Final Phase - Python Lexer and Parser

CSE439-Design of Compilers
17 May 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:

This document details the design, implementation, and functionality of the Python Lexer and Parser project, with the primary objective of developing the initial two critical phases of a compiler—lexical analysis and syntax analysis—tailored for a defined subset of the Python programming language. The entire system, encompassing the lexer, parser, and a supporting Graphical User Interface (GUI), has been implemented using the C++ programming language. The project required a thorough understanding of compiler theory and Python language constructs, offering a practical application of these principles.

A lexical analyzer, or lexer, serves as the first phase of a compiler or interpreter, with the main role of reading the source code and converting it into a stream of tokens—the meaningful building blocks of a programming language, including identifiers, keywords, operators, literals, and punctuation symbols. In this project, the lexer, developed in C++, scans Python source code, identifies valid tokens, and populates a symbol table that tracks identifiers along with their types and values, essential for subsequent phases like parsing and semantic analysis. Python's dynamically typed nature and indentation sensitivity present unique challenges, and building the lexer in C++—leveraging its strong performance and memory control capabilities—provides deep insight into the internal workings of interpreters and how language syntax is processed at a lower level, making it ideal for creating fast and efficient analysis tools.

Subsequently, the syntax analyzer (parser) takes the token stream generated by the lexer, verifies it against a defined grammar for the Python subset, and constructs a parse tree representing the syntactic structure of the code. The parser is also equipped to detect and report syntax errors. A significant aspect of this project is the development of a user-friendly GUI, which facilitates interaction with the compiler components, allowing users to load Python source files, view generated tokens, inspect the symbol table, visualize the parse tree (or relevant error messages), and observe the error handling capabilities of both the lexer and parser. This report aims to provide a clear and detailed overview of our efforts, methodologies, and the final implemented system, demonstrating a practical application of compiler design principles.

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;
    }
}
```

[1]

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.

[1]

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:

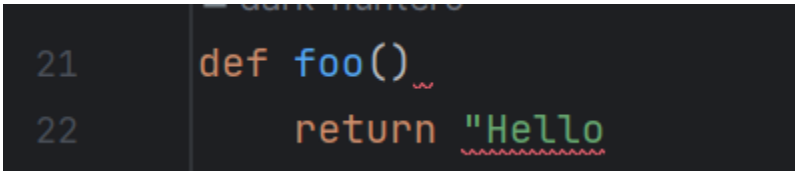
The lexical analyzer has been extended to detect and recover from a variety of **Python-specific lexical errors**. These are essential to prevent malformed tokens from affecting further stages of compilation. Each error type is described below along with how it is detected and managed.

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.



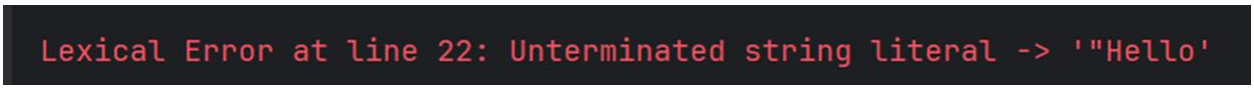


Figure 19

```
    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:

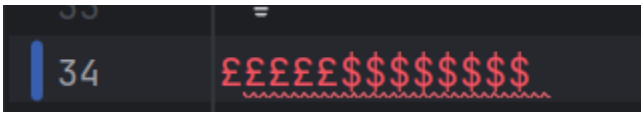
Characters not recognized in Python (e.g., @, $, ~ in invalid contexts).

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.



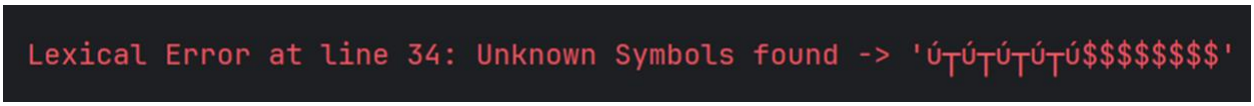


Figure 21

```
//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

```
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.

```
ABCDEFGHIJKLMNOPQRSTUVXYZNOWIKMYABCANDIDGAFAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
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.

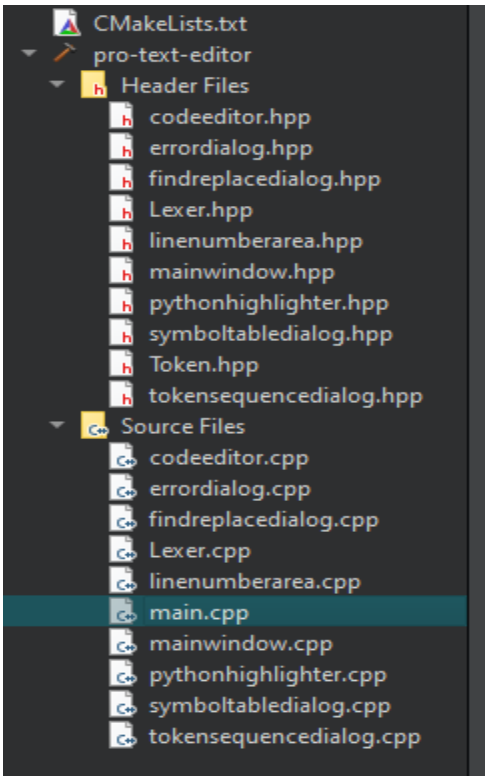


Figure 23

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

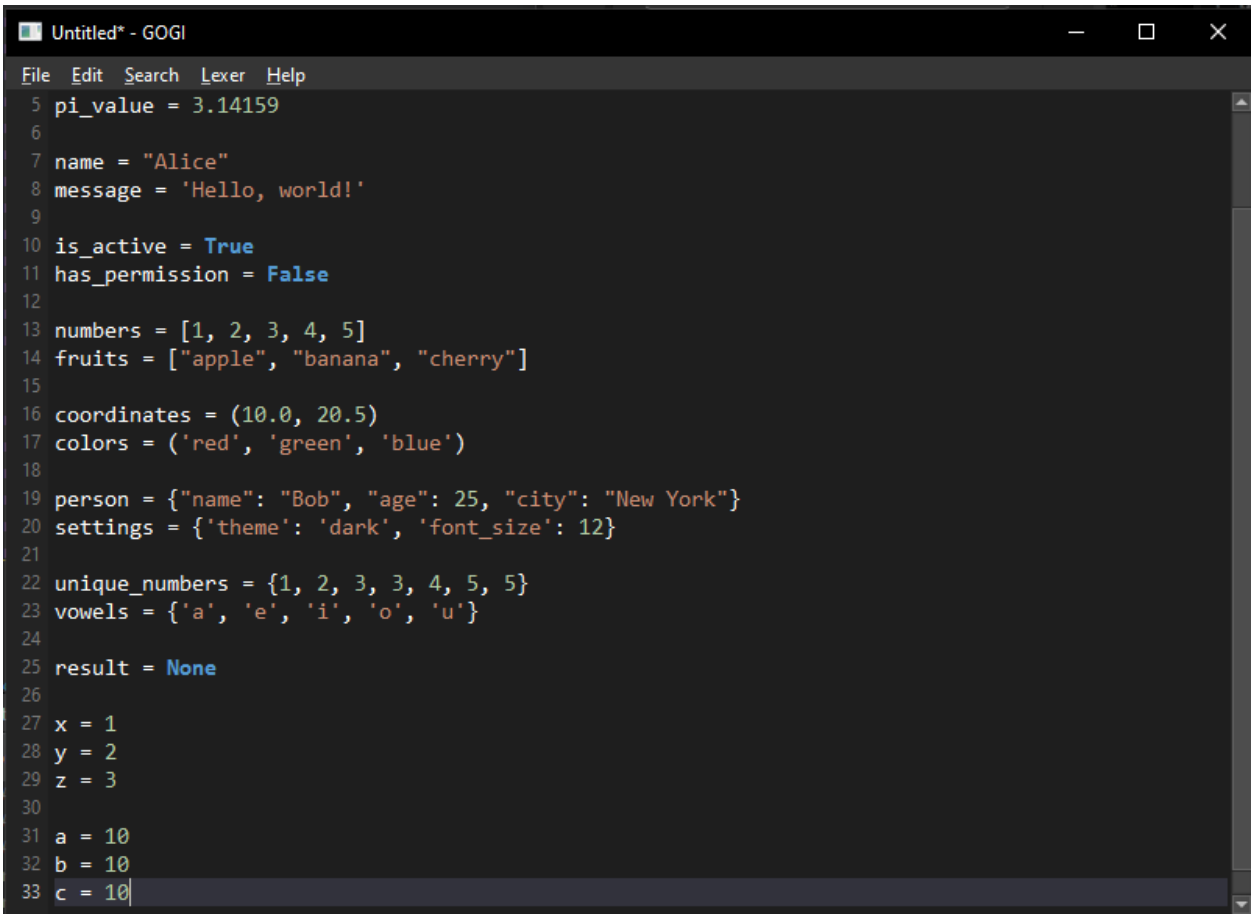
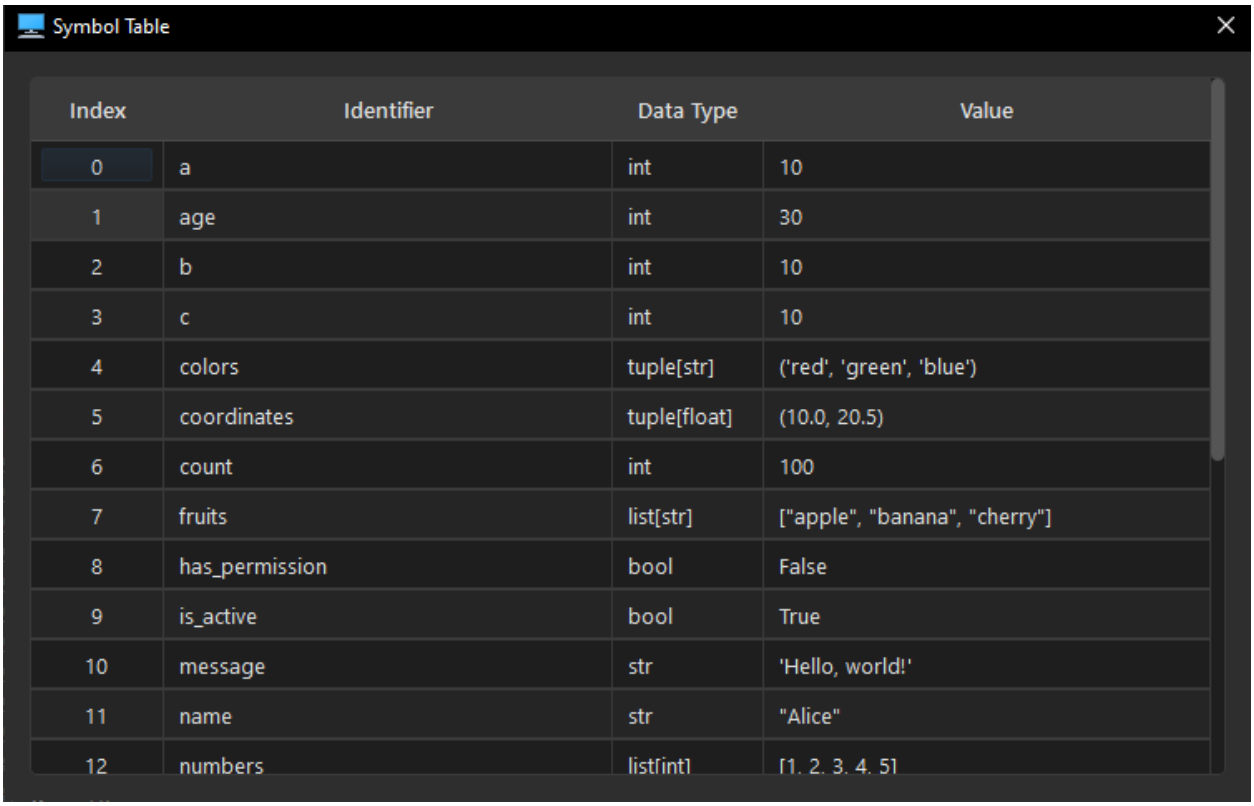
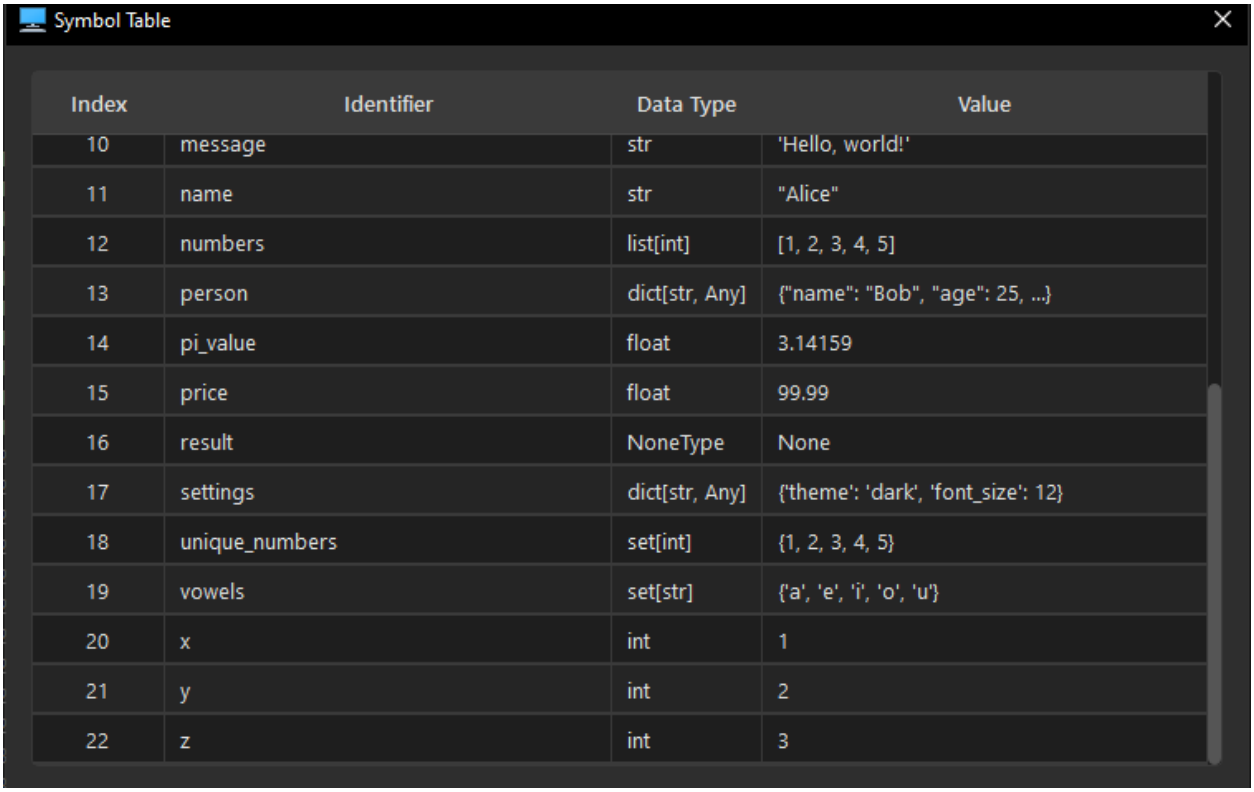


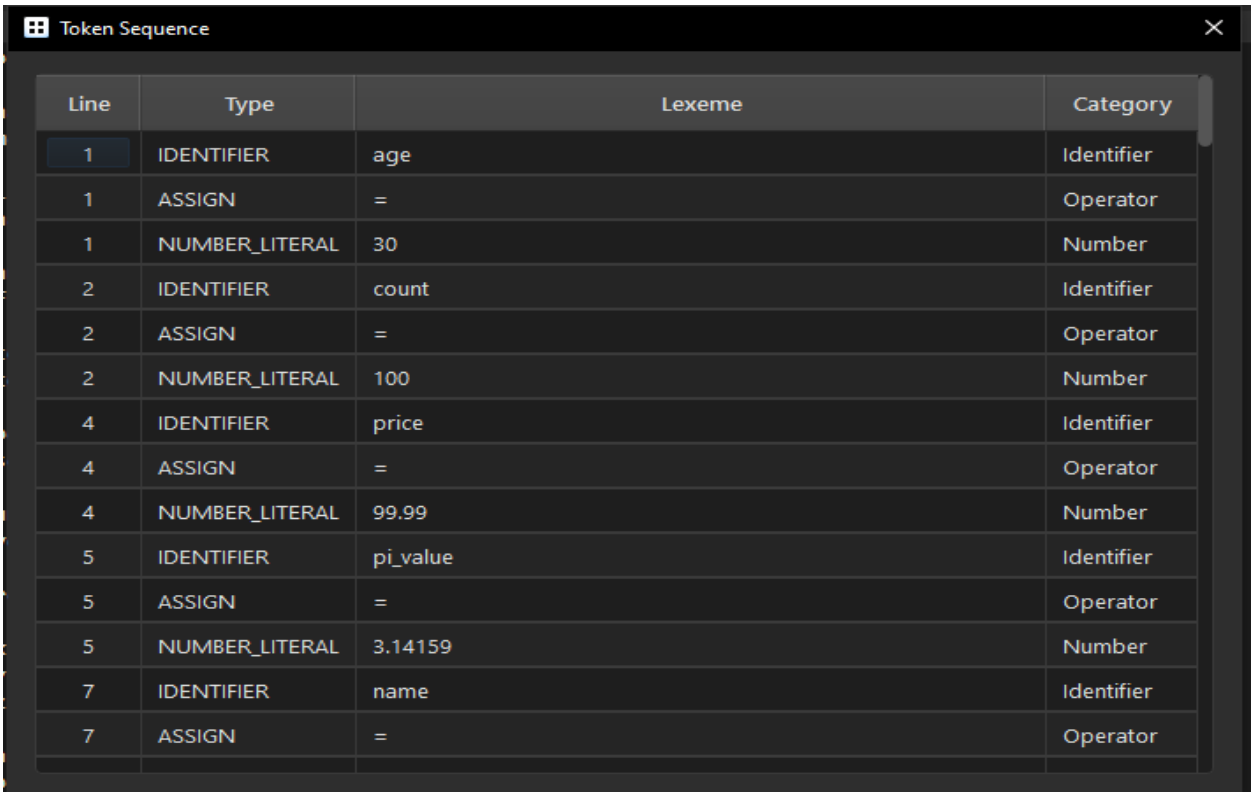
Figure 24

11.2 Symbol Table Window:





11.3 Token Table Window:



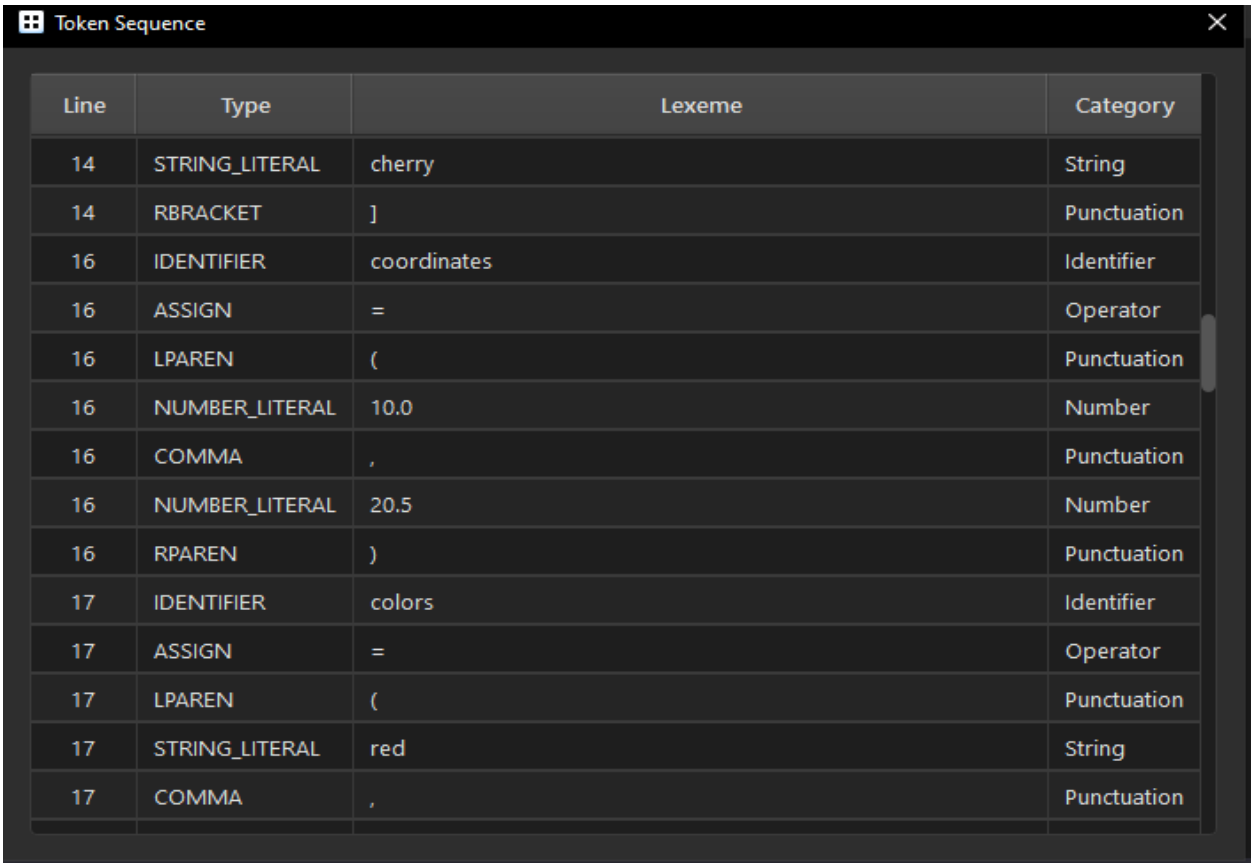
Line	Type	Lexeme	Category
1	IDENTIFIER	age	Identifier
1	ASSIGN	=	Operator
1	NUMBER_LITERAL	30	Number
2	IDENTIFIER	count	Identifier
2	ASSIGN	=	Operator
2	NUMBER_LITERAL	100	Number
4	IDENTIFIER	price	Identifier
4	ASSIGN	=	Operator
4	NUMBER_LITERAL	99.99	Number
5	IDENTIFIER	pi_value	Identifier
5	ASSIGN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



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

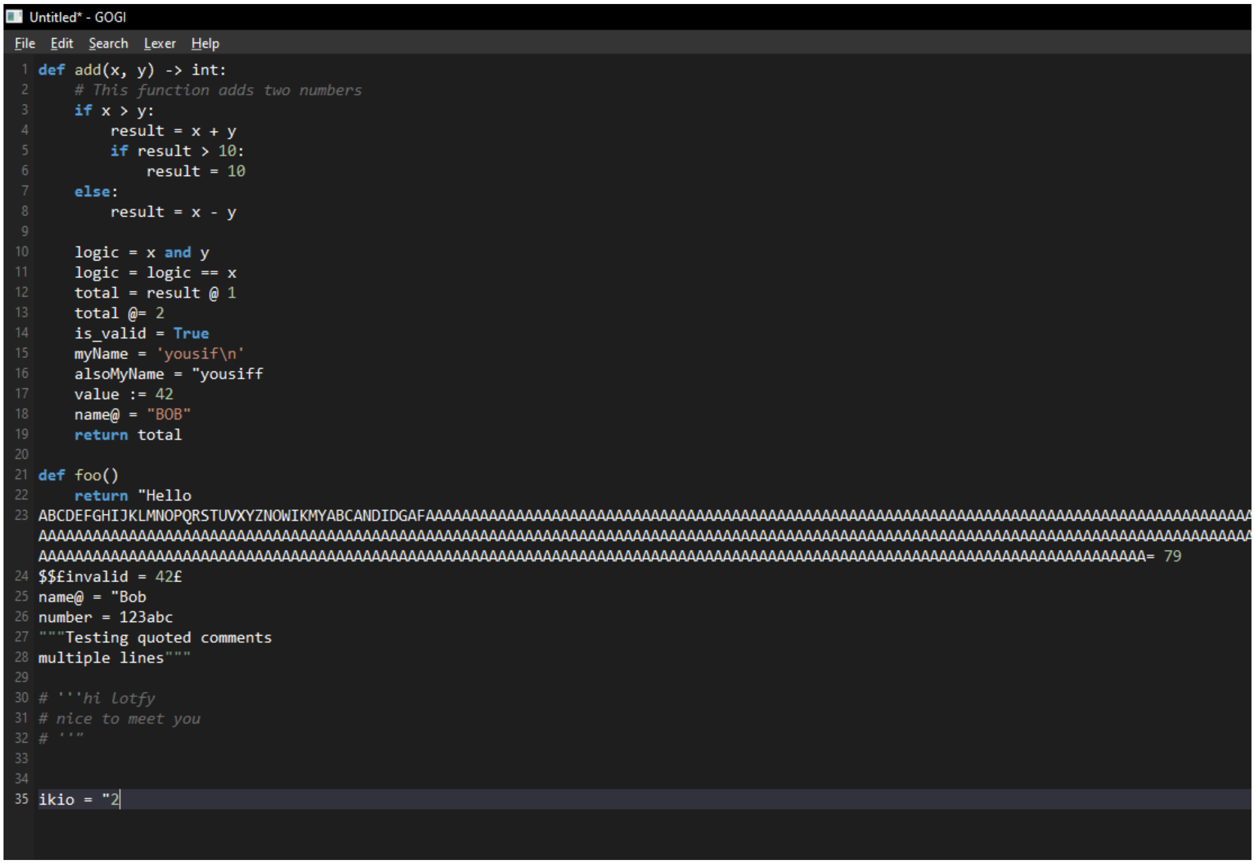


Figure 25

11.5 Symbol Table Window:

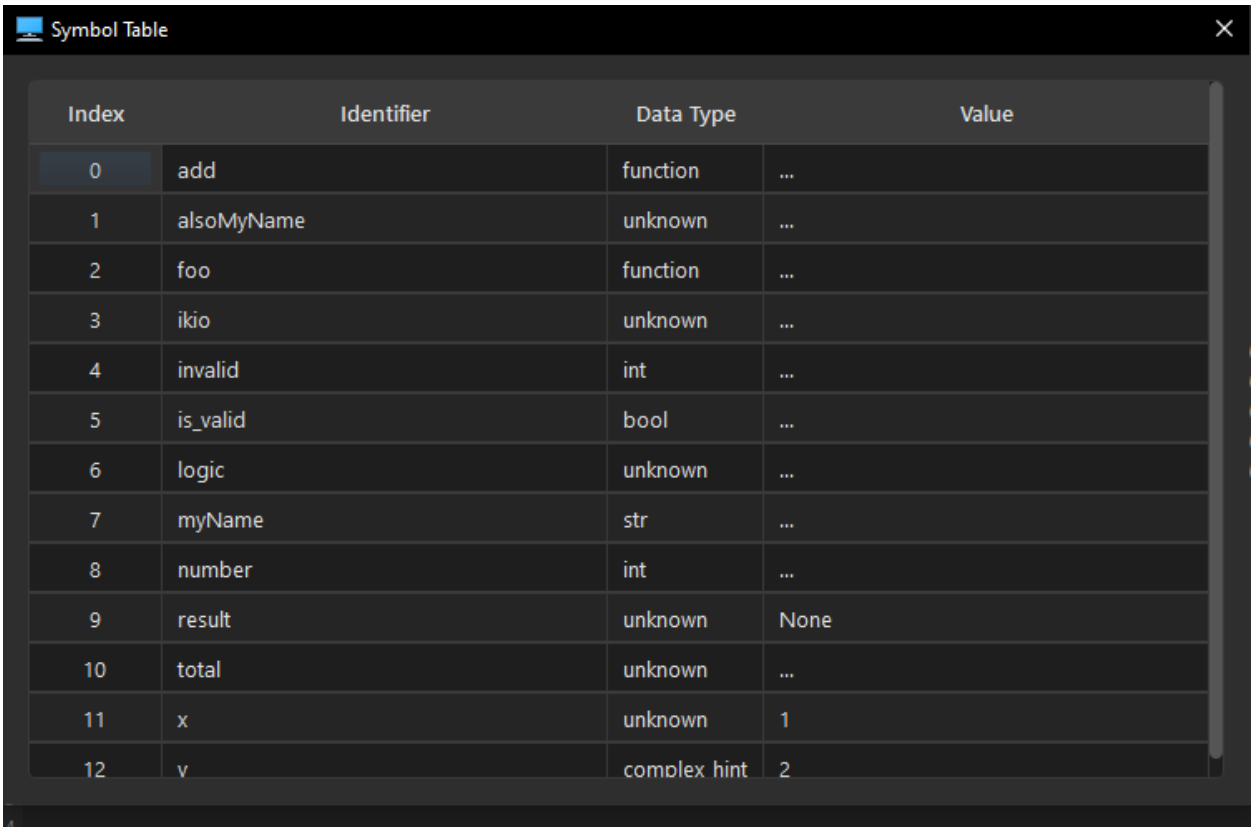
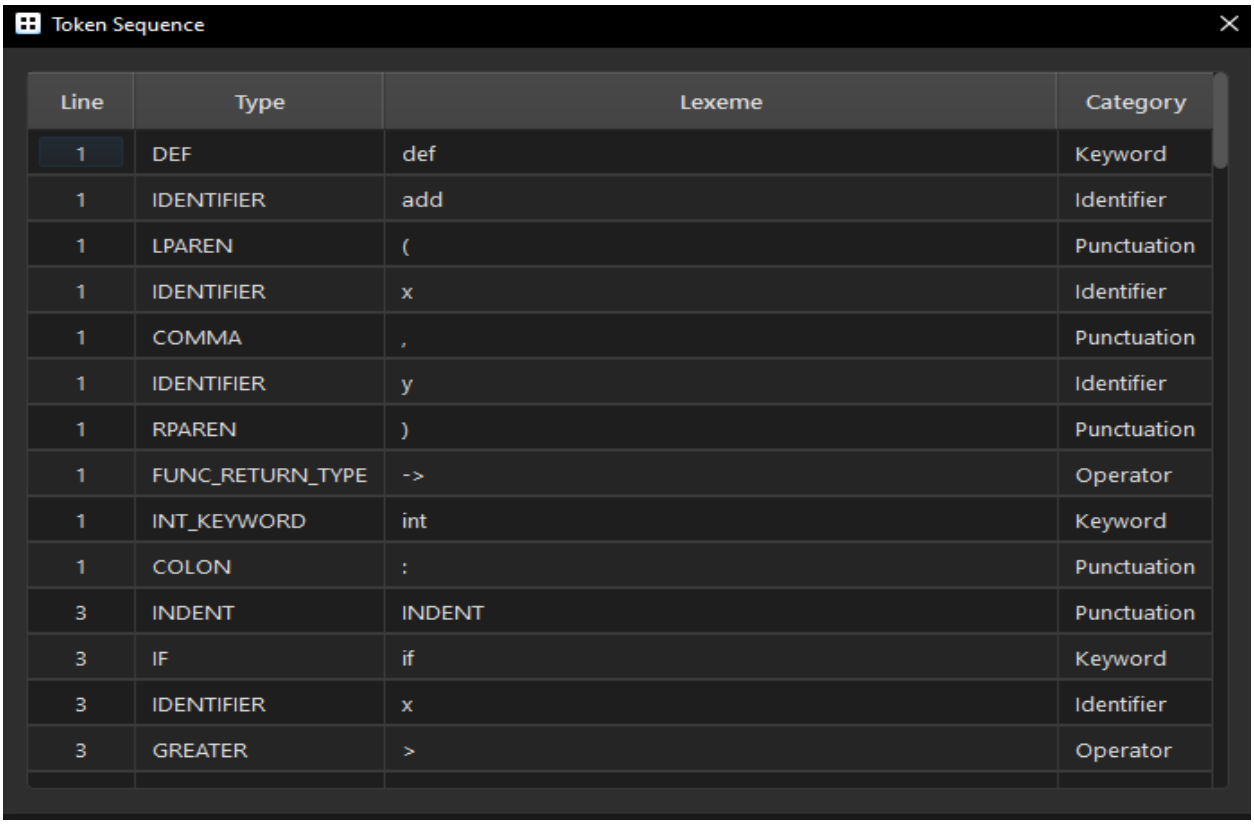


Figure 26

11.6 Token Table Window:



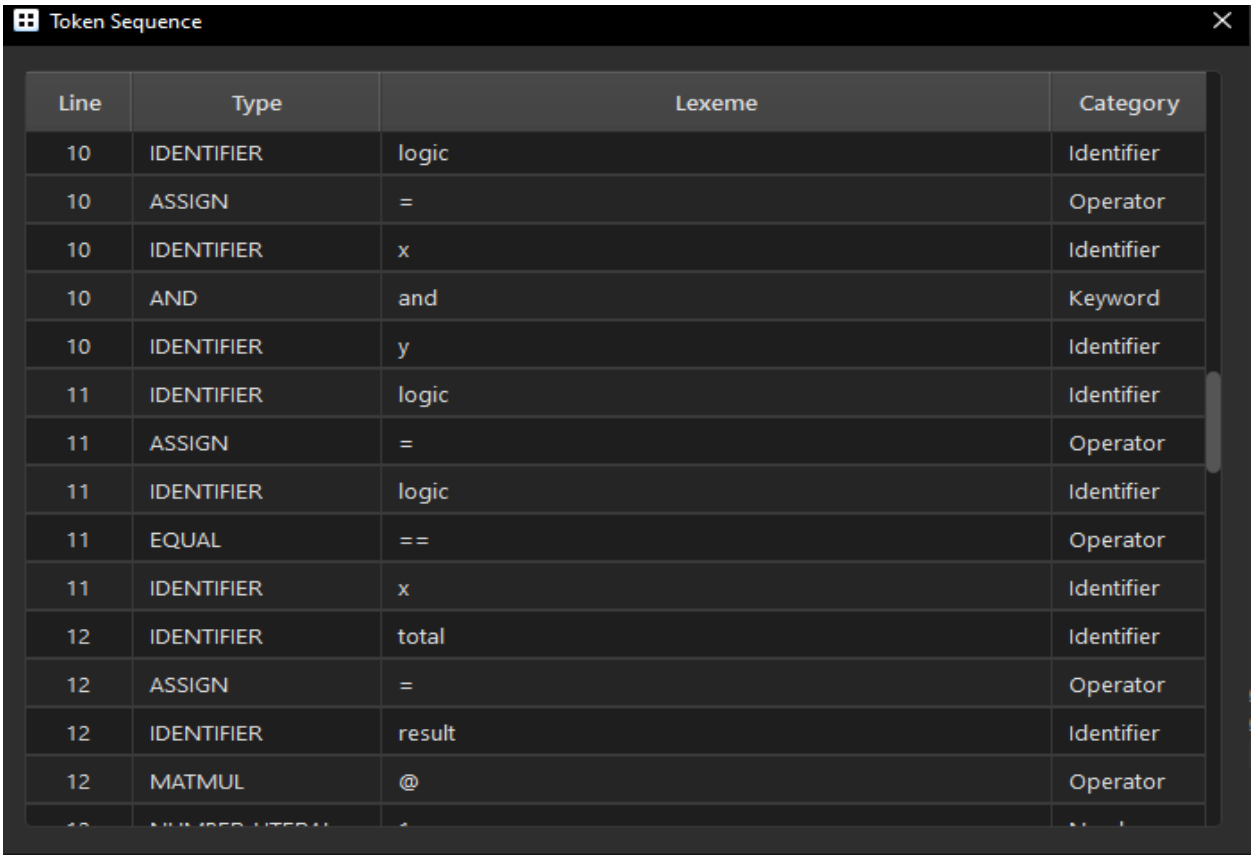
Line	Type	Lexeme	Category
1	DEF	def	Keyword
1	IDENTIFIER	add	Identifier
1	LPAREN	(	Punctuation
1	IDENTIFIER	x	Identifier
1	COMMA	,	Punctuation
1	IDENTIFIER	y	Identifier
1	RPAREN	)	Punctuation
1	FUNC_RETURN_TYPE	->	Operator
1	INT_KEYWORD	int	Keyword
1	COLON	:	Punctuation
3	INDENT	INDENT	Punctuation
3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



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

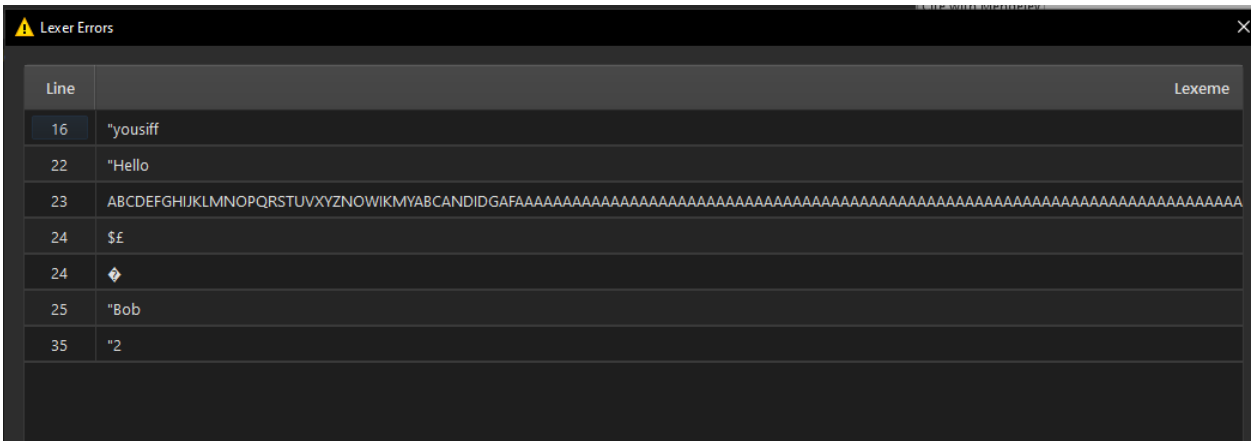
Token Sequence

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. Project Scope and Language Specifications:

12.1. Overall Project Scope

This project focuses on the development of a lexical analyzer (lexer) and a syntax analyzer (parser) for a defined subset of the Python programming language, as detailed in our "Python Specifications Document" (Phase 1 Submitted on Friday 14/3/2025). The lexer and parser are implemented in C++ and include a Graphical User Interface (GUI) for user interaction, token display, symbol table visualization, and parse tree representation.

12.2. Implemented Python Language Subset Summary

Our C++ lexer and parser support the following core Python constructs, which are a subset of those detailed in the "Python Specifications Document":

Core Syntax Elements:

- **Keywords**: A defined set
  including if, else, elif, while, def, return, True, False, None, and, or, not. (Adjust this list to be accurate for your project. Refer to Spec Doc Table 2.1 and list ONLY what you implemented).
- **Identifiers**: Standard Python rules for variable and function names (Ref: Spec Doc Sections 2.3 & 3.1).
- **Literals**: Recognition of Integer, basic String (single/double-quoted), and Boolean literals (Ref: Spec Doc Sections 4.2.1, 4.6.1, 4.3). Floating-point, complex numbers, and collection literals (list, tuple, dict, set) are generally not supported beyond basic tokenization.

Operators:

- **Arithmetic**: +, -, *, / (Parentheses for grouping) (Ref: Spec Doc Section 7.1).
- **Boolean/Comparison**: and, or, not, ==, !=, <, >, <=, >= (Ref: Spec Doc Section 7.2).

Statements:

- **Assignment**: identifier = expression (Ref: Spec Doc Section 6.1).
- **Function Definition**: def function_name(parameters): ... (with simple parameters) (Ref: Spec Doc Section 5.2.2).
- **Function Calls**: function_name(arguments) (with simple arguments) (Ref: Spec Doc Section 6.6).
- **Conditional**: if/elif/else blocks (Ref: Spec Doc Section 6.4).
- **Iteration**: while condition: ... loops (Ref: Spec Doc Section 6.5.2).
- **return statements** (Ref: Spec Doc Section 6.3).

13. Core Project Structure and Files

This section outlines the organization of the source code, header files, and other relevant project files except the Gui files. A clear structure was adopted to promote modularity, readability, and maintainability of the C++ implementation for the Python Lexer and Parser.

13.1. Directory Structure Overview

The project is organized into the following main directories:

```
.
├── Lexer/                  # Source files related to the Lexical Analyzer
│   └── Lexer.cpp
├── Parser/                 # Source files and grammar related to the Syntax Analyzer
│   ├── PYCFG.gram          # Context-Free Grammar for the Python subset
│   └── Parser.cpp
├── include/                # Header files for the project
│   ├── ASTNode.hpp
│   ├── Expressions.hpp
│   ├── Helpers.hpp
│   ├── Lexer.hpp
│   ├── Literals.hpp
│   ├── Parser.hpp
│   ├── Statements.hpp
│   ├── Token.hpp
│   ├── UtilNodes.hpp       # Utility/helper AST node structures
│   └── tempAST.hpp         # Temporary or developmental AST structures
├── CMakeLists.txt          # CMake build system configuration file
├── main.cpp                # Main entry point of the application (likely initializing GUI and compiler)
└── test.py                 # Example Python file for testing the lexer and parser
```

Figure 27

13.2. Key Directories and Their Contents

- **Lexer/**:
  - Contains the core implementation of the lexical analyzer.
  - Lexer.cpp: The C++ source file implementing the logic for tokenizing Python source code based on the specifications.
- **Parser/**:
  - Contains the implementation of the syntax analyzer and its associated grammar.
  - Parser.cpp: The C++ source file implementing the parsing logic (e.g., recursive descent or table-driven parser) to build a parse tree or Abstract Syntax Tree (AST).
  - PYCFG.gram: (Assuming this file contains your grammar definition) This file defines the Context-Free Grammar (CFG) rules for the subset of Python supported by the parser. It specifies the syntactic structure of valid programs.

- **include/**:
  - This crucial directory contains all the C++ header files (.hpp or .h) for the project. These files define the interfaces for classes, structures, and functions used across different modules.
  - ASTNode.hpp: Defines the base structure or class for nodes in the Abstract Syntax Tree (AST), representing different constructs of the language.
  - Expressions.hpp: Header file likely defining structures or classes related to parsing and representing expressions (arithmetic, boolean, etc.).
  - Helpers.hpp: Contains utility functions or helper classes used throughout the project.
  - Lexer.hpp: Header file for the Lexer class, declaring its public interface (e.g., methods to get tokens).
  - Literals.hpp: Header file likely defining how different literal values (integers, strings) are represented or parsed.
  - Parser.hpp: Header file for the Parser class, declaring its public interface (e.g., method to start parsing, get the parse tree).
  - Statements.hpp: Header file likely defining structures or classes related to parsing and representing various Python statements (assignment, if, while, etc.).
  - Token.hpp: Defines the Token structure or class, representing the tokens produced by the lexer (e.g., type, lexeme, line number).
  - UtilNodes.hpp: [If you have a good idea what this is: e.g., "Defines specialized or common utility AST node types."]
  - tempAST.hpp: [If you know: e.g., "Contains temporary or developmental AST structures that might be integrated or deprecated."]

14. Lexical Analyzer (Lexer)

The lexical analyzer, or lexer, serves as the foundational phase of our Python compiler. Its primary responsibility is to meticulously scan the input Python source code, character by character, and transform it into a structured sequence of tokens. These tokens—representing fundamental language elements such as identifiers, keywords, operators, literals, and punctuation—form the input for the subsequent parsing phase.

Our lexer employs a single-pass tokenization algorithm. This approach is designed to efficiently process the source code, identify valid lexemes according to the defined Python subset, and categorize them into distinct token types. Furthermore, the lexer addresses Python-specific features, including the syntactic significance of indentation and performs preliminary type inference to populate a symbol table, enhancing its utility for later compilation stages.

14.1. Lexer Overview and Core Functionalities

The primary objective of our lexer is to accurately identify and categorize various lexical elements within the input Python code. Its core functionalities are comprehensive, encompassing:

1. **Tokenization:**
    - **Lexeme Recognition:** Scans the input to identify valid sequences of characters (lexemes) that correspond to predefined token patterns.
    - **Token Generation:** For each recognized lexeme, a Token object is created, capturing its type (e.g., IDENTIFIER, KEYWORD_IF, INTEGER_LITERAL), the lexeme itself, its line number, and a broader category (e.g., KEYWORD, OPERATOR).
    - **Diverse Token Support:** Handles a wide range of Python tokens including keywords, identifiers, numeric literals (integers, floats, complex), string literals (single, double-quoted, byte strings), operators (arithmetic, comparison, logical, assignment), and punctuation symbols.
2. **Python-Specific Feature Handling:**
    - **Indentation Processing:** Crucially, the lexer manages Python's significant indentation. It tracks changes in indentation levels at the beginning of lines and generates special TK_INDENT and TK_DEDENT tokens, which are vital for the parser to understand block structures.
    - **Whitespace and Comment Management:** Effectively skips insignificant whitespace (spaces, tabs) and single-line comments (#), ensuring they do not interfere with token generation, while correctly processing newlines for line counting and indentation logic.

3. **Symbol Table Population and Basic Type Inference:**
    - **Symbol Table Creation:** As identifiers are encountered, they are added to a symbol table.
    - **Type Inference:** A distinctive feature of this lexer is its ability to perform basic type inference for identifiers. By analyzing assignment statements (e.g., x = 10), type hints (e.g., x: int), and the structure of literals (e.g., [1,2]), the lexer infers and records potential types (e.g., "int", "str", "list[int]", "function") for identifiers in the symbol table. This provides enriched information for subsequent analysis or IDE-like features.
4. **Lexical Error Handling and Recovery:**
    - **Error Detection:** Identifies various lexical errors, such as unterminated string literals, unknown symbols (invalid characters), unterminated triple-quoted strings, and overly long identifiers.
    - **Error Reporting:** Logs detected errors with relevant information (line number, problematic lexeme, error message).
    - **Recovery Mechanisms:** Implements strategies, notably "panic mode" for unknown symbols, to allow the lexer to recover from errors and continue processing the remainder of the input file, preventing a cascade of subsequent spurious errors.
5. **Structured Output:**
    - Produces a linear stream of Token objects for the parser.
    - Provides access to the populated symbol table.
    - Makes available a list of any lexical errors encountered.

For a comprehensive and in-depth exploration of the lexer's architecture, specific algorithms for token recognition, detailed implementation of indentation handling, exhaustive error handling strategies, and extensive code examples, please refer to our dedicated "Phase 2 - The Lexical Analyzer" report (Submitted on Friday 19 April 2025). That document provides a granular view of the lexer's internal workings and design rationale.

15. Syntax Analyzer (Parser) Design and Implementation

This section elaborates on the design and implementation of the syntax analyzer, or parser, for our Python compiler. The parser is the second major phase, taking the stream of tokens generated by the lexical analyzer as input. Its primary role is to verify that this token stream conforms to the grammatical rules of our defined Python subset and to construct a hierarchical representation of the source code, in this case, an Abstract Syntax Tree (AST).

15.1. Parser Overview and Responsibilities

The syntax analyzer is responsible for determining the grammatical structure of the token sequence provided by the lexer. It essentially checks if the sequence of tokens forms a syntactically valid program according to the language's defined grammar.

- **Input:** The parser receives a stream of tokens from the Lexer instance it is initialized with. The Lexer::nextToken() method is used internally by the parser to fetch tokens one by one as needed.
- **Syntactic Validation:** It validates whether the arrangement of tokens corresponds to valid Python language constructs such as statements (assignments, conditionals, loops), expressions (arithmetic, boolean), function definitions, class definitions, and import statements, as defined in our grammar (PYCFG.gram).
- **AST Construction:** Upon successful validation, the parser constructs an Abstract Syntax Tree (AST). Each node in this tree (AstNode) represents a construct in the source code, hierarchically organized to reflect the program's structure.
- **Error Reporting:** If the token stream violates the grammar rules, the parser identifies this as a syntax error, reports it with contextual information (line number and unexpected token), and in the current implementation, typically halts or produces a partial/null AST for the erroneous section.
- **Output:** The primary output is a std::shared_ptr<AstNode> pointing to the root of the constructed AST for a valid program. It also generates a .dot file representation of this AST for visualization. A list of encountered syntax errors is maintained and can be retrieved.

The parser implemented in Parser.cpp systematically attempts to match sequences of tokens against the production rules defined in our Python subset grammar.

15.2. Grammar Definition (Referencing PYCFG.gram)

The syntactic structure of the supported Python subset is formally defined by a Context-Free Grammar (CFG), located in the PYCFG.gram file. This grammar specifies the set of rules that dictate how valid sequences of tokens can be combined to form various language constructs.

- **Formalism:** The grammar is expressed in a notation like Backus-Naur Form (BNF), where non-terminal symbols (representing language constructs) are defined by sequences of terminal symbols (tokens from the lexer) and other non-terminal symbols.
- **Starting Rule:** The grammar parsing begins with the file rule:

  FILE: STATEMENTS_OPT TK_EOF
  STATEMENTS_OPT: STATEMENTS |

  This indicates that a valid Python file consists of an optional sequence of statements followed by an end-of-file token.

- **Key Grammar Rule:**

```
# STARTING RULES
# ==============

file: statements_opt TK_EOF
statements_opt: statements | epsilon

# GENERAL STATEMENTS
# ==================

statements: statement statement_star
statement_star: statement statement_star | epsilon

statement: compound_stmt | simple_stmts

simple_stmts: simple_stmt_list optional_semicolon
simple_stmt_list: simple_stmt simple_stmt_list_tail_star
simple_stmt_list_tail_star: TK_SEMICOLON simple_stmt simple_stmt_list_tail_star | epsilon
optional_semicolon: TK_SEMICOLON | epsilon

simple_stmt:
    | assignment
    | expressions  # Was star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | TK_PASS
    | TK_BREAK
    | TK_CONTINUE
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | for_stmt
    | try_stmt
    | while_stmt
```

Figure 28

```
# SIMPLE STATEMENTS
# =================

assignment:
    | targets TK_ASSIGN expressions        # Simplified from chained assignment and
star_targets/star_expressions
    | single_target augassign expressions  # Was star_expressions

# star_targets_eq_plus and star_targets_eq_plus_rest are removed (handled by simplification above)

augassign:
    | TK_PLUS_ASSIGN | TK_MINUS_ASSIGN | TK_MULTIPLY_ASSIGN | TK_DIVIDE_ASSIGN # TK_IMATMUL removed
    | TK_MOD_ASSIGN | TK_BIT_AND_ASSIGN | TK_BIT_OR_ASSIGN | TK_BIT_XOR_ASSIGN
    | TK_BIT_LEFT_SHIFT_ASSIGN | TK_BIT_RIGHT_SHIFT_ASSIGN | TK_POWER_ASSIGN | TK_FLOORDIV_ASSIGN

return_stmt: TK_RETURN expressions_opt # Was star_expressions_opt
expressions_opt: expressions | epsilon # Was star_expressions_opt

raise_stmt:
    | TK_RAISE expression raise_from_opt
    | TK_RAISE
raise_from_opt: TK_FROM expression | epsilon

global_stmt: TK_GLOBAL name_comma_list
nonlocal_stmt: TK_NONLOCAL name_comma_list

name_comma_list: TK_IDENTIFIER name_comma_list_tail_star
name_comma_list_tail_star: TK_COMMA TK_IDENTIFIER name_comma_list_tail_star | epsilon

import_stmt: import_name | import_from
```

Figure 29

```
# Import statements
# -----------------
# (Kept as is, considered standard)
import_name: TK_IMPORT dotted_as_names

import_from:
    | TK_FROM dot_or_ellipsis_star dotted_name TK_IMPORT import_from_targets
    | TK_FROM dot_or_ellipsis_plus TK_IMPORT import_from_targets

dot_or_ellipsis_star: dot_or_ellipsis dot_or_ellipsis_star | epsilon
dot_or_ellipsis_plus: dot_or_ellipsis dot_or_ellipsis_star
dot_or_ellipsis: TK_PERIOD

import_from_targets:
    | TK_LPAREN import_from_as_names optional_comma TK_RPAREN
    | import_from_as_names
    | TK_MULTIPLY

optional_comma: TK_COMMA | epsilon

import_from_as_names: import_from_as_name import_from_as_name_comma_list_star
import_from_as_name_comma_list_star: TK_COMMA import_from_as_name import_from_as_name_comma_list_star |
epsilon

import_from_as_name: TK_IDENTIFIER import_from_as_name_as_opt
import_from_as_name_as_opt: TK_AS TK_IDENTIFIER | epsilon

dotted_as_names: dotted_as_name dotted_as_name_comma_list_star
dotted_as_name_comma_list_star: TK_COMMA dotted_as_name dotted_as_name_comma_list_star | epsilon

dotted_as_name: dotted_name dotted_as_name_as_opt
dotted_as_name_as_opt: TK_AS TK_IDENTIFIER | epsilon

dotted_name: dotted_name TK_PERIOD TK_IDENTIFIER | TK_IDENTIFIER
```

Figure 29

```
# COMPOUND STATEMENTS
# ====================

# Common elements
# ---------------

block:
    | TK_INDENT statements TK_DEDENT
    | simple_stmts

# Class definitions
# -----------------

class_def: class_def_raw

class_def_raw: TK_CLASS TK_IDENTIFIER class_arguments_opt TK_COLON block
class_arguments_opt: TK_LPAREN arguments_opt TK_RPAREN | epsilon # arguments_opt will use simplified
'arguments'
arguments_opt: arguments | epsilon


# Function definitions
# --------------------

function_def: function_def_raw

function_def_raw: TK_DEF TK_IDENTIFIER TK_LPAREN params_opt TK_RPAREN TK_COLON block

params_opt: params | epsilon
```

Figure 30

```
# Function parameters (Simplified)
# --------------------

params: parameters

parameters: # Removed slash_no_default, slash_with_default (positional-only)
    | param_no_default_plus param_with_default_star simplified_star_etc_opt
    | param_with_default_plus simplified_star_etc_opt
    | simplified_star_etc

param_no_default_star: param_no_default param_no_default_star | epsilon
param_with_default_star: param_with_default param_with_default_star | epsilon
simplified_star_etc_opt: simplified_star_etc | epsilon

param_no_default_plus: param_no_default param_no_default_star
param_with_default_plus: param_with_default param_with_default_star

simplified_star_etc: # Simplified to remove keyword-only parameters without *args name
    | TK_MULTIPLY param_no_default kwds_opt # This is *args, **kwargs or *args
    | kwds                                 # This is **kwargs

# param_maybe_default_star, param_maybe_default_plus, param_maybe_default removed (were for keyword-only)
kwds_opt: kwds | epsilon

kwds: TK_POWER param_no_default # **kwargs

param_no_default: param param_ending_char
param_with_default: param default param_ending_char

param_ending_char: TK_COMMA | epsilon

default_opt: default | epsilon # Retained for potential future use, though param_maybe_default removed

param: TK_IDENTIFIER
default: TK_ASSIGN expression
```

Figure 31

```
# If statement
# ------------

if_stmt: # named_expression replaced by expression
    | TK_IF expression TK_COLON block elif_stmt
    | TK_IF expression TK_COLON block else_block_opt

elif_stmt: # named_expression replaced by expression
    | TK_ELIF expression TK_COLON block elif_stmt
    | TK_ELIF expression TK_COLON block else_block_opt

else_block_opt: else_block | epsilon
else_block: TK_ELSE TK_COLON block

# While statement
# ---------------

while_stmt: TK_WHILE expression TK_COLON block else_block_opt # named_expression replaced by expression

# For statement
# -------------
for_stmt: TK_FOR targets TK_IN expressions TK_COLON block else_block_opt # star_targets -> targets,
star_expressions -> expressions

# Try statement
# -------------
try_stmt: # Removed except_star_block_plus (exception groups)
    | TK_TRY TK_COLON block finally_block
    | TK_TRY TK_COLON block except_block_plus else_block_opt finally_block_opt

except_block_plus: except_block except_block_plus_star
except_block_plus_star: except_block except_block_plus_star | epsilon

# except_star_block_plus and related rules removed.

finally_block_opt: finally_block | epsilon

# Except statement
# ----------------

except_block:
    | TK_EXCEPT expression except_as_name_opt TK_COLON block
    | TK_EXCEPT TK_COLON block
except_as_name_opt: TK_AS TK_IDENTIFIER | epsilon

# except_star_block removed.
finally_block: TK_FINALLY TK_COLON block
```

Figure 32

```
# EXPRESSIONS
# -----------

expressions:
    | expression expression_comma_plus optional_comma
    | expression TK_COMMA
    | expression
expression_comma_plus: TK_COMMA expression expression_comma_plus_star
expression_comma_plus_star: TK_COMMA expression expression_comma_plus_star | epsilon


expression:
    | disjunction TK_IF disjunction TK_ELSE expression
    | disjunction


disjunction: conjunction disjunction_tail_star
disjunction_tail_star: TK_OR conjunction disjunction_tail_star | epsilon

conjunction: inversion conjunction_tail_star
conjunction_tail_star: TK_AND inversion conjunction_tail_star | epsilon

inversion: TK_NOT inversion | comparison
```

Figure 33

```
comparison: bitwise_or compare_op_bitwise_or_pair_star
compare_op_bitwise_or_pair_star: compare_op_bitwise_or_pair compare_op_bitwise_or_pair_star | epsilon

compare_op_bitwise_or_pair:
    | eq_bitwise_or | noteq_bitwise_or | lte_bitwise_or | lt_bitwise_or
    | gte_bitwise_or | gt_bitwise_or | notin_bitwise_or | in_bitwise_or
    | isnot_bitwise_or | is_bitwise_or

eq_bitwise_or: TK_EQUAL bitwise_or
noteq_bitwise_or: TK_NOT_EQUAL bitwise_or
lte_bitwise_or: TK_LESS_EQUAL bitwise_or
lt_bitwise_or: TK_LESS bitwise_or
gte_bitwise_or: TK_GREATER_EQUAL bitwise_or
gt_bitwise_or: TK_GREATER bitwise_or
notin_bitwise_or: TK_NOT TK_IN bitwise_or
in_bitwise_or: TK_IN bitwise_or
isnot_bitwise_or: TK_IS TK_NOT bitwise_or
is_bitwise_or: TK_IS bitwise_or

bitwise_or: bitwise_or TK_BIT_OR bitwise_xor | bitwise_xor
bitwise_xor: bitwise_xor TK_BIT_XOR bitwise_and | bitwise_and
bitwise_and: bitwise_and TK_BIT_AND shift_expr | shift_expr
shift_expr: shift_expr TK_BIT_LEFT_SHIFT sum | shift_expr TK_BIT_RIGHT_SHIFT sum | sum

sum: sum TK_PLUS term | sum TK_MINUS term | term
term: term TK_MULTIPLY factor | term TK_DIVIDE factor | term TK_FLOORDIV factor | term TK_MOD factor |
factor # TK_MATMUL factor removed
factor: TK_PLUS factor | TK_MINUS factor | TK_BIT_NOT factor | power
```

Figure 34

```
power: primary TK_POWER factor | primary

primary:
    | primary TK_PERIOD TK_IDENTIFIER
    | primary TK_LPAREN arguments_opt TK_RPAREN # arguments_opt uses simplified 'arguments'
    | primary TK_LBRACKET slices TK_RBRACKET
    | atom

slices:
    | slice
    | slice_or_expr_comma_list optional_comma # Was slice_or_starred_expr_comma_list

slice_or_expr_comma_list: slice_or_expr slice_or_expr_comma_list_tail_star # Was slice_or_starred_expr...
slice_or_expr_comma_list_tail_star: TK_COMMA slice_or_expr slice_or_expr_comma_list_tail_star | epsilon #
Was slice_or_starred_expr...
slice_or_expr: slice | expression # Was starred_expression, now simplified to expression

slice: expression_opt TK_COLON expression_opt slice_colon_expr_opt | expression # Was named_expression
expression_opt: expression | epsilon
slice_colon_expr_opt: TK_COLON expression_opt | epsilon

atom:
    | TK_IDENTIFIER
    | TK_TRUE | TK_FALSE | TK_NONE
    | strings
    | TK_NUMBER
    | TK_STR | TK_INT | TK_FLOAT | TK_COMPLEX | TK_LIST | TK_TUPLE | TK_RANGE
    | TK_DICT | TK_SET | TK_FROZENSET | TK_BOOL | TK_BYTES
    | TK_BYTEARRAY | TK_MEMORYVIEW | TK_NONETYPE
    | tuple_group_variant # Was tuple_group_genexp_variant, genexp part implicitly removed with
comprehensions
    | list_variant        # Was list_listcomp_variant
    | dict_set_variant    # Was dict_set_comp_variant
```

Figure 35

```
tuple_group_variant: tuple | group # genexp removed
list_variant: list               # listcomp removed
dict_set_variant: dict | set     # dictcomp and setcomp removed

group: TK_LPAREN expression TK_RPAREN # Was named_expression

# LITERALS
# ========

string: TK_STRING

strings: fstring_or_string_plus
fstring_or_string_plus: fstring_or_string fstring_or_string_plus_star
fstring_or_string_plus_star: fstring_or_string fstring_or_string_plus_star | epsilon
fstring_or_string: string | TK_BYTES # Assuming f-strings are handled as TK_STRING or a variant not
detailed here

list: TK_LBRACKET expressions_opt TK_RBRACKET # Was star_named_expressions_opt_for_collections
# star_named_expressions_opt_for_collections rule removed.

tuple: TK_LPAREN tuple_content_opt TK_RPAREN
tuple_content_opt: expression TK_COMMA expressions_opt | epsilon # Was star_named_expression and
star_named_expressions_opt_for_collections

set: TK_LBRACE expressions TK_RBRACE # Was star_named_expressions

dict: TK_LBRACE kvpairs_opt TK_RBRACE # Was double_starred_kvpairs_opt
kvpairs_opt: kvpairs | epsilon       # Was double_starred_kvpairs_opt

kvpairs: kvpair_comma_list optional_comma # Was double_starred_kvpairs
kvpair_comma_list: kvpair kvpair_comma_list_tail_star # Was double_starred_kvpair...
kvpair_comma_list_tail_star: TK_COMMA kvpair kvpair_comma_list_tail_star | epsilon # Was
double_starred_kvpair...
```

Figure 36

```
# double_starred_kvpair simplified to kvpair as TK_POWER bitwise_or (for **d) is removed.
kvpair: expression TK_COLON expression

arguments: args optional_comma

args:
    | positional_arguments_list TK_COMMA keyword_arguments_list
    | positional_arguments_list
    | keyword_arguments_list
    | epsilon # To allow f()

positional_arguments_list: expression positional_arguments_list_tail_star
positional_arguments_list_tail_star: TK_COMMA expression positional_arguments_list_tail_star | epsilon

keyword_arguments_list: keyword_item keyword_arguments_list_tail_star
keyword_arguments_list_tail_star: TK_COMMA keyword_item keyword_arguments_list_tail_star | epsilon

keyword_item: TK_IDENTIFIER TK_ASSIGN expression
```

Figure 37

```
# ASSIGNMENT TARGETS (Simplified: no *target)
# ==================

targets: # Was star_targets
    | target
    | target target_comma_list_star optional_comma
target_comma_list_star: TK_COMMA target target_comma_list_star | epsilon # Was
star_target_comma_list_star

targets_list_seq: target_comma_list optional_comma # Was star_targets_list_seq
target_comma_list: target target_comma_list_tail_star # Was star_target_comma_list
target_comma_list_tail_star: TK_COMMA target target_comma_list_tail_star | epsilon # Was
star_target_comma_list_tail_star

targets_tuple_seq: # Was star_targets_tuple_seq
    | target target_comma_list_plus optional_comma
    | target TK_COMMA
target_comma_list_plus: TK_COMMA target target_comma_list_star # Was star_target_comma_list_plus


target: # Was star_target, TK_MULTIPLY unstarred_star_target option removed
    | t_primary TK_PERIOD TK_IDENTIFIER
    | t_primary TK_LBRACKET slices TK_RBRACKET
    | target_atom # Was star_atom

target_atom: # Was star_atom, simplified to remove internal packing/unpacking targets
    | TK_IDENTIFIER
    | TK_LPAREN target TK_RPAREN # Was target_with_star_atom
    | TK_LPAREN targets_tuple_seq_opt TK_RPAREN # Was star_targets_tuple_seq_opt
    | TK_LBRACKET targets_list_seq_opt TK_RBRACKET # Was star_targets_list_seq_opt
targets_tuple_seq_opt: targets_tuple_seq | epsilon # Was star_targets_tuple_seq_opt
targets_list_seq_opt: targets_list_seq | epsilon # Was star_targets_list_seq_opt

single_target:
    | single_subscript_attribute_target
    | TK_IDENTIFIER
    | TK_LPAREN single_target TK_RPAREN

single_subscript_attribute_target:
    | t_primary TK_PERIOD TK_IDENTIFIER
    | t_primary TK_LBRACKET slices TK_RBRACKET

t_primary:
    | t_primary TK_PERIOD TK_IDENTIFIER
    | t_primary TK_LBRACKET slices TK_RBRACKET
    | t_primary TK_LPAREN arguments_opt TK_RPAREN # arguments_opt uses simplified 'arguments'
    | atom

# ======================== END OF THE MODIFIED GRAMMAR ==========================
```

Figure 38

15.3. Parsing Technique

The parser employs a **Recursive Descent Parsing (RDP)** strategy. This is a top-down parsing technique where a set of mutually recursive procedures is used to process the input.

- **Mapping Grammar to Functions:** Each significant non-terminal symbol in our PYCFG.gram (e.g., statement, expression, function_def, if_stmt) typically corresponds to a dedicated parsing method in the Parser class (e.g., parseStatement(), parseExpression(), parseFunctionDef(), parseIfStmt()).
- **Top-Down Approach:** The parsing process begins with the method corresponding to the grammar's start symbol (parseFile() which calls parseStatementsOpt()). This method then calls other parsing methods based on the expected tokens and grammar rules.
- **Token Consumption:** The parser maintains a currentToken (of type Token). Helper methods like advance() are used to consume the current token and fetch the next one from the lexer. The match(TokenType) function checks if the currentToken matches an expected type, and expect(TokenType, errorMsg) attempts to match and advance, reporting an error if the match fails.
- **Lookahead:** The parser primarily operates with a lookahead of one token (LL(1) characteristics in many parts). Decisions on which production rule to apply are often made based on the current token. In some specific cases within parseSimpleStmt() and parseAssignment(), a limited form of two-token lookahead is simulated by peeking at the nextToken from the lexer to differentiate between assignment and expression statements, or different forms of assignments. This is managed by temporarily advancing and then "backtracking" the lexer's token stream if the initial assumption based on the first token was incorrect. (This "backtracking" with lexer.tokens.push_back() and pop_back() is a less common RDP pattern and indicates a specific way of handling grammar ambiguities or left-factoring issues).

The choice of Recursive Descent Parsing offers a relatively straightforward implementation that closely mirrors the structure of the CFG, making the parser's logic traceable to the grammar rules.

15.4. Parser Class Design (Parser.hpp, Parser.cpp)

The entire syntax analysis process is orchestrated and managed by the Parser class. This class serves as the central engine for interpreting the token stream provided by the lexer, validating it against the defined Python subset grammar, constructing an Abstract Syntax Tree (AST) to represent the program's hierarchical structure, and reporting any syntactic errors encountered. The design emphasizes a modular approach through recursive descent.

15.4.1. Class Declaration and Member Variables (Parser.hpp)

The Parser.hpp file declares the Parser class, outlining its public interface and the private member variables essential for its operation:

```
class Parser {
public:
    explicit Parser(Lexer& lexer_instance);

    std::unique_ptr<ProgramNode> parse();


    const std::vector<std::string>& getErrors() const { return errors_list;
}   bool hasError() const { return had_error; }

private:
    Lexer& lexer_ref;
    std::vector<Token> tokens;
    size_t current_pos;
    bool had_error;
    std::vector<std::string> errors_list;
    static Token eof_token;
    Token& peek(int offset = 0);
    Token& previous();
    bool isAtEnd(int offset = 0);
    Token advance();
    bool check(TokenType type) const;
    bool check(const std::vector<TokenType>& types) const;
    bool match(TokenType type);
    Token consume(TokenType type, const std::string& message);
    void reportError(const Token& token, const std::string& message); /
    void synchronize();
    void unputToken();

    std::unique_ptr<ProgramNode> parseFile();
    std::vector<std::unique_ptr<StatementNode>> parseStatementsOpt();
    std::vector<std::unique_ptr<StatementNode>> parseStatements();
    std::unique_ptr<StatementNode> parseStatement();
    std::unique_ptr<StatementNode> parseSimpleStmt();
    std::unique_ptr<StatementNode> parseCompoundStmt();
    std::unique_ptr<ExpressionNode> parseExpression();
    std::unique_ptr<ExpressionNode> parseDisjunction();
    std::unique_ptr<IfStatementNode> parseIfStmt();
    std::unique_ptr<FunctionDefinitionNode> parseFunctionDef();
```

Figure 39

- Lexer& lexer_ref (Private Member): A reference to the Lexer instance. In this version, the constructor pre-fetches all tokens from the lexer.
- std::vector<Token> tokens (Private Member): A local copy of the entire token stream obtained from the lexer_ref during parser construction. The parser operates on this vector.
- size_t current_pos (Private Member): An index indicating the current position within the tokens vector. This replaces directly calling lexer.nextToken() repeatedly during parsing.
- bool had_error (Private Member): A flag set to true if any lexer error was propagated or if a parser error occurs. This can be used to gate further processing.
- std::vector<std::string> errors_list (Private Member): Stores formatted error messages from both propagated lexer errors and syntax errors detected by the parser.
- static Token eof_token (Private Member): A static member representing an EOF token, used for boundary checks and as a return value for peek() when out of bounds.
- Constructor Parser(Lexer& lexer_instance):
    - Consumes all tokens from the lexer_instance by repeatedly calling lexer_ref.nextToken() until TK_EOF.
    - Propagates any errors reported by the lexer into its own errors_list and sets had_error.
    - Calls lexer_ref.processIdentifierTypes() to allow the lexer to perform its type inference pass on the complete token stream.
    - Copies the processed tokens (including INDENT/DEDENT and lexer-inferred types) into its local this->tokens vector.
    - If lexer errors occurred, it effectively prepares an empty token stream (just EOF) to prevent parsing attempts on erroneous input.
- parse() (Public Method):
    - The main entry point to begin parsing.
    - Checks if had_error from the lexer phase or if the token stream is essentially empty. If so, returns an empty ProgramNode to signify failure.
    - Resets current_pos to 0.
    - Calls parseFile() to start the recursive descent process from the grammar's top-level rule.
- getErrors() const and hasError() const (Public Methods): Provide external access to the accumulated error messages and the overall error status.

15.4.2. Core Operational Logic and Helper Methods (Parser.cpp)

- The implementation in Parser.cpp defines the operational heart of the syntax analyzer. It relies on a set of core helper methods for managing the pre-fetched token stream and a large suite of recursive functions to parse specific grammatical constructs.

A. Token Stream Management and Querying (Private Helpers):

These methods provide the parser with controlled access to the tokens vector, which is populated during the parser's construction phase.

- **Token& peek(int offset = 0):** This crucial lookahead function allows the parser to inspect tokens at the current position (current_pos) or at a specified offset from it without advancing the current_pos. It returns a reference to the static eof_token if the requested position is out of bounds. This is fundamental for predictive parsing decisions.

```
Token& Parser::peek(int offset) {
    if (current_pos + offset >= tokens.size()) {
        return eof_token; // Static EOF token for boundary conditions
    }
    return tokens[current_pos + offset];
}
```

Figure 40

- **Token Parser::advance():** This method "consumes" the current token by incrementing current_pos. It returns the token that *was* current before the advance (i.e., tokens[current_pos - 1]).
- **bool Parser::isAtEnd(int offset = 0):** Checks if the parser has reached the end of the token stream (or would be at the end if advanced by offset), or if the peeked token is TK_EOF.
- **bool Parser::check(TokenType type) const and bool Parser::check(const std::vector<TokenType>& types) const:** These non-consuming methods test whether the token at current_pos matches a specific TokenType or any type within a given vector of types, respectively. They are used extensively for making decisions in parsing functions.
- **Token& Parser::previous():** Returns the token that was just consumed by advance(). Useful for associating an AST node with the primary token that triggered its rule.

B. Token Consumption and Rule Enforcement (Private Helpers):

- **bool Parser::match(TokenType type):** A conditional consumption method. If the currentToken (obtained via peek()) matches the specified type, advance() is called to consume it, and true is returned. Otherwise, false is returned without advancing. This is ideal for optional parts of grammar rules or for choosing between alternative productions.
- **Token Parser::consume(TokenType type, const std::string& message):** This method enforces that the currentToken *must* be of the specified type. If it is, the token is consumed via advance(), and the consumed token is returned. If not, reportError() is called with the provided message, and a std::runtime_error is thrown. This hard-checks mandatory tokens in grammar productions.

```
Token Parser::consume(TokenType type, const string& message) {
    if (check(type)) {
        return advance(); // Advance and return the consumed token
    }
    reportError(peek(), message); // Report error with the current unexpected token
    throw runtime_error("Parse error: " + message); // Halt current parsing path
}
```

Figure 41

C. Error Handling and Synchronization (Private Helpers):

- **void Parser::reportError(const Token& token, const std::string& message):** This function is invoked when a syntax error is detected. It sets the had_error flag to true and appends a formatted error message (including the line number and lexeme of the offending token) to the errors_list vector.
- **void Parser::synchronize():** Implements a "panic mode" error recovery strategy. When an exception (typically from consume()) is caught in a higher-level parsing loop (like parseStatements()), synchronize() is called. It discards tokens by repeatedly calling advance() until it encounters a token that is likely to start a new, valid statement (e.g., common keywords like def, if, class, return, or structural tokens like TK_DEDENT, TK_SEMICOLON). This allows the parser to attempt to resume parsing after an error and potentially find subsequent errors in the same pass.

```
void Parser::synchronize() {
    if (isAtEnd()) return;
    advance(); // Consume the token that caused the error.

    while (!isAtEnd()) {
        if (previous().type == TokenType::TK_SEMICOLON) return;
        if (peek().type == TokenType::TK_DEDENT) return;

        switch (peek().type) {
            case TokenType::TK_CLASS:
            case TokenType::TK_DEF:
            // ... other statement-starting keywords ...
            case TokenType::TK_RETURN:
                return;
            default:
                // Additional checks like TK_INDENT could be here
                ; // No action, just advance
        }
        advance();
    }
}
```

Figure 42

D. Recursive Descent Parsing Logic (parse<NonTerminalName>() methods):

The core of the parser lies in its numerous parse<NonTerminalName>() methods, each designed to recognize and construct an AST for a specific non-terminal symbol from the PYCFG.gram. These methods are public in Parser.hpp but primarily function as a set of mutually recursive procedures.

- **General Structure:**
    1. **Prediction/Choice:** The method often starts by peek()-ing at the currentToken (and sometimes subsequent tokens using peek(offset)) or using check() to decide which production rule of the non-terminal to apply.
    2. **Terminal Consumption:** Mandatory terminal tokens are consumed using consume(). Optional terminals are handled with match().
    3. **Non-Terminal Recursion:** For non-terminal symbols on the right-hand side of a production, the corresponding parse<OtherNonTerminal>() method is called.
    4. **AST Node Creation:** Upon successful recognition of a rule, an appropriate AST node is created using std::make_unique<SpecificAstNodeType>(...). The constructor is usually passed the line number (from a significant token) and any parsed child nodes or literal values.
    5. **Looping for * and + Productions:** Grammar rules involving Kleene star (zero or more occurrences) or Kleene plus (one or more occurrences) are typically

implemented using while or do-while loops that repeatedly call parsing methods for the recurring element, often conditioned by match() or check().

- **Example: parseIfStmt() (Illustrative of Compound Statements):**
  This method handles if-elif-else structures. It consume()s TK_IF, recursively calls parseExpression() for the condition, consume()s TK_COLON, calls parseBlock() for the body, and then uses peek() and match() to handle optional elif clauses (recursively calling itself or a similar parseElifBlock) and an optional else clause (calling parseElseBlockOpt()). An IfStatementNode is constructed with these components.
  - *Screenshot Suggestion 4: A portion of parseIfStmt() showing consume, recursive calls like parseExpression and parseBlock, and AST node creation.*
- **Example: parseSum() (Illustrative of Expression Parsing with Precedence/Associativity):**
  Methods like parseSum(), parseTerm(), parseFactor(), etc., implement the precedence and associativity of operators. parseSum() (for +, -) typically calls parseTerm() to get a higher-precedence operand, then enters a while loop that match()es TK_PLUS or TK_MINUS. Inside the loop, it consumes the operator, recursively calls parseTerm() for the right-hand operand, and constructs a BinaryOpNode, reassigning the result to the left operand to handle left-associativity.
  - *Screenshot Suggestion 5: The parseSum() or parseTerm() method, demonstrating the handling of binary operators and recursive calls for operands.*
- **Example: parseSimpleStmt() (Illustrative of Ambiguity Handling/Multi-Attempt Parsing):**
  This complex method demonstrates how the parser differentiates between various simple statements that might start similarly (e.g., an identifier could begin an assignment, an augmented assignment, or an expression statement). It employs a multi-attempt strategy:
  1. It first attempts to parse as targets TK_ASSIGN expressions.
  2. If that fails or isn't applicable, it backtracks (current_pos = initial_pos; and error state restoration) and attempts to parse as single_target augassign expressions.
  3. If that also fails, it backtracks again and finally attempts to parse it as a general expressions (which becomes an ExpressionStatementNode).
     This backtracking, involving saving and restoring current_pos and the error state (had_error, errors_list.size()), is a key technique for handling such ambiguities in a recursive descent parser.

E. Constructor Logic (Parser::Parser(...)):

The constructor plays a vital role by pre-processing the entire token stream from the lexer:

1. It iteratively calls lexer_ref.nextToken() until TK_EOF, accumulating all tokens.
2. During this process, it checks for and propagates any errors reported by the lexer_ref into its own errors_list, setting had_error if lexer errors are found.
3. It ensures a final TK_EOF token is present.
4. Crucially, it then calls lexer_ref.processIdentifierTypes(). This allows the lexer to perform its type inference pass *after* all tokens (including INDENT/DEDENT) have been generated and are available for contextual analysis.
5. Finally, it copies the (potentially type-annotated by the lexer) tokens from lexer_ref.tokens into its local this->tokens vector.
6. If lexer errors were detected, it effectively prepares a minimal token stream (just EOF) to prevent the parser from attempting to process a known-faulty input.

```cpp
std::unique_ptr<ExpressionNode> Parser::parseAtom(bool in_target_context) {
    int line = peek().line;
    switch (peek().type) {
        case TokenType::TK_IDENTIFIER: {
            Token id_token = advance();
            return make_unique<IdentifierNode>(id_token.line, id_token.lexeme);
        }
        case TokenType::TK_TRUE:  advance(); return make_unique<BooleanLiteralNode>(line, true);
        case TokenType::TK_FALSE: advance(); return make_unique<BooleanLiteralNode>(line, false);
        case TokenType::TK_NUMBER: {
            Token num_token = advance();
            // ... logic to determine if INTEGER or FLOAT ...
            NumberLiteralNode::Type num_type = /* ... */;
            return make_unique<NumberLiteralNode>(line, num_token.lexeme, num_type);
        }
        case TokenType::TK_STRING:
            return parseStrings(); // Handles concatenation
        case TokenType::TK_LPAREN:
            return parseTupleGroupVariant(in_target_context);
        // ... other cases ...
        default:
            reportError(peek(), "Expected an atom (identifier, literal, '(', '[', or '{').");
            throw runtime_error("Invalid atom...");
    }
}
```

Figure 43

```cpp
unique_ptr<BlockNode> Parser::parseBlock() {
    int line = peek().line; // Line where the block effectively starts or would start
    if (match(TokenType::TK_INDENT)) {
        auto stmts = parseStatements(); // Recursively parse statements within the block
        consume(TokenType::TK_DEDENT, "Expected DEDENT to end indented block.");
        return make_unique<BlockNode>(line, std::move(stmts));
    } else {
        auto stmts_vec = parseSimpleStmts(); // Parses one or more simple statements on current line
        return make_unique<BlockNode>(line, std::move(stmts_vec));
    }
}
```

Figure 44

15.5. Abstract Syntax Tree (AST) Construction

Upon successfully parsing a sequence of tokens that conforms to the defined grammar, the parser constructs an Abstract Syntax Tree (AST). The AST serves as an intermediate, hierarchical representation of the source code's structure, abstracting away much of the concrete syntax (like parentheses for grouping or specific keyword tokens once their purpose is captured by a node type). This tree is then amenable to further processing, such as semantic analysis, code generation, or, in this project, visualization.

15.5.1. AST Node Hierarchy (ASTNode.hpp, Statements.hpp, Expressions.hpp, UtilNodes.hpp)

The AST is composed of various types of nodes, each representing a specific element of the Python language subset. A rich class hierarchy is defined for these nodes, with ASTNode (from ASTNode.hpp) serving as the polymorphic base class.

```
class ASTNode {
public:
    int line; // Line number from the source, for error reporting & debugging

    ASTNode(int line) : line(line) {}
    virtual ~ASTNode() = default;

    virtual void accept(ASTVisitor* visitor) = 0; // For the Visitor pattern
    virtual std::string getNodeName() const = 0;  // For
};bugging/visualization
```

Figure 45

- All specific AST nodes inherit from ASTNode and must implement the accept method (for the Visitor design pattern, enabling operations to be performed on the tree without modifying node classes) and getNodeName (useful for debugging or generating tree visualizations like DOT files). The line member stores the source line number, critical for error messages.
- **Specialized Node Categories and Examples:**
  The AST nodes are organized into logical categories, typically defined in separate header files:
  - **Expressions.hpp**: Defines nodes for various expression types:
    - IdentifierNode: Represents an identifier (e.g., variable name). Stores std::string name.

        - NumberLiteralNode, StringLiteralNode, BooleanLiteralNode, NoneLiteralNode, ComplexLiteralNode, BytesLiteralNode: Represent different types of literal values.
        - BinaryOpNode: Represents binary operations (e.g., a + b). Stores left operand, operator token (Token op), and right operand.
        - UnaryOpNode: Represents unary operations (e.g., -x, not flag). Stores operator token and operand.
        - FunctionCallNode: Represents a function call. Stores the callee expression, positional arguments, and keyword arguments.
        - AttributeAccessNode: For object.attribute.
        - SubscriptionNode: For object[index_or_slice].
        - IfExpNode: For ternary value_if_true if condition else value_if_false.
        - ComparisonNode: For chained comparisons like a < b <= c.
        - SliceNode: For start:stop:step slice objects.
        - ListLiteralNode, TupleLiteralNode, DictLiteralNode, SetLiteralNode: For collection literals.
    - **Statements.hpp**: Defines nodes for statement types:
        - ProgramNode: The root of the AST, holding a list of top-level statements.
        - BlockNode: Represents a sequence of statements, typically an indented block.
        - AssignmentStatementNode: For target(s) = value.
        - AugAssignNode: For augmented assignments like x += 1.
        - IfStatementNode: Represents if-elif-else structures. Stores condition, then-block, elif blocks, and an optional else-block.
        - WhileStatementNode, ForStatementNode: Represent loop constructs.
        - FunctionDefinitionNode, ClassDefinitionNode: Represent definitions.
        - ReturnStatementNode, PassStatementNode, BreakStatementNode, ContinueStatementNode, RaiseStatementNode.
        - ImportStatementNode, ImportFromStatementNode.

        - GlobalStatementNode, NonlocalStatementNode.
        - TryStatementNode: Holds the try block, exception handlers, and optional else/finally blocks.
    - **UtilNodes.hpp**: Defines helper/utility nodes primarily used within other complex nodes:
        - ParameterNode: Represents a single parameter in a function definition (name, kind, optional default value).
        - ArgumentsNode: Encapsulates the entire argument specification for a function definition (positional, vararg, kwarg).
        - (From Helpers.hpp but often grouped with util nodes) KeywordArgNode: For name=value pairs in function calls or class definitions.
        - (From Statements.hpp but utility-like) ExceptionHandlerNode, NamedImportNode, ImportNameNode.

Each specialized node class (e.g., IfStatementNode) inherits from ASTNode (or an intermediate base like StatementNode or ExpressionNode) and contains members specific to the construct it represents (e.g., condition, then_block, else_block for IfStatementNode). Child nodes are typically stored as std::unique_ptr<SpecificAstNodeType> to manage memory automatically and represent the tree's hierarchical structure.

15.5.2. Tree Building Process (Parser.cpp)

The AST is constructed dynamically by the recursive descent parsing methods in Parser (1).cpp. As each method successfully recognizes a grammatical rule (a non-terminal):

1. **Node Instantiation:** A new AST node of the appropriate specialized type is created using std::make_unique<SpecificAstNodeType>(...). The constructor is passed the source line number (usually from a key token of the construct) and any already parsed child nodes or literal values.
2. **Populating Node Members:** The specific fields of the AST node are populated. For instance, an IdentifierNode's name field is set, a BinaryOpNode stores its left operand, op token, and right operand.
3. **Recursive Construction for Children:** Parsing methods for sub-constructs (e.g., parseExpression() for an if-statement's condition, parseBlock() for its body) are called. The std::unique_ptr<ASTNode> (or more specific type) returned by these calls are then std::moved into the appropriate member of the parent node (e.g., if_node->condition = std::move(parsed_condition_node);).

4. **Collections of Children:** For constructs that can have multiple children of the same type (e.g., statements in a BlockNode, elements in a ListLiteralNode), these are collected into a std::vector<std::unique_ptr<SpecificAstNodeType>> within the parent node.

This bottom-up construction (where leaf nodes are created first by lower-level parsing functions, and then aggregated by higher-level functions) continues until the entire valid input is parsed into a single ProgramNode representing the root of the AST.

```
class ASTVisitor;

class ASTNode {
public:
    int line;

    ASTNode(int line) : line(line) {}
    virtual ~ASTNode() = default;

    virtual void accept(ASTVisitor* visitor) = 0;
    virtual std::string getNodeName() const = 0;
};
```

Figure 46

```
unique_ptr<ExpressionNode> Parser::parseSum() {
    auto node = parseTerm();
    while (match(TokenType::TK_PLUS) || match(TokenType::TK_MINUS)) {
        Token op_token = previous();
        auto right_operand = parseTerm();
        node = make_unique<BinaryOpNode>(op_token.line,
                                         std::move(node),
                                         op_token,
                                         std::move(right_operand));
    }
    return node;
}
```

Figure 47

16. Syntax Error Handling and Reporting

The parser is designed to detect deviations from the defined grammar (PYCFG.gram) and report these as syntax errors. It also incorporates a basic error recovery mechanism to attempt parsing beyond the initial error.

16.1. Error Detection Mechanisms

Syntax errors are primarily detected in two ways:

1. **Mismatched Expected Tokens (consume method):**
   The most common point of error detection is within the Token Parser::consume(TokenType type, const std::string& message) method. This method is called when a specific token type is grammatically required at the current parsing position.
   - If check(type) (which inspects peek().type) confirms the current token is of the expected type, the token is consumed via advance(), and the method returns successfully.
   - However, if check(type) fails, it signifies a syntax error: the parser encountered a token different from what the grammar rule dictates. In this scenario, consume() immediately calls reportError() and then throws a std::runtime_error.

```
Token Parser::consume(TokenType type, const string& message) {
    if (check(type)) {
        return advance();
    }
    reportError(peek(), message); // Report error using the unexpected token
    throw runtime_error("Parse error: " + message); // Signal error to
}nwind
```

Figure 48

Failed Semantic or Structural Predicates (Direct reportError calls):
In some parsing functions, after successfully matching a sequence of tokens, further structural or semantic (though primarily structural at this phase) conditions might need to be met. If these conditions fail, the parsing function may directly call reportError().

16.2. Error Reporting (reportError method)

When a syntax error is identified, the void Parser::reportError(const Token& token, const std::string& message) method is responsible for formatting and storing the error information.

```
void Parser::reportError(const Token& token, const string& message) {
    had_error = true; // Set the global error flag for the parser
    if (token.type == TokenType::TK_EOF) {
        errors_list.push_back("[line " + to_string(token.line) + "] Error at end: " + message);
    } else {
        // Include line number, the problematic lexeme, and a descriptive message
        errors_list.push_back("[line " + to_string(token.line) + "] Error at '" + token.lexeme + "': " +
message);
    }
}
```

Figure 49

Key aspects of the error reporting include:

- **had_error Flag:** A boolean member had_error in the Parser class is set to true. This flag can be queried after parsing to determine if any errors (including propagated lexer errors) occurred.
- **errors_list Vector:** A std::vector<std::string> named errors_list stores all formatted error messages.
- **Message Content:** Each error message typically includes:
    - The **line number** where the error was detected (obtained from token.line).
    - The **lexeme** of the token that caused the error (if not TK_EOF), providing concrete context.
    - A **descriptive message** (passed to reportError or generated by consume) explaining what was expected or what went wrong.
- **EOF Handling:** Special formatting is used if the error occurs at the end of the file.

16.3. Error Propagation and Recovery Control Flow

The parser uses an exception-based mechanism to handle the immediate aftermath of an error and to enable recovery:

1. **Throwing an Exception:** As seen in consume(), when an unrecoverable mismatch occurs for a mandatory token, a std::runtime_error is thrown. This causes the current chain of recursive parsing calls to unwind.
2. **Catching Exceptions and Invoking Recovery:**
   Higher-level parsing loops, particularly parseStatements(), are structured to catch these std::runtime_error exceptions.

```
vector<unique_ptr<StatementNode>> Parser::parseStatements() {
    vector<unique_ptr<StatementNode>> stmts_list;
    while (!isAtEnd() && peek().type != TokenType::TK_EOF && peek().type != TokenType::TK_DEDENT) {
        try {
            stmts_list.push_back(parseStatement());
        } catch (const runtime_error& e) {
            // Exception caught, indicating a syntax error in parseStatement() or its descendants
            synchronize(); // Attempt to recover by finding the next likely statement start
            // After synchronization, the loop will re-evaluate its condition
            if (isAtEnd() || peek().type == TokenType::TK_EOF || peek().type == TokenType::TK_DEDENT) {
                break; // Stop if synchronization lands at an end-of-block/file
            }
        }
    }
    return stmts_list;
}
```

Figure 50

When an exception is caught, indicating a syntax error within the parsing of a statement (or its sub-constructs), the synchronize() method is immediately called.

16.4. Error Recovery Strategy (synchronize method)

The void Parser::synchronize() method implements a "panic mode" error recovery strategy. The goal of panic mode is to discard tokens from the input stream until the parser finds a point where it believes it can safely resume parsing, typically the beginning of a new, independent syntactic construct.

```cpp
void Parser::synchronize() {
    if (isAtEnd()) return; // Nothing to synchronize if already at the end
    advance(); // Consume the token that originally caused the error.

    while (!isAtEnd()) {
        // Check for tokens that reliably end a previous statement/block
        if (previous().type == TokenType::TK_SEMICOLON) return;
        if (peek().type == TokenType::TK_DEDENT) return; // End of an indented block

        // Check for tokens that reliably start a new statement
        switch (peek().type) {
            case TokenType::TK_CLASS:
            case TokenType::TK_DEF:
            case TokenType::TK_IF:
            case TokenType::TK_FOR:
            case TokenType::TK_WHILE:
            case TokenType::TK_TRY:
            // case TokenType::TK_WITH: // If supported
            case TokenType::TK_RETURN:
            case TokenType::TK_IMPORT:
            // case TokenType::TK_FROM: // More complex, might need further logic
            case TokenType::TK_GLOBAL:
            case TokenType::TK_NONLOCAL:
            case TokenType::TK_PASS:
            case TokenType::TK_BREAK:
            case TokenType::TK_CONTINUE:
            case TokenType::TK_RAISE:
                return; // Found a good synchronization point
            default:
                // A new indentation level might also signal a new logical block start,
                // though DEDENT is a stronger signal for *ending* the previous one.
                if (peek().type == TokenType::TK_INDENT) return;
        }
```

Figure 51

This strategy helps in:

- **Preventing Cascading Errors:** Without recovery, a single syntax error could cause a cascade of spurious subsequent errors as the parser remains out of sync.
- **Reporting Multiple Errors:** By attempting to resynchronize, the parser can continue its pass over the remaining tokens and potentially identify other independent syntax errors in the code.

While panic mode is a relatively simple recovery technique and may discard valid code segments between the error point and the synchronization point, it is effective in allowing the parser to continue providing feedback beyond the first error encountered. The overall parsing success is still indicated by the had_error flag.

17. GUI Implementation

During the syntax analysis phase of the compiler, the graphical user interface (GUI) was upgraded to allow users to execute the parser on a stream of tokenized source code which was output by the lexer and visualize the resulting abstract syntax tree (AST).

17.1. Parser Integration Workflow

The parsing functionality is initiated from the GUI through the MainWindow::runParser() method. This method coordinates the parsing process and updates the user interface accordingly. Key operations include:

- Verifying that the lexer has been properly initialized.
- Instantiating the Parser class using tokens provided by the lexer.
- Running the parse() function to begin syntax analysis.
- Collecting any syntax errors generated during parsing.
- Retrieving the file path to the generated DOT file representing the AST.
- Updating GUI elements such as the status bar and menu actions.
- Displaying a modal error dialog when issues are encountered.

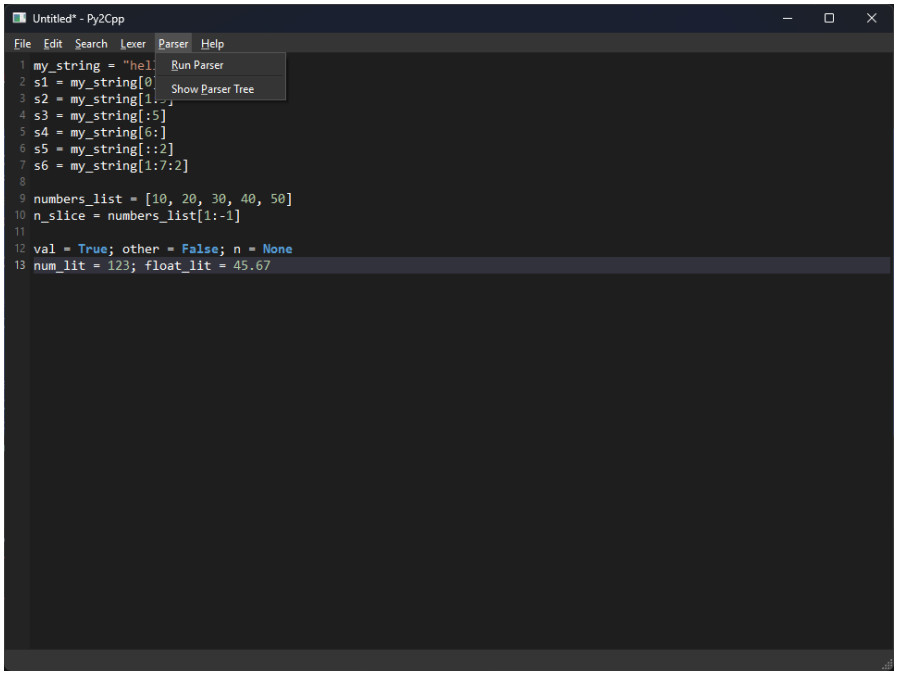


Figure 52

When the parser completes without fatal errors, the user is given the option to view the parse tree in a dedicated dialog. This clean separation of logic ensures maintainability and simplifies debugging.

17.2. Parse Tree Visualization Dialog

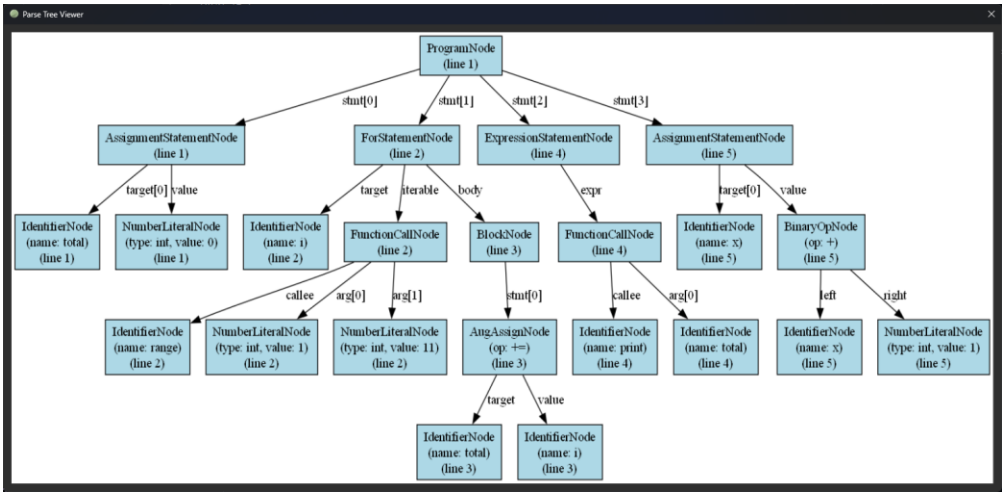


Figure 53

Visualization of the AST is implemented through a new dialog class called ParserTreeDialog. This dialog reads the DOT file output by the parser and displays the corresponding image using Graphviz. Core features include:

- Reading and interpreting the DOT file that encodes the tree structure.
- Converting the DOT file into a PNG image via Graphviz's dot command.
- Displaying the image in a scrollable QLabel embedded within a QScrollArea.
- Automatically adjusting the dialog size to fit the image while maintaining user-friendliness.
- Applying a dark theme to enhance visual clarity and user experience.

17.3. Error Reporting in the GUI

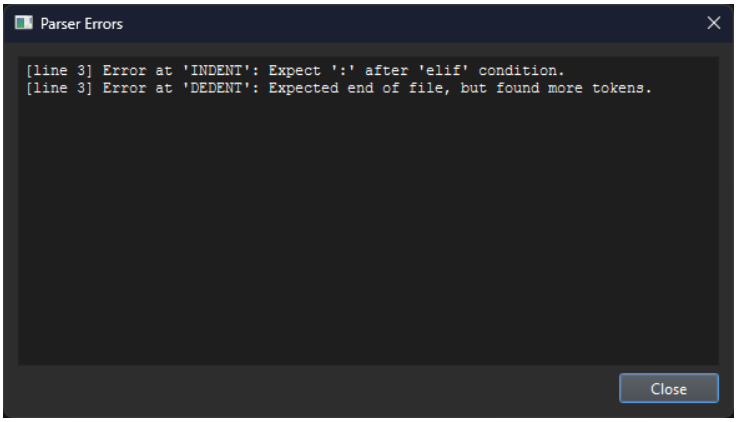


Figure 54

A robust error-handling mechanism was added to ensure that users can easily interpret and respond to parsing issues. ErrorDialog, presents syntax errors in a scrollable, read-only format for clarity and accessibility.

- Errors are displayed in a QPlainTextEdit widget.
- Each error message is listed on its own line.
- The view scrolls to the top automatically so users can focus on the first issue.

This improves the overall workflow by allowing users to stay within the GUI while iteratively addressing syntax problems.

17.4 User Interface Integration Summary

To accommodate the parser features, several updates were made to the main GUI interface:

- A new **"Parser"** menu was added to the menu bar, containing:
  - **Run Parser**: Initiates parsing.
  - **Show Parse Tree**: Displays the parse tree image, if available.
- The status bar provides real-time updates on parsing status.
- Modal dialogs alert users to any errors during execution.
- The ParserTreeDialog applies custom styling and dynamic sizing.

18. Test Cases

18.1 Test Case 1

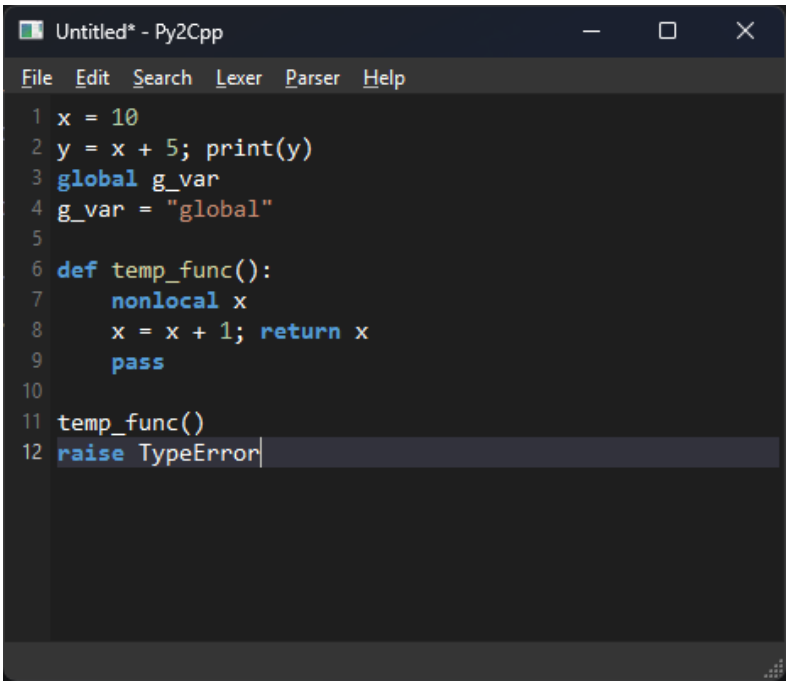


Figure 55

18.2 Test Case 1 Parse Tree

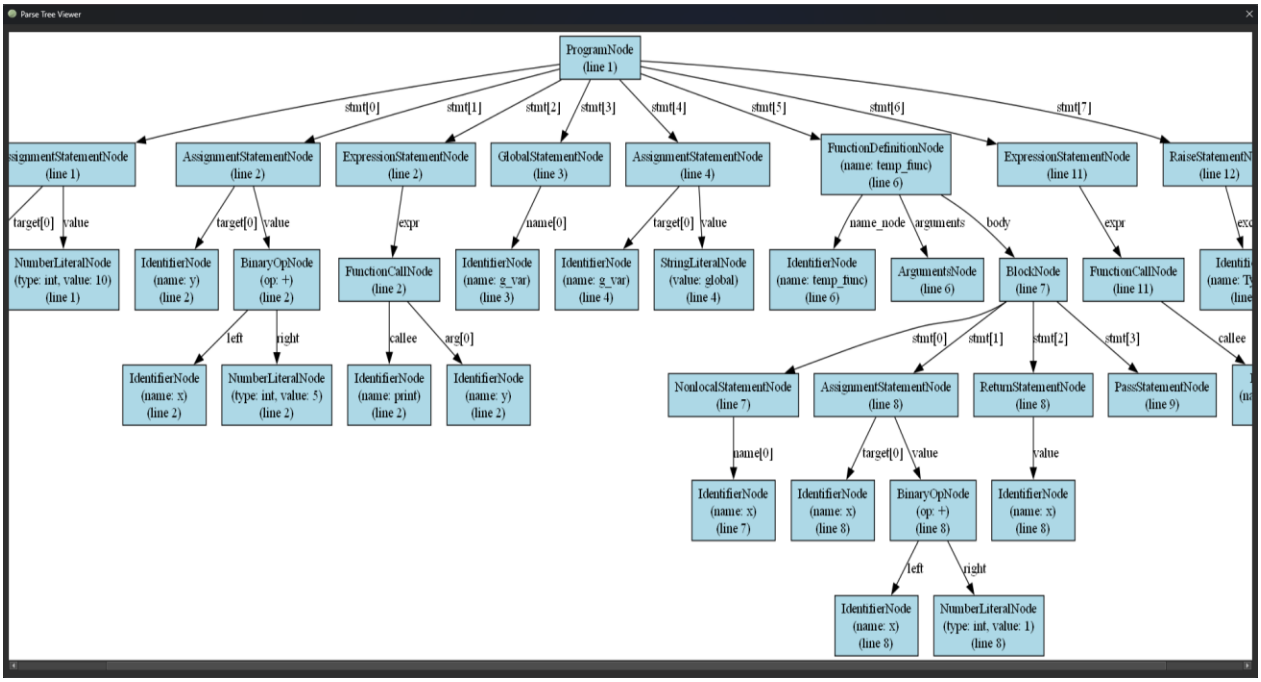


Figure 56

18.3 Test Case 2

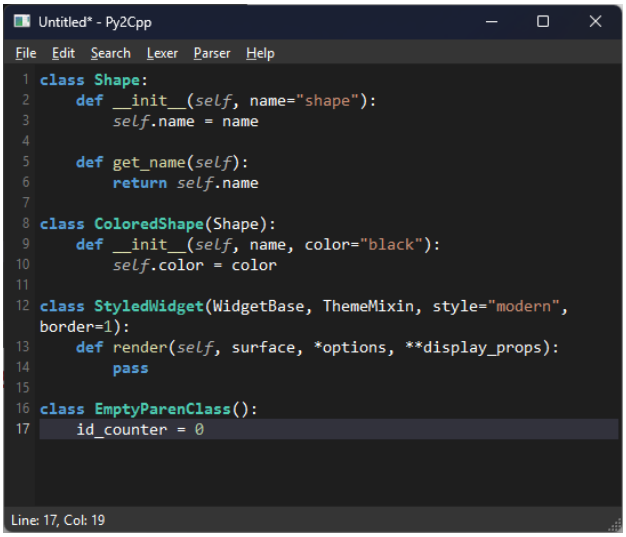


Figure 57

18.4 Test Case 2 Parse Tree

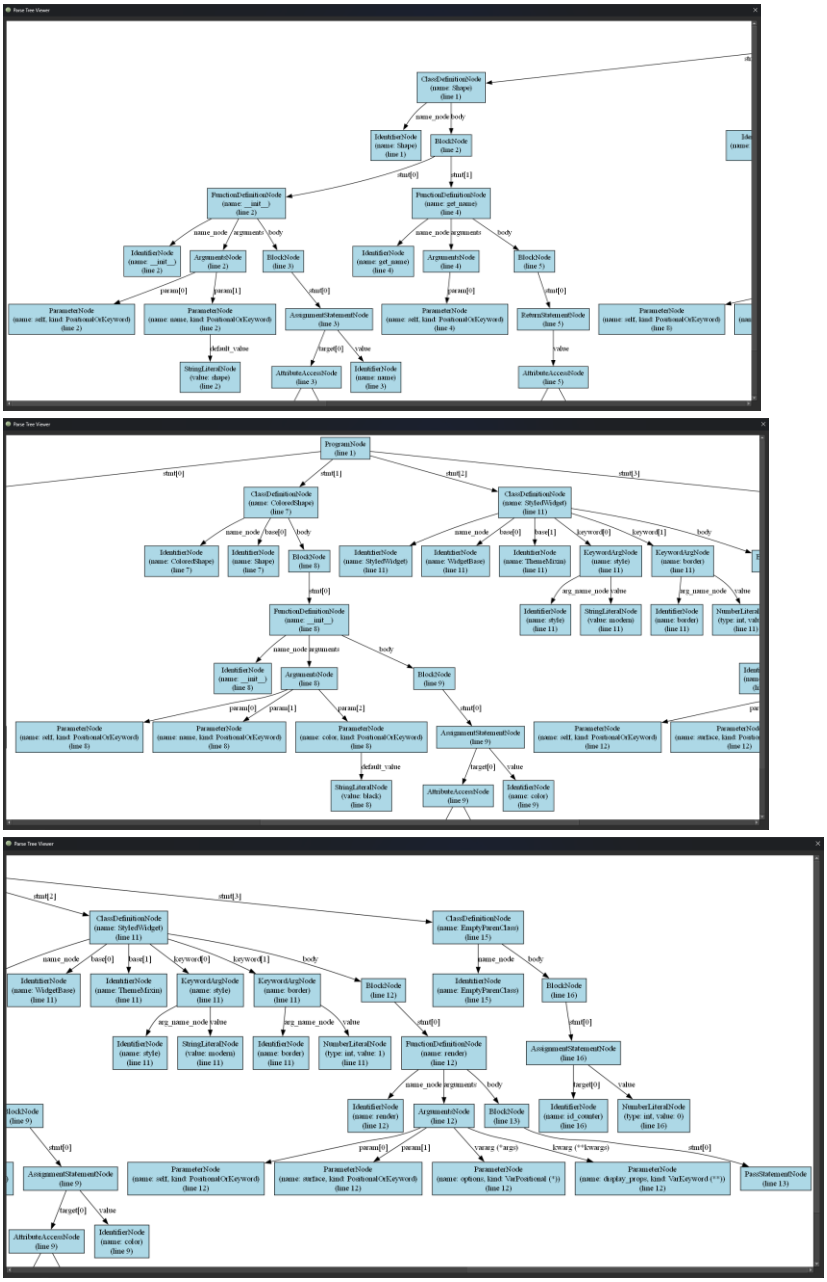


Figure 58

18.5 Test Case 3

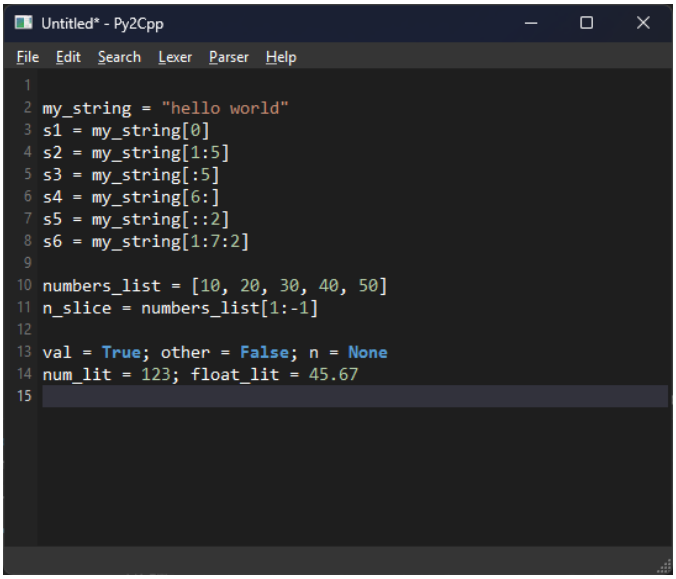
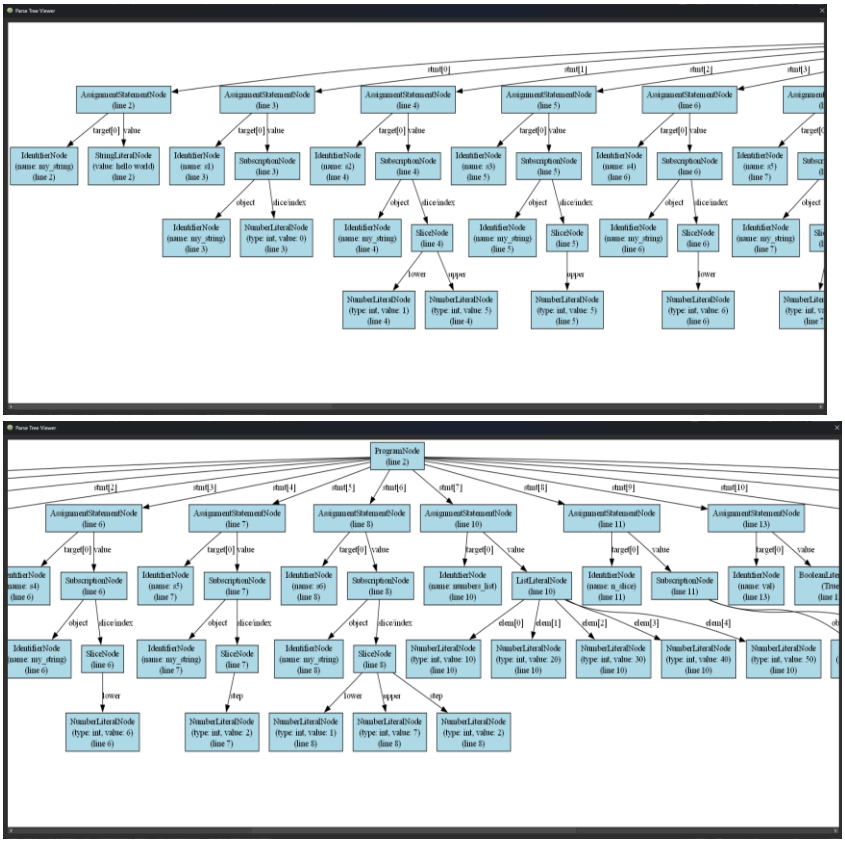


Figure 59

18.6 Test Case 3 Parse Tree



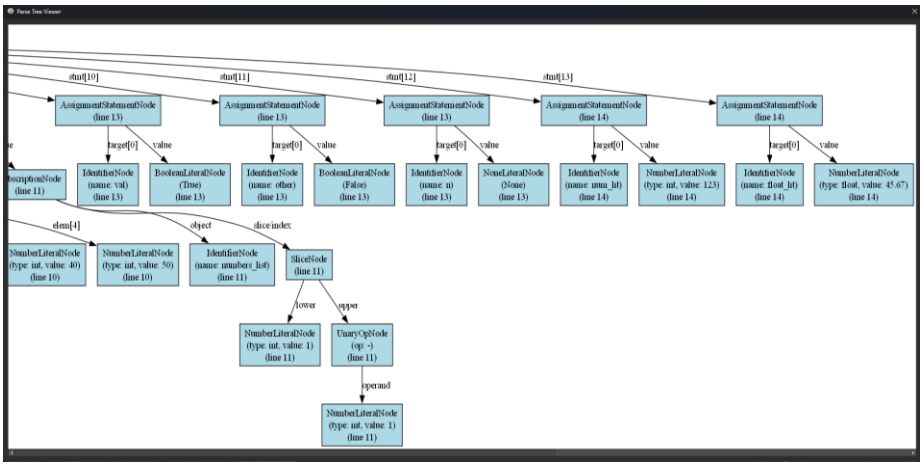


Figure 60

18.7 Test Case 4

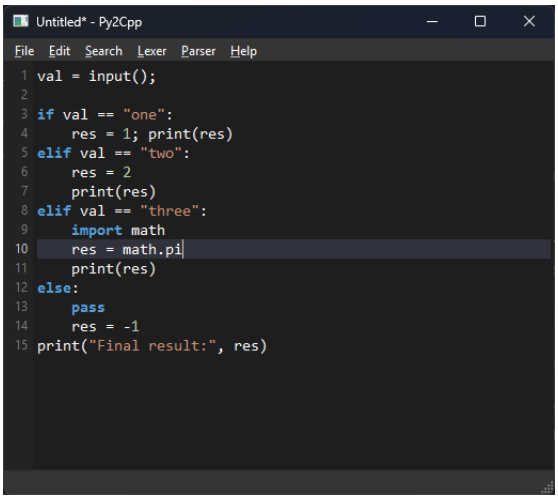


Figure 61

18.8 Test Case 4 Parse Tree

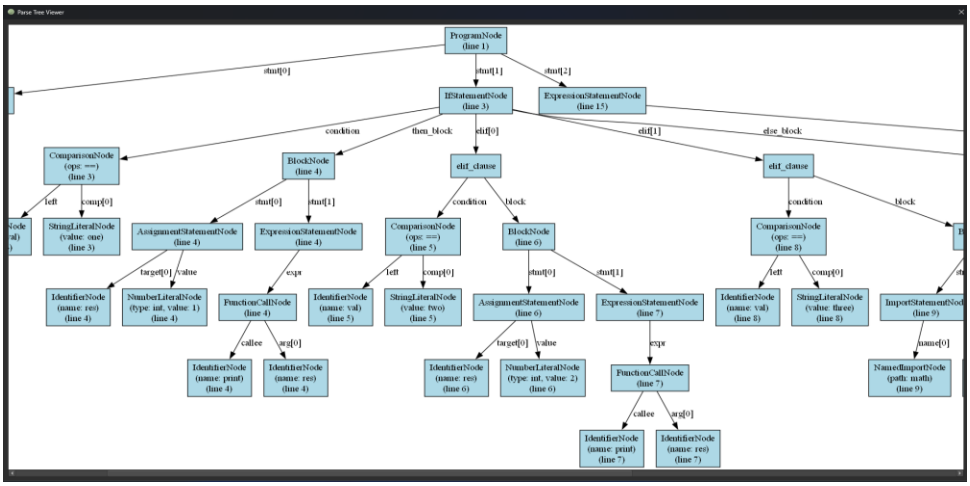


Figure 62

18.9 Errors

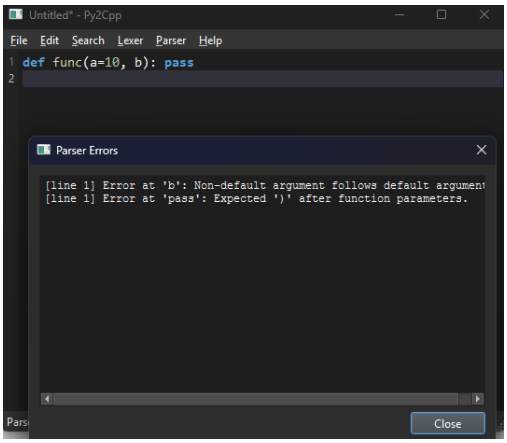


Figure 63

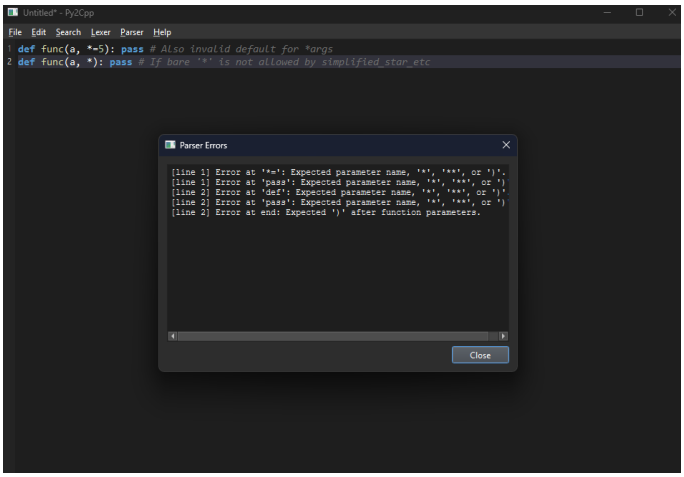


Figure 64

19. Conclusion:

This project has successfully delivered a sophisticated Python lexer and parser, establishing a robust foundation for the initial phases of a compiler targeting a carefully defined subset of the Python language, as outlined in the Python Specifications Document. Developed in C++ and complemented by an intuitive Graphical User Interface (GUI), the system demonstrates a seamless integration of lexical and syntactic analysis. The lexer efficiently tokenizes Python source code, adeptly handles indentation—a hallmark of Python syntax—performs basic type inference to enrich the symbol table and provides detailed lexical error reporting. By breaking down Python source code into a sequence of tokens, the lexer plays a critical role in enabling further syntactic and semantic analysis, successfully identifying and categorizing various elements like keywords, identifiers, literals, and operators despite Python's complex and dynamic syntax.

The parser, employing a Recursive Descent Parsing strategy, meticulously validates token streams against the grammar specified in PYCFG.gram, constructs a well-structured Abstract Syntax Tree (AST), and implements effective error recovery through a "panic mode" mechanism, ensuring resilience against syntactic violations. The project's modular architecture, with its clear directory structure and comprehensive header files, enhances code maintainability and facilitates future extensions. Advanced features, such as the parsing of f-strings, nuanced handling of operator precedence, and resolution of statement ambiguities, highlight the system's capability to manage complex Python constructs within the defined scope. The GUI further elevates the user experience by offering interactive visualization of tokens, symbol tables, parse trees, and error diagnostics, making the compiler's inner workings accessible and educational for users.

Through this project, we demonstrated how core concepts such as tokenization, pattern matching, and symbol table management can be implemented effectively in C++, showcasing the power and flexibility of C++ in handling language analysis tasks. Despite its achievements, the project identifies avenues for further refinement. Expanding support for additional Python features—such as floating-point literals, complex data structures, or advanced control constructs—could broaden its applicability. Enhancements in error recovery, potentially through more sophisticated synchronization techniques, and the addition of real-time AST visualization in the GUI would further improve usability. Looking ahead, integrating semantic analysis and code generation phases could transform this system into a fully operational compiler or interpreter, opening doors to practical applications like code optimization or cross-language translation. In conclusion, this project not only meets its objectives but also exemplifies a deep engagement with compiler design principles, providing a solid platform for future exploration and development in the realm of programming language processing, while highlighting the practical application of compiler theory in real-world scenarios.