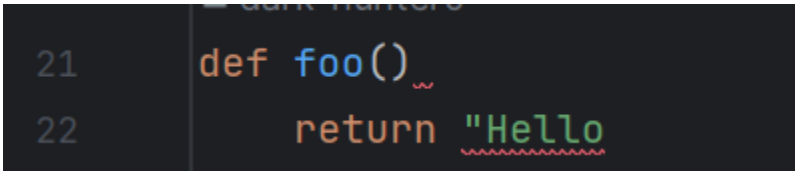
The lexical analyzer has been extended to detect and recover from a variety of **Python-specific lexical errors**. These are essential to prevent malformed tokens from affecting further stages of compilation. Each error type is described below along with how it is detected and managed.

8.1. Unterminated String Literals:

Description:
Occurs when a string literal (enclosed in ' or ") is not closed before the end of the line or input.

Handling:
The analyzer scans the string and checks if a matching closing quote is encountered. If a newline or end-of-file is reached first, it generates a TK_UNKNOWN token and logs the error with line information.

Recovery:
After reporting the error, the lexer resumes tokenizing from the next valid character.



```
Lexical Error at line 22: Unterminated string literal -> '"Hello'
```

Figure 19

```
    reportError( message: "Unterminated string literal", lexeme: input.substr( pos: start - 1, n: pos - start + 1));
    return createToken(TokenType::TK_UNKNOWN, text: input.substr( pos: start - 1, n: pos - start + 1));
}
```

Figure 20

8.2 Unknown Symbols:

Description:

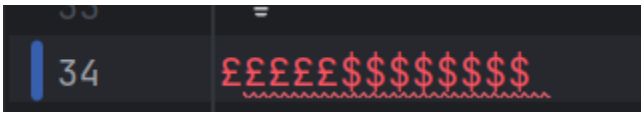
Characters not recognized in Python (e.g., @, $, ~ in invalid contexts).

Handling:

When the lexer encounters an unrecognized character, it logs a LexicalError with type InvalidCharacter and assigns a TK_UNKNOWN token.

Recovery (Panic Mode):

Instead of tokenizing each unrecognized character, the lexer enters **panic mode**, skipping all consecutive unknown symbols until it finds a valid one. This avoids token clustering and prevents unnecessary downstream errors.




```
Lexical Error at line 34: Unknown Symbols found -> 'ÚTÚTÚTÚTÚ$$$$$$$$'
```

Figure 21

```
//skips unknown symbols
string Lexer::panicRecovery() {
    string unknown;
    while (!isAtEnd()) {
        char c = getCurrentCharacter();

        // recovery points: whitespace, known starting characters
        if (isspace(c) || isalpha(c) || isdigit(c) || c == '_' || isKnownSymbol(c)) {
            break;
        }
        unknown.push_back(c);
        advanceToNextCharacter();
    }
    reportError( message: "Unknown Symbols found", lexeme: unknown);
    return unknown;
}
```

Figure 22

8.3 Unterminated triple quote comments:

Description:

In Python, triple-quoted strings (''' or """) can span multiple lines. If not properly closed, they are considered unterminated.

Handling:

The lexer identifies the opening """ and continues scanning until it encounters the closing """. If the end-of-file is reached first, it reports an UnterminatedTripleQuote error.

```
"""Testing quoted comments
multiple lines""
```

```
Lexical Error at line 36: Unterminated triple-quoted string -> '"""Testing quoted comments
multiple lines""

# '''hi lotfy
# nice to meet you
# '''

ТÚТÚТÚТÚТÚ$$$$$$$$
ikio = "2'
```

Figure5: unterminated tripleQuote error

```cpp
bool Lexer::skipMultilineComment() {
    const size_t start = pos;
    const char quoteChar = getCurrentCharacter(); // Store initial quote type

    // Check for triple quotes
    if (matchAndAdvance(quoteChar) && matchAndAdvance(quoteChar) && matchAndAdvance(quoteChar)) {
        while (!isAtEnd()) {
            if (getCurrentCharacter() == quoteChar &&
                pos + 2 < input.size() &&
                input[pos + 1] == quoteChar &&
                input[pos + 2] == quoteChar) {
                advanceToNextCharacter(); // first quote
                advanceToNextCharacter(); // second quote
                advanceToNextCharacter(); // third quote
                return true;
            }

            if (getCurrentCharacter() == '\n') {
                line++;
            }

            advanceToNextCharacter();
        }

        // If we reach here, the triple-quoted string was never closed
        const string unterminated = input.substr( pos: start,  n: pos - start);
        reportError( message: "Unterminated triple-quoted string",  lexeme: unterminated);
        return false;
    }

    pos = start; // rollback if not a triple quote
    return false;
}
```

Figure6: unterminated tripleQuote code

8.4 Overly long identifiers name :

Description:

Python by nature limits the length of identifiers to 79 for performance reasons.

Handling:

Identifiers exceeding the maximum allowed length are truncated, and a lexical error is reported. The token may still be created, but flagged as erroneous.

```
ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAFAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Lexical Error at line 23: Identifier name is too long -> 'ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAFAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

Figure7: overly long identifiers names example

```cpp
Token Lexer::handleIdentifierOrKeyword() {
    const size_t start = pos;
    while (!isAtEnd() && (isalnum( C: getCurrentCharacter()) || getCurrentCharacter() == '_')) {
        advanceToNextCharacter();
    }
    const string text = input.substr( pos: start,  n: pos - start);

    // Check if it's a keyword (including type keywords)
    auto keyword_it = keywords.find(text);
    if (keyword_it != keywords.end()) {
        return createToken(keyword_it->second, text); // Return specific keyword/type token
    } else {
        // It's an identifier
        // Add it to the symbol table
        if (text.size() > 79) {
            reportError( message: "Identifier name is too long",  lexeme: text);
            return createToken(TokenType::TK_UNKNOWN, text);
        }
        // Create an identifier token
        return createToken(TokenType::TK_IDENTIFIER, text);
    }
}
```

Figure8: overly long identifiers names code

8.5 Overly sized numeric literals:

Description:
Languages like C++ raise an error for integers that overflow the data type. However, in Python, integers can be arbitrarily large as they're promoted to long.

Handling:
Since Python does not have fixed integer overflow, this error is acknowledged in theory but not enforced. Any digit sequence is treated as valid, and no overflow error is raised.

8.6 General recovery strategy:

Panic Recovery:
Implemented to avoid cascading errors. When an unrecognized symbol is found, the lexer skips unknown characters until it reaches a recognizable boundary (whitespace, keyword, valid symbol, etc.).

Advantages:

- Prevents flooding the error report with repeated TK_UNKNOWN tokens.
- Allows the lexer to continue scanning the rest of the input meaningfully.

Summary

All common lexical errors in Python are now accounted for. Errors that require contextual understanding (e.g., indentation issues, syntax rules) are deferred to the **parser**, which handles grammar-level rules.

9. Main Program and Output:

The main (2).cpp file demonstrates the Lexer's usage and provides comprehensive output for debugging and analysis:

- **Input Reading**:
  - Reads a Python-like source file (test.py) into a string using ifstream and stringstream.
  - Checks for file opening failures, reporting an error to cerr and exiting gracefully if the file cannot be opened.
- **Token Generation**:
  - Creates a Lexer instance with the input string and invokes nextToken() in a loop until TK_EOF is reached.
  - All generated tokens are stored in the Lexer's public tokens vector for subsequent processing.
- **Output**:
  - **Raw Token Stream**:
    - Iterates over the tokens vector, printing each token's details.
    - Uses tokenTypeToString to display the token type (e.g., identifier, number, if).
    - Escapes newlines (\n) and tabs (\t) in the lexeme for cleaner output.
    - Displays the lexeme for categories IDENTIFIER, NUMBER, STRING, UNKNOWN, and EOF, as well as for OPERATOR and PUNCTUATION tokens.
    - Includes the line number for each token, aiding in source code navigation.
    - Example output format: <identifier, "my_var"> Line: 1.
  - **Symbol Table**:
    - Retrieves the symbol table via getSymbolTable().
    - Sorts identifiers alphabetically for consistent output using a vector<string> and std::sort.
    - Prints each identifier and its inferred type (e.g., my_var : int).
    - Displays (empty) if the symbol table is empty.
  - **Lexical Errors**:
    - Retrieves lexical errors via getErrors(), which returns a collection of error objects containing the line number, error message, and problematic lexeme.

        - Prints each error in the format: Lexical Error at line X: message -> 'lexeme'.
        - Examples include errors for unterminated strings or unrecognized characters, enhancing debugging capabilities.
- **Error Handling**:
    - Handles file I/O errors by checking the file stream's state.
    - Leverages the Lexer's error collection to report lexical issues, ensuring users are informed of tokenization problems.

10. GUI Implementation:

The GUI implementation of the Python lexical analyzer in pro-text-editor combines a feature-rich code editor with robust lexical analysis capabilities. The **CodeEditor**, **PythonHighlighter, SymbolTableDialog**, and **TokenSequenceDialog** work in tandem to provide real-time feedback, syntax highlighting, and detailed lexer outputs. The dark-themed interface, find/replace functionality, and seamless integration with the **Lexer** class ensure an intuitive and powerful tool for Python development and analysis.

The pro-text-editor application provides a robust GUI for editing and analyzing Python code, with a lexical analyzer integrated to tokenize input and generate symbol tables. Built using Qt in C++, the GUI enhances user interaction through a dark-themed interface, syntax highlighting, and dialogs for visualizing lexer outputs. The lexical analyzer, implemented in **Lexer.hpp** and **Lexer.cpp**, processes Python code to produce tokens and symbols, which are displayed via the **CodeEditor**, **SymbolTableDialog**, and **TokenSequenceDialog**. The **MainWindow** class orchestrates these components, ensuring seamless integration and responsive user feedback.

Key Files: main.cpp, mainwindow.hpp, mainwindow.cpp, Lexer.hpp, Lexer.cpp, codeeditor.hpp, codeeditor.cpp.

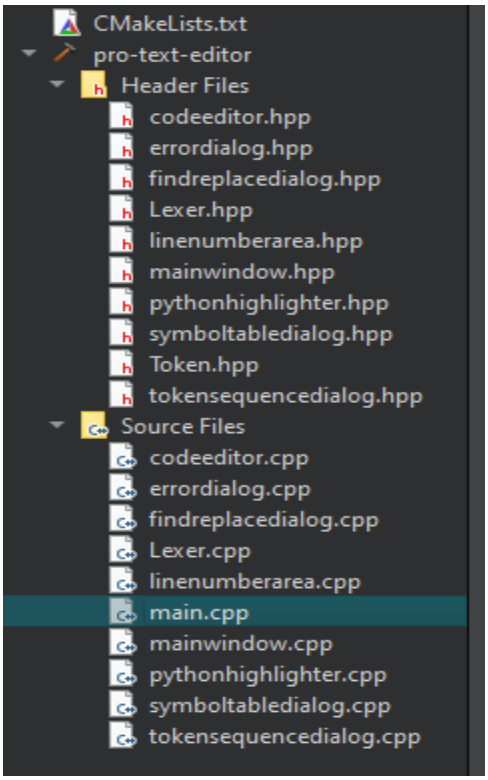


Figure 23

10.1 Code Editor Interface for Lexical Analysis:

The **CodeEditor** class, derived from **QPlainTextEdit,** serves as the central GUI component for editing Python code. It integrates with the lexical analyzer to provide real-time syntax highlighting and supports user interactions for lexer execution. Key features include:

- **Monospaced Font and Formatting**: Configures Consolas with a 4-space tab stop to align with Python's indentation rules (**codeeditor.cpp**).
- **Line Number Area**: Implements LineNumberArea for code navigation, synchronized with the editor's scroll and cursor (**linenumberarea.hpp, codeeditor.cpp**).
- **Event Handling**: Supports Python-specific editing with auto-indentation on Enter, tab-to-space conversion, and bracket matching (**codeeditor.cpp**).
- **Syntax Highlighting**: Uses PythonHighlighter to color-code tokens based on lexer-compatible categories (e.g., keywords, strings), enhancing readability (**pythonhighlighter.hpp, pythonhighlighter.cpp**).

The editor connects to the MainWindow for lexer actions, enabling users to trigger analysis via the "Run Lexer" menu option (**mainwindow.cpp**).

Key Files: codeeditor.hpp, codeeditor.cpp, linenumberarea.hpp, pythonhighlighter.hpp, pythonhighlighter.cpp.

10.2 Lexical Analyzer Implementation and Tokenization:

The **Lexer** class, defined in **Lexer.hpp** and implemented in **Lexer.cpp**, is the core of the lexical analysis system. It tokenizes Python code into a sequence of Token structs, categorized by **TokenType** (e.g., TK_KEYWORD, TK_IDENTIFIER) and **TokenCategory** (e.g., KEYWORD, IDENTIFIER). The lexer processes the editor's text, handling Python constructs like keywords, literals, and indentation. Key features include:

- **Keyword Recognition**: Identifies Python keywords (e.g., if, def, str) and type hints (e.g., int, list) using a keyword map (**Lexer.cpp**).
- **Literal Handling**: Processes numbers (integers, floats, complex), strings (single/double quotes, triple-quoted docstrings), and bytes literals (handleNumeric, handleString in **Lexer.cpp**).

- **Token Storage**: Stores tokens in a tokens vector, accessible for GUI display and symbol table generation (**Lexer.hpp**).
- **Error Handling**: Detects unknown tokens (TK_UNKNOWN), logging warnings for invalid input (**mainwindow.cpp**).

The **MainWindow::runLexer** method invokes the lexer, storing results in lastTokens for display (**mainwindow.cpp**).

Key Files: Lexer.hpp, Lexer.cpp, Token.hpp, mainwindow.cpp.

10.3 Symbol Table Generation and Visualization:

The lexical analyzer generates a symbol table mapping identifiers to inferred types, which is visualized via the **SymbolTableDialog**. The **Lexer::processIdentifierTypes** method analyzes tokens to infer types from assignments, type hints, and contexts (e.g., self in classes). Key features include:

- **Type Inference**: Supports Python's dynamic typing, inferring types like int, list, or custom classes based on assignments and hints (inferType, inferListType in **Lexer.cpp**).
- **Symbol Table Storage**: Stores results in an unordered_map<std::string, std::string> (**Lexer.hpp**), retrieved by **MainWindow** for display (**mainwindow.cpp**).
- **GUI Visualization**: The **SymbolTableDialog** presents the symbol table in a sortable table with columns for index, identifier, and data type. It uses NumericTableWidgetItem for numeric sorting and supports copying cell content via a context menu (**symboltabledialog.cpp**).
- **Styling**: Applies a dark theme with alternating row colors and custom scrollbars, matching the editor's aesthetic (**symboltabledialog.cpp**).

The dialog is triggered via the "Show Symbol Table" action, enabled only when symbols are present (**mainwindow.cpp**).

Key Files: Lexer.hpp, Lexer.cpp, symboltabledialog.hpp, symboltabledialog.cpp, mainwindow.cpp.

10.4 Token Sequence Visualization:

The **TokenSequenceDialog** displays the sequence of tokens produced by the lexer, providing insight into the tokenization process. Key features include:

- **Table Display**: Presents tokens in a table with columns for line number, token type, lexeme, and category (e.g., Keyword, Operator). The **populateTable** method maps **TokenType** and **TokenCategory** to strings (**tokensequencedialog.cpp, Token.hpp**).
- **Styling**: Mirrors the dark theme of **SymbolTableDialog**, with stretchable lexeme columns and pixel-based scrolling for smooth navigation (**tokensequencedialog.cpp**).
- **Integration**: Triggered via the "Show Token Sequence" action, enabled only when tokens are available (**mainwindow.cpp**).

This dialog aids users in debugging and understanding the lexer's output, complementing the symbol table.

Key Files: tokensequencedialog.hpp, tokensequencedialog.cpp, Token.hpp, mainwindow.cpp.

10.5 Find and Replace Functionality:

The **FindReplaceDialog** provides a GUI for searching and replacing text in the **CodeEditor**, leveraging the lexer's tokenization to ensure accurate modifications. Key features include:

- **Search Capabilities**: Supports case-sensitive and whole-word searches with forward/backward navigation, wrapping around the document if needed (**mainwindow.cpp**).
- **Replace Functionality**: Offers "Replace Next" and "Replace All" options, updating the editor's text while preserving token integrity (**mainwindow.cpp**).
- **Integration**: Connects to the editor via signals (findNext, replaceAll), with results displayed in the status bar (**mainwindow.cpp**).
- **GUI Design**: Uses a modal dialog with input fields and checkboxes, ensuring focus and usability (**findreplacedialog.hpp**).

This functionality enhances the editor's utility, allowing users to refine code before lexical analysis.

Key Files: findreplacedialog.hpp, mainwindow.hpp, mainwindow.cpp.

10.6 Dark Mode and Aesthetic Integration:

The GUI employs a dark theme for visual consistency and reduced eye strain, implemented in **main.cpp** via **setupDarkModePalette**. Key features include:

- **Palette Configuration**: Sets dark colors for windows, text, and selections (e.g., QColor(45, 45, 45) for backgrounds, Qt::white for text) using the Fusion style (**main.cpp**).
- **Stylesheet Enhancements**: Customizes tooltips, menus, and scrollbars for a cohesive look, applied across **MainWindow**, **SymbolTableDialog**, and **TokenSequenceDialog** (**main.cpp, symboltabledialog.cpp, tokensequencedialog.cpp**).
- **Syntax Highlighting Colors**: The PythonHighlighter uses contrasting colors (e.g., blue for keywords, green for strings) to align with the dark theme, ensuring readability (**pythonhighlighter.cpp**).

Key Files: main.cpp, pythonhighlighter.cpp, symboltabledialog.cpp, tokensequencedialog.cpp.

10.7 Error Reporting Interface:

Describe the **errordialog** class (**errordialog.hpp**, **errordialog.cpp**), which displays lexical errors detected by the analyzer. Key features include:

- **Table Structure**: A **QTableWidget** displays errors with columns for line number, lexeme, and error message, populated via **setErrorData** using std::vector<std::tuple<int, std::string, std::string>> from **MainWindow::lastErrors**.
- **Styling**: The dialog adopts a dark theme with alternating row colors, scrollable content, and a modern look (CSS-like stylesheet in **applyStyling**). Row selection and grid lines enhance readability.
- **Integration with Lexer**: Errors are collected in **MainWindow::runLexer** from **Lexer::getErrors** and passed to the dialog. The Lexer_error struct (**Lexer.hpp**) defines error attributes (message, line, lexeme).
- **Context Menu**: A right-click context menu allows copying cell content to the clipboard, improving usability (**showContextMenu, copyCell**).

Key Files: errordialog.cpp, errordialog.hpp, mainwindow.cpp,mainwindow.hpp

11. User Interface :

11.1 Test Case 1 Window: without errors

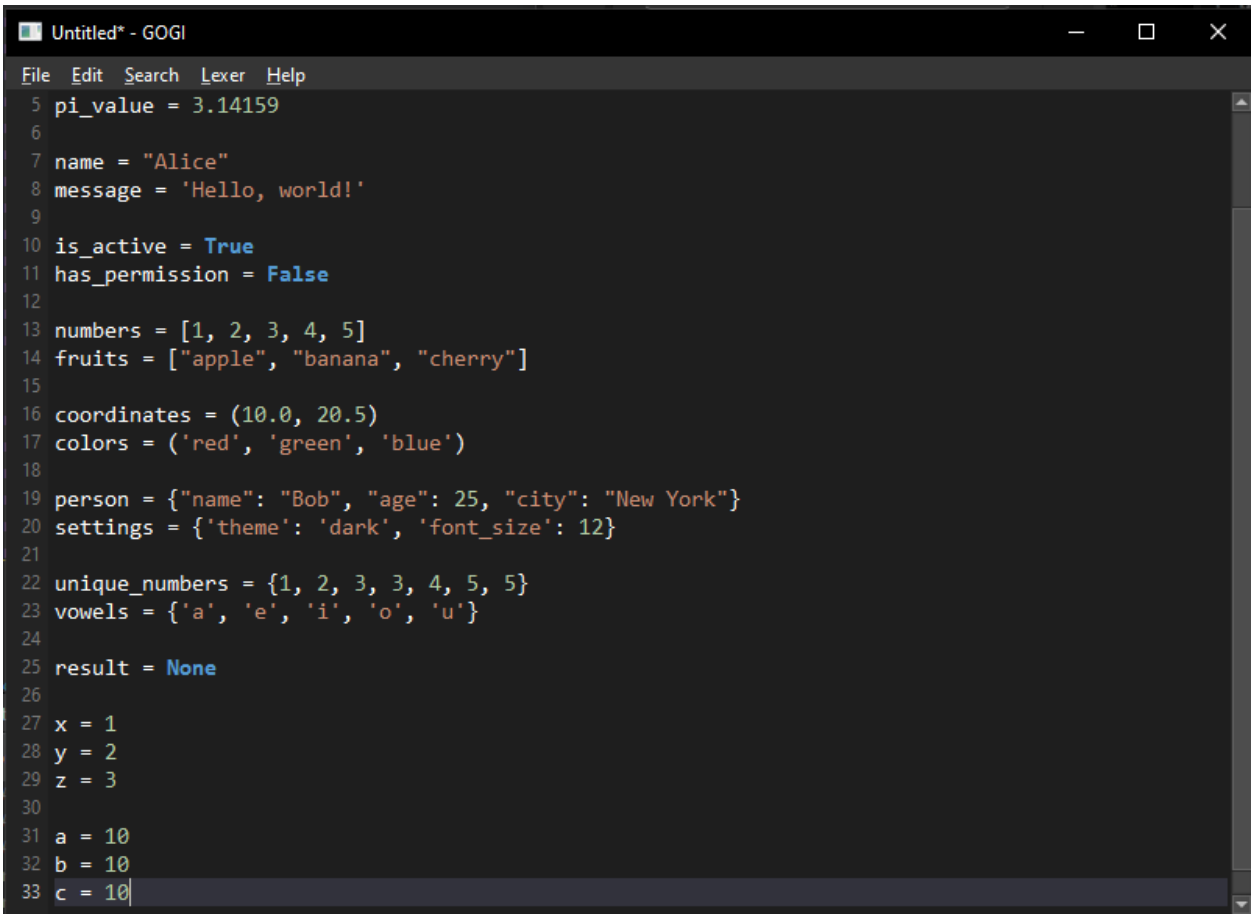
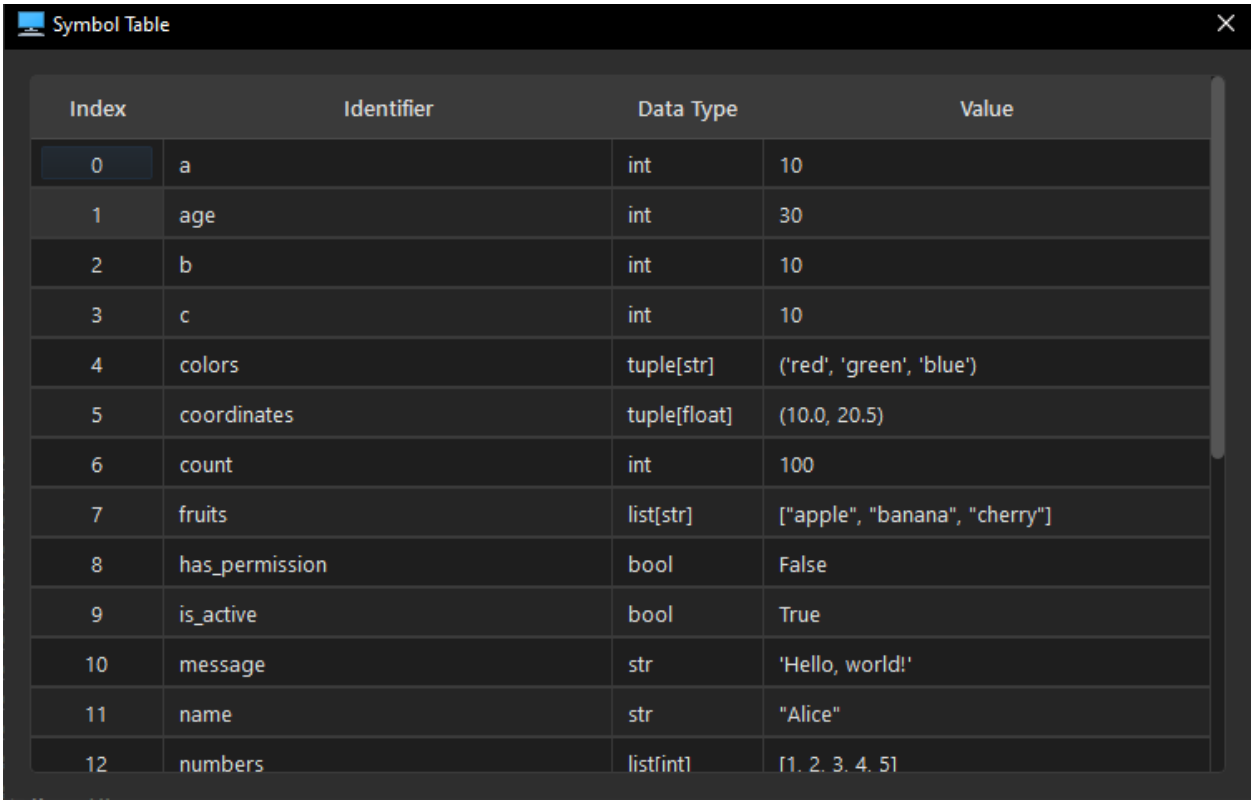
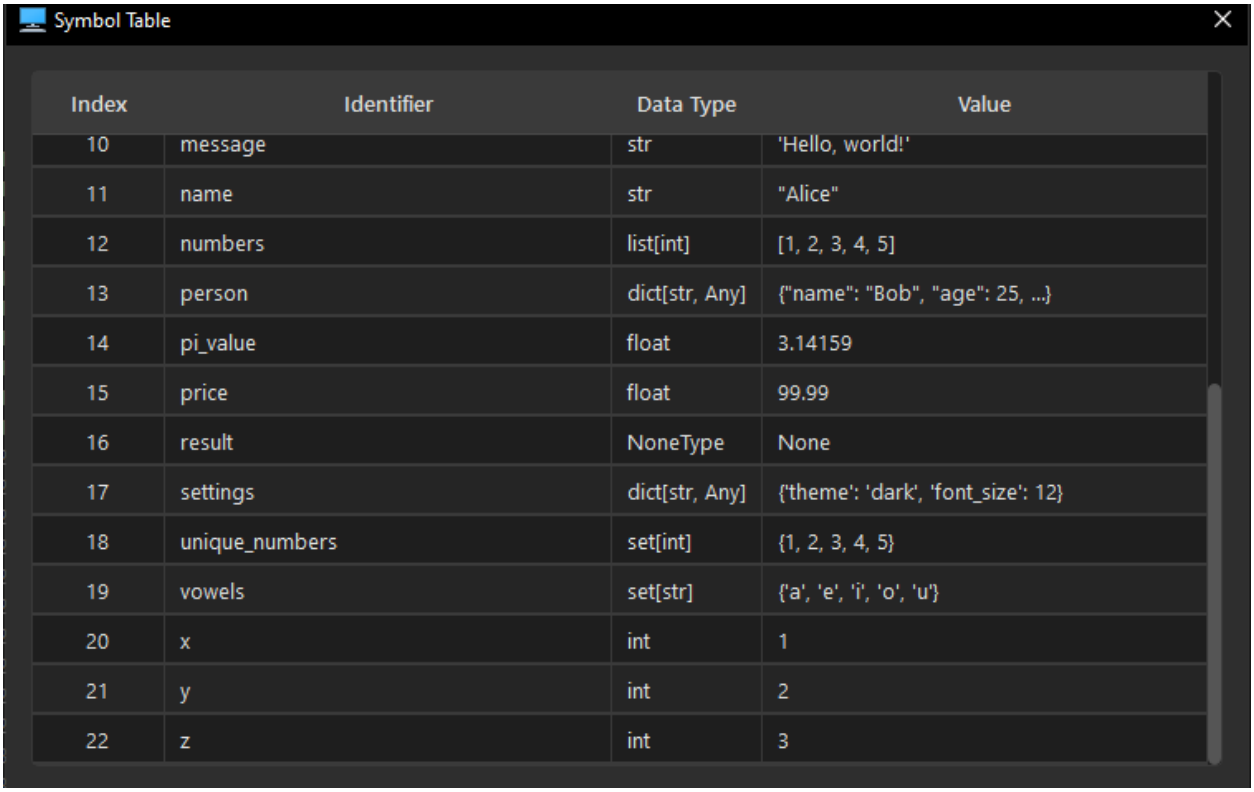


Figure 24

11.2 Symbol Table Window:





11.3 Token Table Window:

Token Sequence

Line	Type	Lexeme	Category
1	IDENTIFIER	age	Identifier
1	ASSIGN	=	Operator
1	NUMBER_LITERAL	30	Number
2	IDENTIFIER	count	Identifier
2	ASSIGN	=	Operator
2	NUMBER_LITERAL	100	Number
4	IDENTIFIER	price	Identifier
4	ASSIGN	=	Operator
4	NUMBER_LITERAL	99.99	Number
5	IDENTIFIER	pi_value	Identifier
5	ASSIGN	=	Operator
5	NUMBER_LITERAL	3.14159	Number
7	IDENTIFIER	name	Identifier
7	ASSIGN	=	Operator

Token Sequence

Line	Type	Lexeme	Category
7	STRING_LITERAL	Alice	String
8	IDENTIFIER	message	Identifier
8	ASSIGN	=	Operator
8	STRING_LITERAL	Hello, world!	String
10	IDENTIFIER	is_active	Identifier
10	ASSIGN	=	Operator
10	TRUE	True	Keyword
11	IDENTIFIER	has_permission	Identifier
11	ASSIGN	=	Operator
11	FALSE	False	Keyword
13	IDENTIFIER	numbers	Identifier
13	ASSIGN	=	Operator
13	LBRACKET	[	Punctuation
13	NUMBER_LITERAL	1	Number

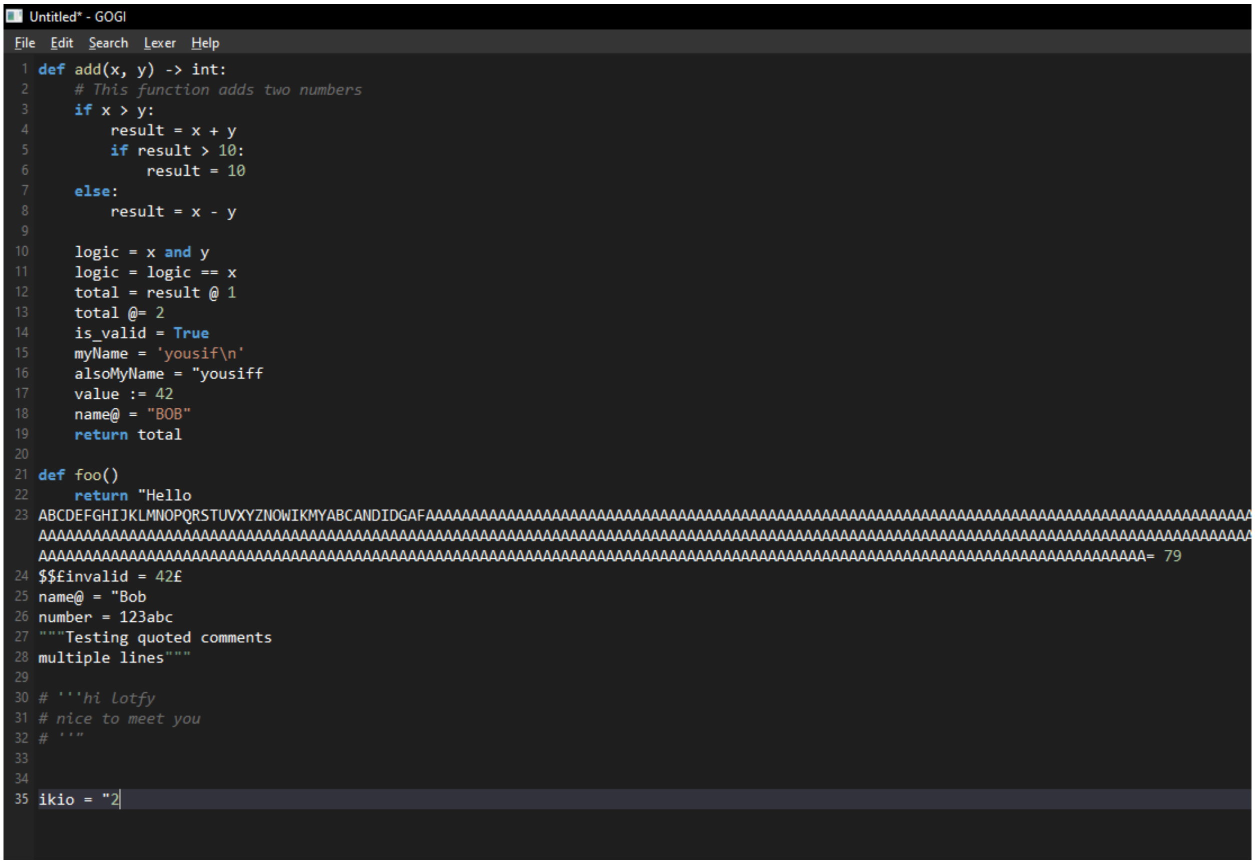
Token Sequence

Line	Type	Lexeme	Category
13	COMMA	,	Punctuation
13	NUMBER_LITERAL	3	Number
13	COMMA	,	Punctuation
13	NUMBER_LITERAL	4	Number
13	COMMA	,	Punctuation
13	NUMBER_LITERAL	5	Number
13	RBRACKET	]	Punctuation
14	IDENTIFIER	fruits	Identifier
14	ASSIGN	=	Operator
14	LBRACKET	[	Punctuation
14	STRING_LITERAL	apple	String
14	COMMA	,	Punctuation
14	STRING_LITERAL	banana	String
14	COMMA	,	Punctuation

Token Sequence

Line	Type	Lexeme	Category
14	STRING_LITERAL	cherry	String
14	RBRACKET	]	Punctuation
16	IDENTIFIER	coordinates	Identifier
16	ASSIGN	=	Operator
16	LPAREN	(	Punctuation
16	NUMBER_LITERAL	10.0	Number
16	COMMA	,	Punctuation
16	NUMBER_LITERAL	20.5	Number
16	RPAREN	)	Punctuation
17	IDENTIFIER	colors	Identifier
17	ASSIGN	=	Operator
17	LPAREN	(	Punctuation
17	STRING_LITERAL	red	String
17	COMMA	,	Punctuation

11.4 Test Case 2 Window: with errors



```
Untitled* - GOGI
File  Edit  Search  Lexer  Help
def add(x, y) -> int:
    # This function adds two numbers
    if x > y:
        result = x + y
        if result > 10:
            result = 10
    else:
        result = x - y

    logic = x and y
    logic = logic == x
    total = result @ 1
    total @= 2
    is_valid = True
    myName = 'yousif\n'
    alsoMyName = "yousiff
    value := 42
    name@ = "BOB"
    return total

def foo()
    return "Hello
ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAFAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA= 79
$$£invalid = 42£
name@ = "Bob
number = 123abc
"""Testing quoted comments
multiple lines"""

# '''hi lotfy
# nice to meet you
# '''


ikio = "2
```

Figure 25

11.5 Symbol Table Window:

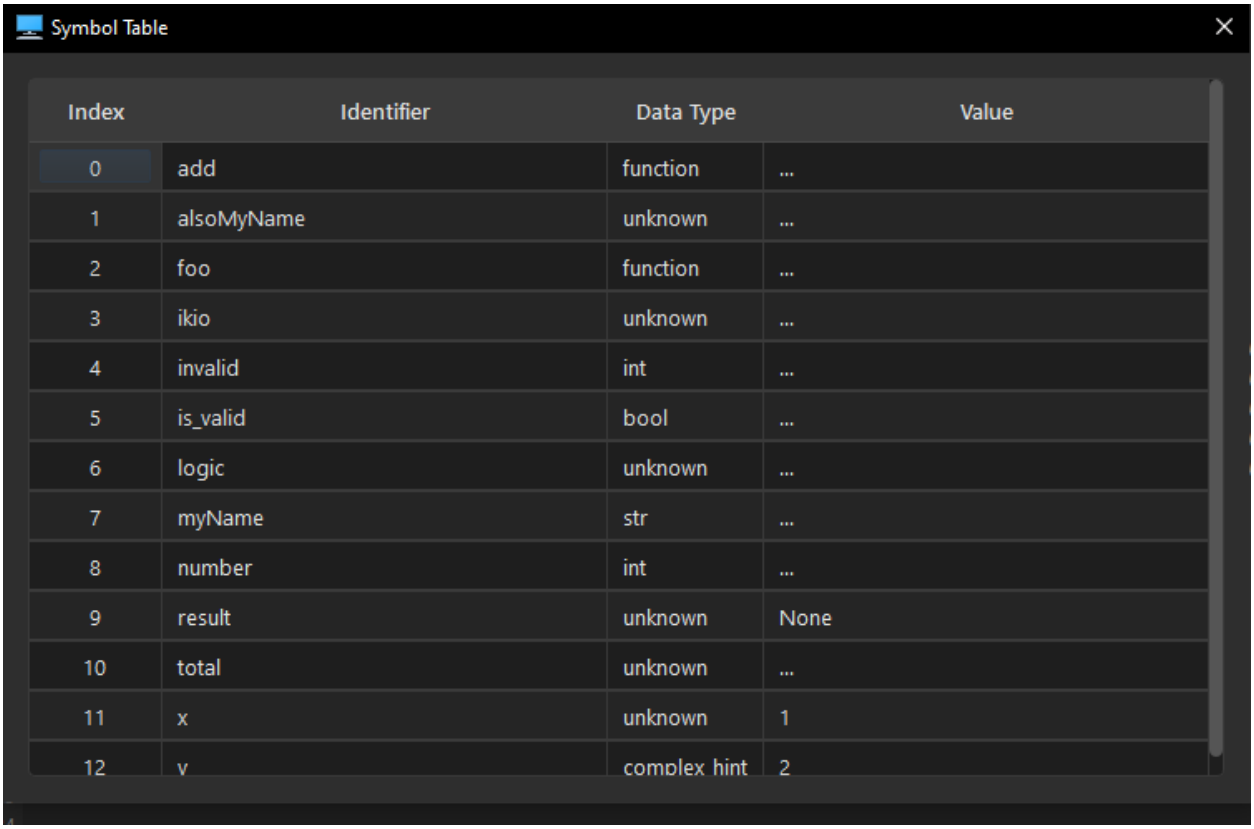


Figure 26

11.6 Token Table Window:

Token Sequence

Line	Type	Lexeme	Category
1	DEF	def	Keyword
1	IDENTIFIER	add	Identifier
1	LPAREN	(	Punctuation
1	IDENTIFIER	x	Identifier
1	COMMA	,	Punctuation
1	IDENTIFIER	y	Identifier
1	RPAREN	)	Punctuation
1	FUNC_RETURN_TYPE	->	Operator
1	INT_KEYWORD	int	Keyword
1	COLON	:	Punctuation
3	INDENT	INDENT	Punctuation
3	IF	if	Keyword
3	IDENTIFIER	x	Identifier
3	GREATER	>	Operator

Token Sequence

Line	Type	Lexeme	Category
3	IDENTIFIER	y	Identifier
3	COLON	:	Punctuation
4	INDENT	INDENT	Punctuation
4	IDENTIFIER	result	Identifier
4	ASSIGN	=	Operator
4	IDENTIFIER	x	Identifier
4	PLUS	+	Operator
4	IDENTIFIER	y	Identifier
5	IF	if	Keyword
5	IDENTIFIER	result	Identifier
5	GREATER	>	Operator
5	NUMBER_LITERAL	10	Number
5	COLON	:	Punctuation
6	INDENT	INDENT	Punctuation

Token Sequence

Line	Type	Lexeme	Category
6	IDENTIFIER	result	Identifier
6	ASSIGN	=	Operator
6	NUMBER_LITERAL	10	Number
7	DEDENT	DEDENT	Punctuation
7	DEDENT	DEDENT	Punctuation
7	ELSE	else	Keyword
7	COLON	:	Punctuation
8	INDENT	INDENT	Punctuation
8	IDENTIFIER	result	Identifier
8	ASSIGN	=	Operator
8	IDENTIFIER	x	Identifier
8	MINUS	-	Operator
8	IDENTIFIER	y	Identifier
10	DEDENT	DEDENT	Punctuation

Token Sequence

Line	Type	Lexeme	Category
10	IDENTIFIER	logic	Identifier
10	ASSIGN	=	Operator
10	IDENTIFIER	x	Identifier
10	AND	and	Keyword
10	IDENTIFIER	y	Identifier
11	IDENTIFIER	logic	Identifier
11	ASSIGN	=	Operator
11	IDENTIFIER	logic	Identifier
11	EQUAL	==	Operator
11	IDENTIFIER	x	Identifier
12	IDENTIFIER	total	Identifier
12	ASSIGN	=	Operator
12	IDENTIFIER	result	Identifier
12	MATMUL	@	Operator

Token Sequence

Line	Type	Lexeme	Category
12	NUMBER_LITERAL	1	Number
13	IDENTIFIER	total	Identifier
13	MATMUL_ASSIGN	@=	Operator
13	NUMBER_LITERAL	2	Number
14	IDENTIFIER	is_valid	Identifier
14	ASSIGN	=	Operator
14	TRUE	True	Keyword
15	IDENTIFIER	myName	Identifier
15	ASSIGN	=	Operator
15	STRING_LITERAL	yousif\n	String
16	IDENTIFIER	alsoMyName	Identifier
16	ASSIGN	=	Operator
16	UNKNOWN	"yousiff	Unknown
17	IDENTIFIER	value	Identifier

Token Sequence

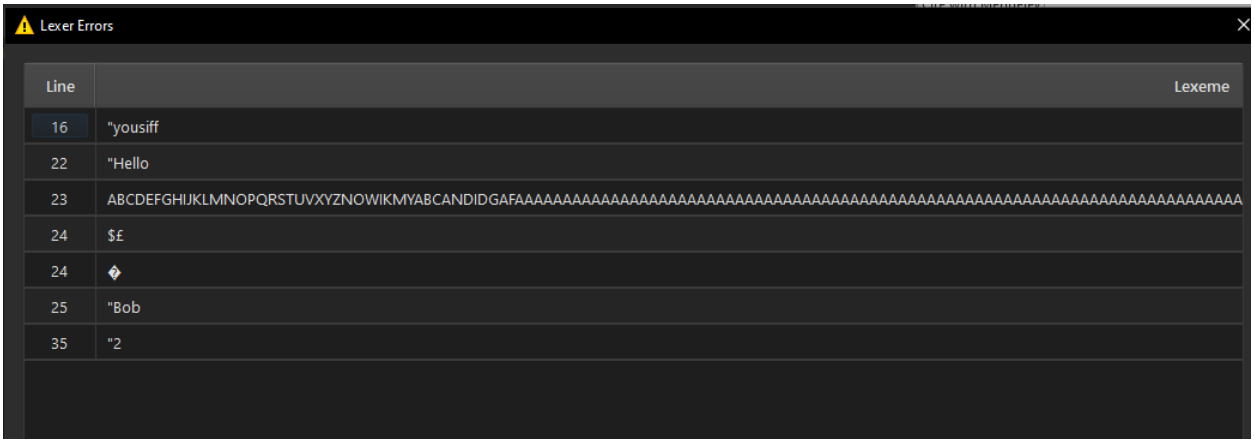
Line	Type	Lexeme	Category
17	WALNUT	:=	Operator
17	NUMBER_LITERAL	42	Number
18	IDENTIFIER	name	Identifier
18	MATMUL	@	Operator
18	ASSIGN	=	Operator
18	STRING_LITERAL	BOB	String
19	RETURN	return	Keyword
19	IDENTIFIER	total	Identifier
21	DEDENT	DEDENT	Punctuation
21	DEF	def	Keyword
21	IDENTIFIER	foo	Identifier
21	LPAREN	(	Punctuation
21	RPAREN	)	Punctuation
22	INDENT	INDENT	Punctuation
22	RETURN	return	Keyword

Token Sequence

Line	Type	Lexeme	Category
22	UNKNOWN	"Hello	Unknown
23	DEDENT	DEDENT	Punctuation
23	UNKNOWN	ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAF...	Unknown
23	ASSIGN	=	Operator
23	NUMBER_LITERAL	79	Number
24	UNKNOWN	$£	Unknown
24	IDENTIFIER	invalid	Identifier
24	ASSIGN	=	Operator
24	NUMBER_LITERAL	42	Number
24	UNKNOWN	◆	Unknown
25	IDENTIFIER	name	Identifier
25	MATMUL	@	Operator
25	ASSIGN	=	Operator
25	UNKNOWN	"Bob	Unknown

26	IDENTIFIER	number	Identifier
26	ASSIGN	=	Operator
26	NUMBER_LITERAL	123	Number
26	IDENTIFIER	abc	Identifier
35	IDENTIFIER	ikio	Identifier
35	ASSIGN	=	Operator
35	UNKNOWN	"2	Unknown

11.7 Error Table Window:


Lexer Errors
Line
Lexeme
16
"yousiff
22
"Hello
23
ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAFAAA

12. Conclusion:

The development of **a lexical analyzer for Python using C++** has provided valuable insights into the foundational stages of language processing. By breaking down Python source code into a sequence of tokens, the lexer plays a critical role in enabling further syntactic and semantic analysis.

Through this project, we demonstrated how core concepts such as tokenization, pattern matching, and symbol table management can be implemented effectively in C++. Despite Python's complex and dynamic syntax, the lexer successfully identifies and categorizes various elements like keywords, identifiers, literals, and operators.

Implementing a Python lexer in C++ not only deepens our understanding of compiler design but also showcases the power and flexibility of C++ in handling language analysis tasks. This work lays the groundwork for extending the project into a full parser or interpreter and highlights the practical application of compiler theory in real-world scenarios.