

## **Table of Contents: -**

1. Introduction: - .....	5
2. Lexical Analyzer Function: - .....	6
2.1 Indentation & Space Handling: .....	6
2.2 Token Identification: - .....	7
2.2.1 Start State: .....	7
2.2.2 Integer state: .....	9
2.2.3 Float state: .....	10
2.2.4 Identifier state: .....	10
2.2.5 String state: .....	12
2.2.6 Comment state: .....	13
2.2.7 Operators state: .....	13
2.2.8 Error state: .....	14
3. Find symbol table index function: - .....	16
4. Check Error function: - .....	16
5. Get Next Token Function: - .....	17
6. Parser Function: - .....	18
7. Find Parsing Table: - .....	20
8. Evaluating expression: - .....	22
8.1 Precedence function: .....	22
8.2 isRightAssociative .....	22
8.3 Is Number: .....	23
8.4 Apply operation .....	23
8.5 Apply Unary .....	23
8.6 Evaluate Expression: .....	24
9. Updating Symbol table: - .....	26

10. GUI: - .....	27
10.1 Pages:.....	27
10.1.1 Input Page:.....	27
10.1.2 Output Page: .....	28
10.2 GUI Features: .....	29
10.3 GUI error handling.....	33
11. Test Scenario: - .....	34
13. References: -.....	38

# **1. Introduction: -**

This project presents the initial phase in the construction of a compiler for the Python programming language, focusing primarily on lexical analysis. Lexical analysis is a critical component of the compilation process, responsible for converting raw source code into a stream of meaningful tokens that the later stages of the compiler can process. The implementation parses the input character by character to identify syntactic structures such as identifiers, keywords, literals (integers, floats, strings, and characters), operators, delimiters, and indentation levels all of which are essential in Python's syntax. Additionally, the lexical analyzer manages a symbol table to store metadata about identifiers and an error-checking mechanism to detect and log lexical errors. By developing a clear and structured tokenization process, this phase lays a solid foundation for the subsequent stages of syntax and semantic analysis, making it a pivotal step in building a fully functional Python compiler.

## 2. Lexical Analyzer Function: -

The `lexical_analyzer()` function serves as the central component of the lexical analysis stage in a compiler for the Python language. Its primary role is to read the source code character by character, classify sequences of characters into meaningful tokens (like keywords, identifiers, numbers, strings, operators, etc.), and maintain both a token stream and a symbol table. To ensure that the final line of code is processed correctly, the function first appends a newline character to the input text.

```
void MainWindow::lexical_analyzer(vector<char>& text)
{
    text.push_back('\n'); // Ensure final line is processed
}
```

It initializes several key variables, including the following:

- **State:** used to manage scanning logic as part of a finite state machine.
- **startPtr:** which marks the beginning of a potential token.
- **lineCount:** to track line numbers for error and token records.
- **indentStack:** which handles Python's indentation-based block structure.

The function also sets up containers like:

- **Tokens:** to store all identified tokens.
- **Errors:** to keep track of any lexical issues that arise.

### 2.1 Indentation & Space Handling:

The function enters a loop that runs over each character in the input. If a new line is encountered, it calculates the level of indentation by counting spaces and tabs. This indentation is compared against the top of the **indentStack**, and if it's deeper, it adds an **INDENT** token. If it's shallower, it generates one or more

**DEDENT** tokens to signal the end of a block. The core of the function relies on a switch-case structure that uses the current state to determine how the current character should be processed.

```
for (int i = 0; i < text.size(); i++)
{
    char ch = text[i];

    if (newLine)
    {
        spaceCount = 0;
        while (ch == ' ' || ch == '\t')
        {
            spaceCount += (ch == '\t') ? 4 : 1;
            i++;
            ch = text[i];
        }

        if (ch == '\n' || ch == '#')
        {
            newLine = true; // Skip empty or comment-only lines
            if (ch == '#') state = COMMENT;
            else lineCount++;
            continue;
        }

        if (spaceCount > indentStack.back())
        {
            indentStack.push_back(spaceCount);
            Tokens.push_back({ "INDENT", to_string(spaceCount) });
        }
        else
        {
            while (spaceCount < indentStack.back())
            {
                indentStack.pop_back();
                Tokens.push_back({ "DEDENT", to_string(spaceCount) });
            }
        }

        newLine = false;
    }
}
```

## 2.2 Token Identification: -

### 2.2.1 Start State:

The START case is the initial and default state of the lexical analyzer's finite state machine. Each time the analyzer finishes identifying a token or is beginning to process a new line; it enters this state. In **START**, the analyzer examines the current character and decides what kind of token may be

```
if (isdigit(ch))
    state = INTEGER;
```

starting. If the character is a digit, it transitions to the **INTEGER** state.

If it is a letter or an underscore, it switches to **IDENTIFIER**.

```
else if (isalpha(ch) || ch == '_')
    state = IDENTIFIER;
```

Assuming the beginning of a variable or keyword. Quotation marks lead to the **STRINGS** state.

```
else if (ch == '\'' || ch == '\"')
    state = STRINGS;
```

While common operator symbols such as (=, +, -, etc.) cause a transition to the **OPERATORS** state.

```
else if (ch == '=' || ch == '!' || ch == '+' || ch == '-' ||
        ch == '*' || ch == '/' || ch == '%' || ch == '<' || ch == '>' ||
        ch == '&' || ch == '|' || ch == '^' || ch == '~')
    state = OPERATORS;
```

The analyzer also directly identifies brackets and punctuation (like (, ), {, }, :, ,) and adds them as tokens immediately. If there is a digit after the ".", then it is a float, like (.5)

```
else if (ch == ':' || ch == ';' || ch == ',')
    Tokens.push_back({ "punctuation", string(1, ch) });
else if (ch == '.') {
    if (isdigit(text[i + 1]))
        state = FLOAT_STATE;
    else Tokens.push_back({ "punctuation", string(1, ch) });
}
```

The code also handles comments by switching to COMMENT state when finding the "#" symbol.

```
else if (ch == '#')
    state = COMMENT;
```

If a newline is encountered, it increments the line counter and resets the newLine flag. White spaces and tabs are ignored, allowing the analyzer to focus only on meaningful characters. If none of these conditions match, the character is considered invalid, and the analyzer enters the **ERROR** state with a specific error code indicating an unrecognized character.

```

else if (ch == '\n') {
    lineCount++;
    //Tokens.push_back({"newline", "\\n"});
    newLine = true;
}
else if (ch == ' ' || ch == '\t') continue;
else{
    state = ERROR;
    error = InvalidChar;
}
break;

```

## 2.2.2 Integer state:

Entered when the current character (at startPtr) is a digit. As long as you see more digits, you stay in INTEGER.

On “.” :

1.If the very next char (text[i+1]) is not a digit, you treat this as a float with an implicit zero (e.g. scanning 10. → you build "10.0") and immediately emit FLOAT("10.0"), then return to START.

2.Otherwise (next char is a digit), you switch to FLOAT\_STATE to continue parsing a multi digit fractional part (e.g. 10.25).

If you see a letter or underscore (e.g. 123abc), you flag InvalidIdent and go to ERROR.

On any other character (space, operator, bracket, newline, etc.), you slice out text [startPtr...i) as an integer string (e.g. "42"), emit INTEGER ("42"), back up i one position, and return to START.

```

case INTEGER:
{
    if (isdigit(ch)) {///so that when a digit is present it stays in integer state and doesnt go to other states
    }
    else if (ch == '.') {
        if (isdigit(text[i+1])) {
            string tempString(text.begin() + startPtr, text.begin() + i+1);
            tempString += "0";
            Tokens.push_back({ "FLOAT", tempString });
            state = START;
        }
        else state = FLOAT_STATE;
    }
    else if (isalpha(ch) || ch == '_')
    {
        error = InvalidIdent;
        state = ERROR; //Invalid Identifier ex: 123abc
    }
    else
    {
        string tempString(text.begin() + startPtr, text.begin() + i);
        Tokens.push_back({ "INTEGER", tempString });
        state = START;
        i--;
    }
    break;
}

```

### 2.2.3 Float state:

This state is entered when a decimal point follows a sequence of digits, signaling the potential formation of a floating-point number. While in **FLOAT\_STATE**, the analyzer expects to see more digits. If it encounters a second decimal point or a letter/underscore, the number is marked invalid (e.g., 4.3.5 or 2.5a) and transitions to the **ERROR** state with the appropriate error code. If a non-digit character ends the float, the number is extracted, labeled as a **FLOAT** token, added to the token stream, and the state is reset to **START**. This state ensures that floats conform to standard syntax like 3.14 and helps prevent misinterpretation of malformed floats.

```
case FLOAT_STATE:
{
    if (isdigit(ch)) {}
    else if (ch == '.')
    {
        error = InvalidFloat;
        state = ERROR; //Invalid float ex: 5.4.2
    }
    else if (isalpha(ch) || ch == '_')
    {
        error = InvalidIdent;
        state = ERROR; //Invalid Identifier ex: 123abc
    }
    else
    {
        string tempString(text.begin() + startPtr, text.begin() + i);
        Tokens.push_back({lineCount, { "FLOAT", tempString }});
        state = START;
        i--;
    }
    break;
}
```

### 2.2.4 Identifier state:

The **IDENTIFIER** case is responsible for processing names of variables, functions, classes, or any other user-defined identifiers, as



well as recognizing Python keywords. As long as the current character is alphanumeric or an underscore, the analyzer continues in this state. Once a non-valid character for identifiers is found (like a space, operator, or punctuation), the analyzer extracts the full identifier using **startPtr** and current index **i**. It then checks whether the identifier matches any reserved Python keywords (like **if**, **while**, **def**, etc.). If so, it is recorded as a **KEYWORD** token. Otherwise, it's treated as a normal identifier: it is either found or inserted in the symbol table using **findSymbolTableIndex()**, and a reference to it (by index) is added to the token list. Finally, **setValue()** is called to check if the identifier is being assigned a value right after its declaration. Once this process is done, the analyzer resets to the **START** state.

```
case IDENTIFIER:
{
    if (isalnum(ch) || ch == '_') {}
    else
    {
        string tempString(text.begin() + startPtr, text.begin() + i);
        vector<string> keywords = {
            "if", "else", "elif", "def", "return", "for", "while",
            "continue", "break", "and", "or", "not", "print", "in",
            "__init__", "__main__", "None", "range", "pass", "class", "__name__", "self"
        };

        bool isKeyword = false;
        for (const string& k : keywords)
        {
            if (tempString == k)
            {
                Tokens.push_back({lineCount, { "KEYWORD", tempString }});
                isKeyword = true;
                break;
            }
        }

        if (!isKeyword)
        {
            int index = findSymbolTableIndex(symbolTable, lineCount, tempString);
            Tokens.push_back({lineCount, { "ID", to_string(index) }});
            setValue(text, i, symbolTable, tempString);
        }

        state = START;
        i--;
    }
    break;
}
```

## 2.2.5 String state:

Once we see a quote (single or double) in the START state, we enter STRINGS. We record the opening quote in quoteType and set iPtr to the first character of the literal. We then loop until we hit either:

- An escape sequence (\ + next char): we append the backslash (only if the following char is alphabetic) and the escaped character itself to content, then advance iPtr by 2.
- A matching closing quote (text[iPtr] == quoteType): we break out of the loop and mark the string terminated.
- A newline before the closing quote: we abort the scan (unterminated string).

If terminated successfully, we emit three tokens—Quotation for the opening quote, STRING with the full content, and Quotation for the closing quote—then set i = iPtr and return to START.

Otherwise, we set error = UnterminatedString, switch to the ERROR state (which will report the line and message), back up i one position, and let the error-handler kick in.

This ensures that any backslash escape (like "\n", "\\\"", or "\\t") is preserved in the string's content, and that unterminated literals are flagged cleanly.

```
case STRINGS:
{
    char quoteType = text[startPtr];
    int iPtr = startPtr + 1;
    string content;
    bool terminated = false;
    while (iPtr < text.size()) {
        if (text[iPtr] == '\\' && iPtr + 1 < text.size()) {
            // include escape sequence
            if (isalpha(text[iPtr+1]))
                content += text[iPtr];
            content += text[iPtr + 1];
            iPtr += 2;
        }
        else if (text[iPtr] == quoteType) {
            terminated = true;
            break;
        }
        else if (text[iPtr] == '\n') {
            // unterminated string
            break;
        }
        else {
            content += text[iPtr];
            iPtr++;
        }
    }

    if (terminated)
    {
        string quote(1, quoteType);
        Tokens.push_back({ "Quotation", quote });
        Tokens.push_back({ "STRING", content });
        Tokens.push_back({ "Quotation", quote });
        i = iPtr; // advance main index to closing quote
        state = START;
    }

    else
    {
        error = UnterminatedString;
        state = ERROR;
        i--;
    }
    break;
}
```

## 2.2.6 Comment state:

The **COMMENT** state deals with Python comments, which begin with a character which begins with a "#". Once this character is encountered, the analyzer skips all characters until it reaches a new line. Comments are not tokenized since they are ignored during compilation, but they are properly skipped to avoid interfering with the rest of the code. When a newline is found, the analyzer increments the line count and resets the **newLine** flag, ensuring the next line is analyzed correctly. After this, the state is reset to **START**.

```
case COMMENT:
{
    while (i < text.size() && text[i] != '\n') {
        i++;
    }
    state = START;
    newLine = true;
    lineCount++;
    break;
}
```

## 2.2.7 Operators state:

This case is responsible for identifying both single-character and compound operators. The analyzer starts with a character such as (=, +, -, etc.) and looks ahead to the next character to see if a valid two-character operator can be formed. Examples include (==, !=, >=, <=, +=, \*\*, and others). If a valid two-character operator is found, it is added to the Tokens list as a single token. If not, only the first character is considered the operator, and it's added as a single-character token. In both cases, the analyzer then returns to the **START** state. If a second character was not part of the operator, it is reprocessed by decrementing **i**.

```

case OPERATORS:
{
    string op(1, text[startPtr]); // Use the original character that caused transition to OPERATORS
    if (i < text.size())
    {
        string twoChar = op + text[i];
        if (twoChar == "==" || twoChar == "!=" || twoChar == "<=" ||
            twoChar == ">=" || twoChar == "<<" || twoChar == ">>" ||
            twoChar == "***" || twoChar == "*=" || twoChar == "+=" ||
            twoChar == "-=" || twoChar == "/=" || twoChar == "%=" ||
            twoChar == "&=" || twoChar == "|=" || twoChar == "^=")
        {
            Tokens.push_back({lineCount, { "OPERATOR", twoChar }});
            state = START;
            break;
        }
    }

    Tokens.push_back({lineCount, { "OPERATOR", op }});
    state = START;
    i--; // since ch wasn't part of this operator, reprocess it
    break;
}

```

## 2.2.8 Error state:

The **ERROR** state manages all types of lexical errors. Based on the value of the error variable, it identifies what kind of issue occurred such as an invalid character, an invalid identifier (e.g., starting with a digit), an invalid float (with multiple dots), or an unterminated string. It uses the helper function **checkErrors()** to log the error while avoiding duplicate entries for the same line and error type. After recording the error, the analyzer attempts to recover by transitioning back to the **START** state to continue processing the remaining input, rather than halting execution entirely.

```

case ERROR:
{
    switch (error) {
    case InvalidChar:
        checkErrors(errors, lineCount, string("invalid char: ") + text[i - 1]);
        i--;
        error = 0;
        state = START;
        errorFound = true;
        break;

    case InvalidIdent:
        if (!(isalnum(ch) || ch == '_')) {
            string errorString(text.begin() + startPtr, text.begin() + i);
            checkErrors(errors, lineCount, string("invalid identifier: ") + errorString);
            i--;
            state = START;
            error = 0;
        }
        errorFound = true;
        break;

    case InvalidFloat:
        if (!(isdigit(ch) || ch == '.'))
        {
            string errorString(text.begin() + startPtr, text.begin() + i);
            checkErrors(errors, lineCount, string("invalid Float: ") + errorString);
            state = START;
            error = 0;
            i--;
        }
        errorFound = true;
        break;

    case UnterminatedString:
        checkErrors(errors, lineCount, "Unterminated String");
        error = 0;
        i--;
        state = START;
        errorFound = true;
        break;
    }
}
}

```

### 3. Find symbol table index function: -

The **findSymbolTableIndex()** function manages the symbol table, ensuring that identifiers are uniquely tracked. When given an identifier name (**tempString**), it searches through the existing **symbolTable**, which is a vector of pairs. Each entry contains a line number and a vector<string> representing identifier details: name, value, and type. The function checks if the identifier already exists in the table by comparing the name (first string in the vector). If it finds a match, it returns the index of that entry, allowing other parts of the analyzer to reference or update it efficiently. If the identifier doesn't exist yet, the function adds a new entry with the current line number and the identifier's name, leaving the value and type fields initially empty. It then returns the index of this new entry. This function is vital for maintaining a consistent and accurate symbol table and helps facilitate semantic analysis and type checking down the line.

```
int MainWindow::findSymbolTableIndex(vector<pair<int, vector<string>>>& symbolTable, int lineCount, string tempString)
{
    for (int i = 0; i < symbolTable.size(); i++) {
        // Check if the symbol matches (stored in the first element of the vector<string>)
        if (symbolTable[i].second[0] == tempString) {
            return i; // Return the index if the symbol already exists
        }
    }

    // Add a new entry to the symbolTable with the line count and symbol details
    symbolTable.push_back({lineCount, {tempString, "", ""}});
    return symbolTable.size() - 1; // Return the index of the newly added symbol
}
```

### 4. Check Error function: -

The **checkErrors()** function serves as a lightweight mechanism for tracking and storing unique lexical errors. It accepts a list of existing errors (**errors**), the current line number (**lineCount**), and the specific error message (**error**). The function first checks whether an identical error (same line number and message) already exists in the list to prevent duplicates. If it doesn't find a match, it appends the error as a pair consisting of the line number and error message. This helps in

generating clear, readable feedback to the user or developer about issues in their code, without overwhelming them with repetitive messages. By centralizing error tracking here, the main analyzer function can simply call this utility when an error occurs, keeping the overall design modular and clean.

```
void MainWindow::checkErrors(vector<pair<int, string>>& errors, int lineCount, string error) {  
    for (auto i:errors) {  
        if (i.first==lineCount) {  
            if (i.second==error) {  
                return;  
            }  
        }  
    }  
    errors.push_back({ lineCount,error});  
}
```

## **5. Get Next Token Function: -**

The getNextToken function is designed to sequentially return tokens from a vector of token pairs. It uses a static int variable named currentTokenIndex to keep track of the position in the token stream across multiple calls to the function. This allows the function to "remember" where it left off without needing to reset or pass additional state externally. Each time the function is called, it checks if the current index has reached or exceeded the size of the token stream. If so, it returns a special end-of-stream token, represented by the pair {"\$", "\$"}, which commonly signifies the end of input in parsers. Otherwise, it returns the token at the current index and then increments the index for the next call.

```
pair<string, string> getNextToken(vector<pair<string, string>> tokenStream) {  
    static int currentTokenIndex = 0;  
    if (currentTokenIndex >= tokenStream.size()) {  
        return { "$", "$" };  
    }  
    return tokenStream[currentTokenIndex++];  
}
```

## 6. Parser Function: -

This part sets up the initial state of the parser. The token is initialized by fetching the first token using getNextToken(). The Seifos stack is used as

```
pair<string, string> token = getNextToken(tokens);
stack<string> Seifos;
Seifos.push("$");
Seifos.push("program");
stack<string> INDENT;
INDENT.push("0");
int lineCount = 1;
string ID;
bool assignment = false;
vector<string> assignment_expr;
```

the parser's main stack, where \$ is the bottom-of-stack marker, and "program" is the start symbol of the grammar. The INDENT stack is used to handle Python-like indentation tracking, starting with a base indent level of 0. The variables lineCount, ID, assignment, and assignment\_expr are used to track line numbers, current identifier for assignment, whether an assignment is in progress, and the list of tokens making up the right-hand side expression of an assignment, respectively.

```
while (Seifos.top() != "$")
{
```

This is the core parsing loop. It continues running until the top of the parsing stack (Seifos) is \$, which marks the successful end of parsing. Inside the loop, the parser compares tokens against grammar rules or terminal symbols, consumes matched tokens, and expands non-terminals using a parsing table.

```
if (token.first == "ID")
{
    token.first = "identifier";
    if (assignment == false)
        ID = token.second; //index in symbol table
}
else if (token.first == "INTEGER" || token.first == "FLOAT")
{
    token.first = "numeric_literal";
}
```



This block normalizes token types. ID tokens are renamed as "identifier", and either "INTEGER" or "FLOAT" tokens are grouped as "numeric\_literal". If the current token is an identifier and an assignment is not active, the parser saves the token's value into ID, assuming it's the variable being assigned to.

```
if (token.first == "identifier" || token.first == "numeric_literal" || token.first == "NEWLINE" ||
    token.first == "INDENT" || token.first == "DEDENT" || token.first == "string_literal")
{
```

These exact words were written in this condition because they match the first part of the token provided. Here, the parser checks if the token is one of the expected high-level grammar tokens. If the top of the stack matches the token, it consumes it (by popping the stack and getting the next token). If in assignment mode, it also collects the expression parts for evaluation later.

```
if (Seifos.top() == token.first) //MATCH
{
    if (assignment == true) {
        if (token.first == "numeric_literal") {
            assignment_expr.push_back(token.second);
        }
        else if (token.first == "identifier") {
            assignment_expr.push_back(symbolTable[stoi(token.second)][1]);
        }
    }

    if (token.first == "NEWLINE") {
        if (assignment == true) {
            updateSymbolTable(ID, assignment_expr, symbolTable);
            assignment = false;
        }
        lineCount++;
    }
}

if (token.first == "INDENT") {
    INDENT.push(token.second);
}
else if (token.first == "DEDENT") {
    while (stoi(token.second) < stoi(INDENT.top())) {
        INDENT.pop();
    }
    if (INDENT.top() != token.second) {
        cout << "Unindent amount does not match previous indent" << endl;
        cout << "top of indent stack: " << INDENT.top() << endl;
        break;
    }
}
```

```

else
{
    if (Seifos.top() == token.second) //MATCH
    {
        if (assignment == true) {
            assignment_expr.push_back(token.second);
        }

        if (token.second == "=") {
            assignment = true;
        }

        Seifos.pop();
        token = getNextToken(tokens);
        continue;
    }

    if (!findInParsingTable(Seifos, token.second, parsingTable))
    {
        break;
    }
}

```

If the top of the stack is a **literal symbol** (like "=", "+", or "("), the parser checks for a match with token.second (the actual character from the token). If matched, the token is consumed. If the token is "=", it activates assignment mode. If it's part of an expression and assignment is active, the symbol

is added to the expression vector.

```

if (token.first == "$" && Seifos.top() == "$")
{
    cout << "ACCEPTED!!!";
}
else
{
    cout << "top of stack: " << Seifos.top() << endl;
    cout << "Error in line: " << lineCount << ", unexpected token \"" << token.second << "\"";
}

```

After exiting the loop, the parser checks whether parsing ended cleanly: both the token stream and parsing stack must reach the end symbol (\$). If so, it prints "ACCEPTED!" Otherwise, it prints a descriptive error message with the current stack top and token, along with the line number where the issue occurred.

## 7.Find Parsing Table: -

```

bool findInParsingTable(stack<string>& parseStack, const string& token, map<string, map<string, string>>& parsingTable) {
    string top = parseStack.top();
    string production = parsingTable[top][token];
}

```

This function is a core part of an LL (1) predictive parser. It's used when the current token does not match a terminal on top of the stack. It looks up the non-terminal at the top of the stack and the current input token in the parsing table and attempts to expand the non-terminal based on the appropriate production rule. Here, the function looks at

the top of the parser stack (assumed to be a non-terminal) and uses it together with the current input token into the parsing table. The value retrieved is the production rule (as a string) that should be applied for this combination.

If no production rule exists in the table for this combination, it means the parser doesn't know how to proceed. This is a syntax error, and the function returns

```
if (production == "") {  
    return false;  
}  
else if (production == "epsilon") {  
    parseStack.pop();  
    return true;  
}
```

false, signaling failure. Also, If the production rule is "epsilon", it means the non-terminal can vanish. So, we simply pop it off the stack and return true, signaling success.

```
else {  
    // Split the production string into symbols  
    vector<string> symbols;  
    istringstream iss(production);  
    string sym;  
    while (iss >> sym) {  
        symbols.push_back(sym);  
    }  
  
    parseStack.pop();  
  
    // Push symbols in reverse order  
    for (auto it = symbols.rbegin(); it != symbols.rend(); ++it) {  
        parseStack.push(*it);  
    }  
  
    return true;  
}
```

If there is a valid production rule found in the parsing table, the function processes it by first splitting the production string into individual grammar symbols. This is done using an istringstream, under the assumption that the symbols in the production are separated by spaces. After splitting the production, the function pops the current non-terminal from the top of the parser stack, since it is about to be expanded. Then, it pushes the right-hand side symbols of the production onto the stack in reverse order. This reverse pushing is crucial because the parser stack operates in a last-in, first-out manner.

By reversing the order, the leftmost symbol in the production ends up on top of the stack, ensuring that parsing proceeds in the correct left-to-right order as required by the grammar.

## 8. Evaluating expression: -

### 8.1 Precedence function:

```
int precedence(const string& op) {  
    if (op == "or") return 1;  
    if (op == "and") return 2;  
    if (op == "not") return 3;  
    if (op == "<" || op == ">" || op == "==" || op == "!=" ||  
        op == "<=" || op == ">=") return 4;  
    if (op == "|") return 5;  
    if (op == "^") return 6;  
    if (op == "&") return 7;  
    if (op == "<<" || op == ">>") return 8;  
    if (op == "+" || op == "-") return 9;  
    if (op == "*" || op == "/" || op == "%") return 10;  
    if (op == "**") return 11;  
    return -1;  
}
```

This function determines the **precedence** level of a given operator. It returns an integer where a higher number represents higher precedence. Operators like or, and, not, comparison operators (e.g., <, >, ==, !=), bitwise operators (e.g., &, |), arithmetic operators (e.g., +, \*), and exponentiation (\*\*) are ranked according to standard operator precedence in most programming languages. If the operator is not recognized, the function returns -1.

### 8.2 isRightAssociative:

```
bool isRightAssociative(const string& op) {  
    return op == "**" || op == "not";  
}
```

This function checks if an operator is right-associative, meaning it groups from right to left during evaluation. Operators such as exponentiation (\*\*) and logical negation (not) are considered the right associative. It returns true for these and false otherwise.

## 8.3 Is Number:

This function determines whether a given token is a valid number or boolean literal. It checks

```
bool isNumber(const string& token) {  
    if (token == "true" || token == "false") return true;  
    char* end = nullptr;  
    strtod(token.c_str(), &end);  
    return *end == '\\0';  
}
```

for "true" and "false" directly. Otherwise, it uses strtod() to attempt to convert the string to a floating-point number, and checks if the entire string was successfully converted. If so, it's considered a number.

## 8.4 Apply operation:

This function takes two numerical operands and an operator and applies the operator to the operands. It supports arithmetic operations (+, -, \*, /, %, \*\*), comparison operations (==, <, >=, etc.), bitwise operations (&, |, ^, <<, >>), and logical operations (and, or). If the operator is not recognized, it throws a runtime error.

```
// Apply binary or unary operator  
double applyOp(double a, double b, const string& op) {  
    if (op == "+") return a + b;  
    if (op == "-") return a - b;  
    if (op == "*") return a * b;  
    if (op == "/") return a / b;  
    if (op == "%") return fmod(a, b);  
    if (op == "**") return pow(a, b);  
  
    if (op == "==") return a == b;  
    if (op == "!=") return a != b;  
    if (op == "<") return a < b;  
    if (op == "<=") return a <= b;  
    if (op == ">") return a > b;  
    if (op == ">=") return a >= b;  
  
    if (op == "&") return (int)a & (int)b;  
    if (op == "|") return (int)a | (int)b;  
    if (op == "^") return (int)a ^ (int)b;  
    if (op == "<<") return (int)a << (int)b;  
    if (op == ">>") return (int)a >> (int)b;  
  
    if (op == "and") return a && b;  
    if (op == "or") return a || b;  
}
```

## 8.5 Apply Unary:

This function applies a unary operator to a single operand. Currently, it supports the logical negation operator (not). If the operator is unrecognized, it throws a runtime error.

## 8.6 Evaluate Expression:

The `evaluateExpression` function is designed to evaluate a mathematical or logical expression given as a sequence of tokens in infix notation (e.g., {"3", "+", "4", "\*", "2"}). To perform this evaluation correctly while honoring the precedence and associativity of operators, the function first converts the infix expression into **Reverse Polish Notation (RPN)** using the **Shunting Yard algorithm**. This two-phase process—conversion followed by evaluation—allows for reliable and structured computation of even complex expressions involving parentheses, multiple operators, and unary operations like negation or logical NOT.

```
double evaluateExpression(const vector<string>& tokens) {
    vector<string> output;
    stack<string> ops;

    unordered_set<string> unaryOps = { "not", "-" };

    for (size_t i = 0; i < tokens.size(); ++i) {
        const string& token = tokens[i];

        if (isNumber(token)) {
            output.push_back(token);
        }
        else if (token == "(") {
            ops.push(token);
        }
        else if (token == ")") {
            while (!ops.empty() && ops.top() != "(") {
                output.push_back(ops.top());
                ops.pop();
            }
            if (!ops.empty()) ops.pop(); // remove "("
        }
        else {
            // Check for unary operators
            bool isUnary = false;
            if (token == "not") {
                isUnary = true;
            }
            else if (token == "-") {
                if (i == 0 || tokens[i - 1] == "(" || isOperator(tokens[i - 1])) {
                    isUnary = true;
                }
            }

            if (isUnary) {
                ops.push("u" + token); // Mark as unary
            }
            else {
                // Binary operator logic
                while (!ops.empty() && ops.top() != "(" &&
                    (precedence(ops.top()) > precedence(token) ||
                     (precedence(ops.top()) == precedence(token) && !isRightAssociative(token)))) {
                    output.push_back(ops.top());
                    ops.pop();
                }
                ops.push(token);
            }
        }
    }
}
```

In the first phase, the function uses two data structures: an output list to hold the RPN expression and a stack to temporarily store operators and parentheses. It iterates over each token in the input vector and classifies it as a number, parenthesis, or operator. Numbers and boolean values like "true" or "false" are added directly to the output list. Left parentheses are pushed onto the operator stack, while right parentheses trigger the popping of operators from the stack into the output list until a matching left parenthesis is found. For operators, the function carefully checks for precedence and associativity. Unary operators such as "not" and "-" (used for negation) are detected based on their context and marked uniquely (e.g., "u-" for unary minus). Binary operators are pushed onto the stack only after popping other operators of higher or equal precedence, ensuring the correct evaluation order.

Once all input tokens have been processed, any remaining operators on the stack are moved to the output list, completing the RPN conversion. In the second phase, the function evaluates the resulting RPN expression. It does so using another stack to hold intermediate numeric results. As it scans the RPN tokens, numbers are pushed onto this stack, while operators cause one or two values to be popped, an operation performed (like addition, multiplication, or logical comparison), and the result pushed back. Unary operations (like "u-" or "unot") only use one operand, while binary operations like "+", "\*", or ">" use two. The correct evaluation is ensured by helper functions `applyUnary` and `applyOp`, which handle both arithmetic and logical behavior.

```
while (!ops.empty()) {
    output.push_back(ops.top());
    ops.pop();
}

// Evaluate RPN
stack<double> values;
for (const string& token : output) {
    if (isNumber(token)) {
        if (token == "true") values.push(1.0);
        else if (token == "false") values.push(0.0);
        else values.push(stod(token));
    }
    else if (token.substr(0, 1) == "u") { // Unary operator
        string op = token.substr(1);
        double a = values.top(); values.pop();
        values.push(applyUnary(a, op));
    }
    else {
        double b = values.top(); values.pop();
        double a = values.top(); values.pop();
        values.push(applyOp(a, b, token));
    }
}

return values.top();
```



The final value left on the stack after processing the entire RPN expression is the computed result of the original infix expression. This value is returned by the function as a double, supporting both integer and floating-point arithmetic. Overall, this function robustly supports a wide variety of Python-like expressions, including logical operators ("and", "or", "not"), bitwise operations, comparison operators, and nested expressions using parentheses.

## 9. Updating Symbol table: -

```
void updateSymbolTable(string ID, vector<string> assignment_expr, vector<vector<string>> symbolTable) {
    for (int i = 0; i < assignment_expr.size(); i++) {
        if (assignment_expr[i] == "") {
            return;
        }
    }
    double value = evaluateExpression(assignment_expr);
    symbolTable[stoi(ID)][1] = value;
}
```

This function is used to **update a symbol table** entry with the result of evaluating an expression. It takes an identifier (as a string index), a list of tokens representing the expression, and the symbol table itself. After evaluating the expression using `evaluateExpression`, it updates the corresponding entry in the symbol table with the computed value. It also logs the current token and the updated value for debugging.

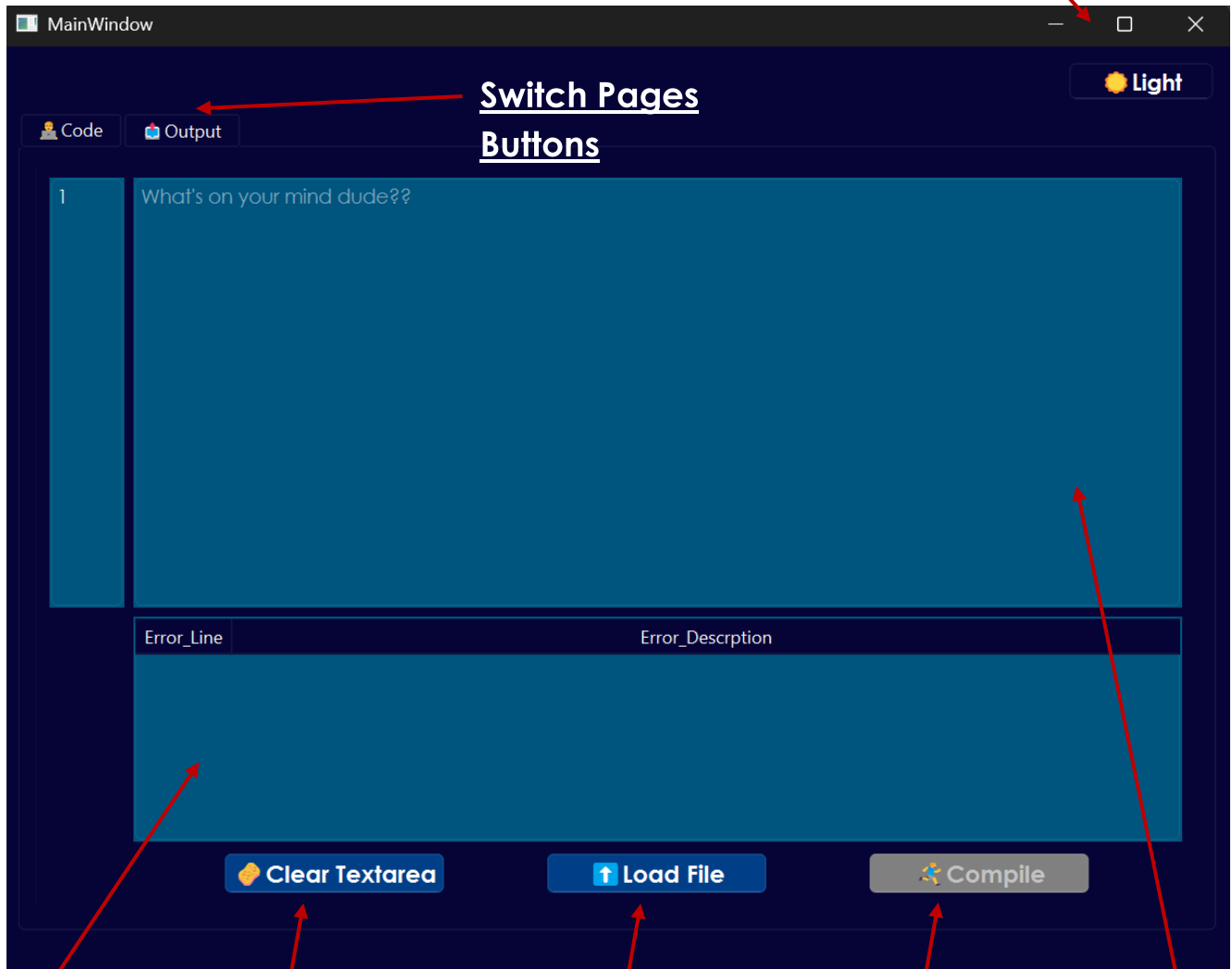


## 10. GUI: -

### 10.1 Pages:

#### 10.1.1 Input Page:

Changing theme



Error Section

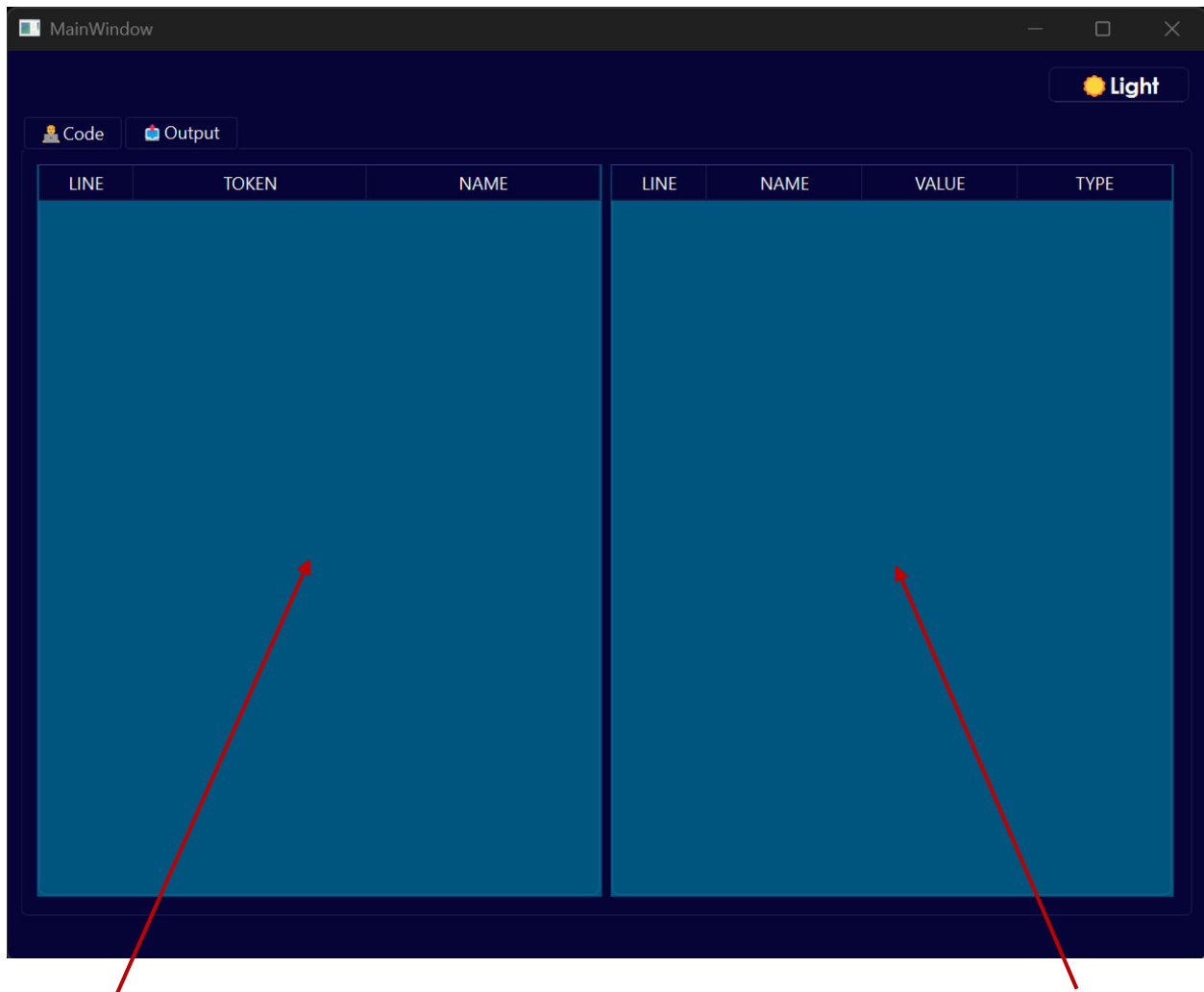
Clear Button

Load file Button

Compile  
Button

Code  
Textarea

## 10.1.2 Output Page:



Token Table

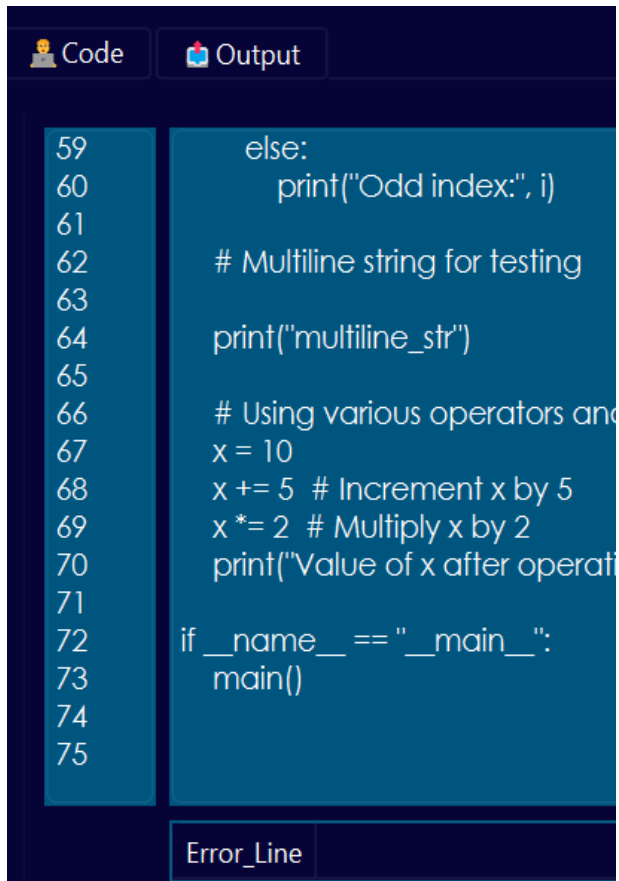
Symbol Table

## 10.2 GUI Features:

The GUI includes several intuitive features to enhance user interaction and functionality.

**-Numbered lines:** The **Numbered Lines** feature ensures that each line of text is clearly indexed, making it easier to reference and navigate large files, code snippets or tracing errors.

By using a read-only textarea which is connected to the code textarea, it changes the numbers of the lines according to how many '\n' is found in the code textarea which of course indicates a new line.



The screenshot shows a code editor interface with two tabs: 'Code' and 'Output'. The 'Code' tab is active, displaying Python code. On the left side of the code area, line numbers are listed from 59 to 75. The code includes an 'else' block, a multiline string, arithmetic operations on 'x', and a main function guard.

```
59     else:
60         print("Odd index:", i)
61
62     # Multiline string for testing
63
64     print("multiline_str")
65
66     # Using various operators and
67     x = 10
68     x += 5 # Increment x by 5
69     x *= 2 # Multiply x by 2
70     print("Value of x after operat
71
72 if __name__ == "__main__":
73     main()
74
75
```

At the bottom of the editor, there is an 'Error\_Line' label followed by an empty input field.

```
void MainWindow::updateLineNumbers() {
    // Get the text from the main text area
    QString text = ui->textarea->toPlainText();

    // Split the text by '\n' to count lines
    QStringList lines = text.split('\n');
    int lineCount = lines.size();

    // Generate line numbers
    QString lineNumbers;
    for (int i = 1; i <= lineCount; ++i) {
        lineNumbers += QString::number(i) + "\n";
    }

    // Update the line number area
    lineNumberArea->setPlainText(lineNumbers);
}
```

**-Theme change:** The **Theme Change** feature allows users to switch between light and dark themes with a single click, providing a comfortable viewing experience tailored to their preferences.

```

1 # This is an example Python program to test the lexical analyzer.
2
3 def factorial(n):
4     if n < 0:
5         return None # Negative number, no factorial exists.
6     elif n == 0:
7         return 1
8     else:
9         result = 1
10        for i in range(1, n + 1):
11            result *= i
12        return result
13
14 def fibonacci(n):
15     a = 0
16     b = 1
17     if n < 0:
18         return None # Negative index not allowed.
  
```

```

1 # This is an example Python program to test the lexical analyzer.
2
3 def factorial(n):
4     if n < 0:
5         return None # Negative number, no factorial exists.
6     elif n == 0:
7         return 1
8     else:
9         result = 1
10        for i in range(1, n + 1):
11            result *= i
12        return result
13
14 def fibonacci(n):
15     a = 0
16     b = 1
17     if n < 0:
18         return None # Negative index not allowed.
  
```

LINE	TOKEN	NAME	VALUE	TYPE
1	INDENT	1		
2	DEDENT	0		
3	KEYWORD	def		
4	ID	n		
5	BRACKET	(		
6	ID	i		
7	BRACKET	)		
8	punctuation	:		
9	INDENT	4		
10	KEYWORD	if		
11	ID	total		
12	OPERATOR	<		
13	INTEGER	0		
14	punctuation	:		
15	INDENT	8		
16	KEYWORD	return		
17	KEYWORD	None		
18	DEDENT	4		
19	KEYWORD	elif		
20	ID	x		

LINE	TOKEN	NAME	VALUE	TYPE
1	INDENT	1		
2	DEDENT	0		
3	KEYWORD	def		
4	ID	n		
5	BRACKET	(		
6	ID	i		
7	BRACKET	)		
8	punctuation	:		
9	INDENT	4		
10	KEYWORD	if		
11	ID	total		
12	OPERATOR	<		
13	INTEGER	0		
14	punctuation	:		
15	INDENT	8		
16	KEYWORD	return		
17	KEYWORD	None		
18	DEDENT	4		
19	KEYWORD	elif		
20	ID	x		

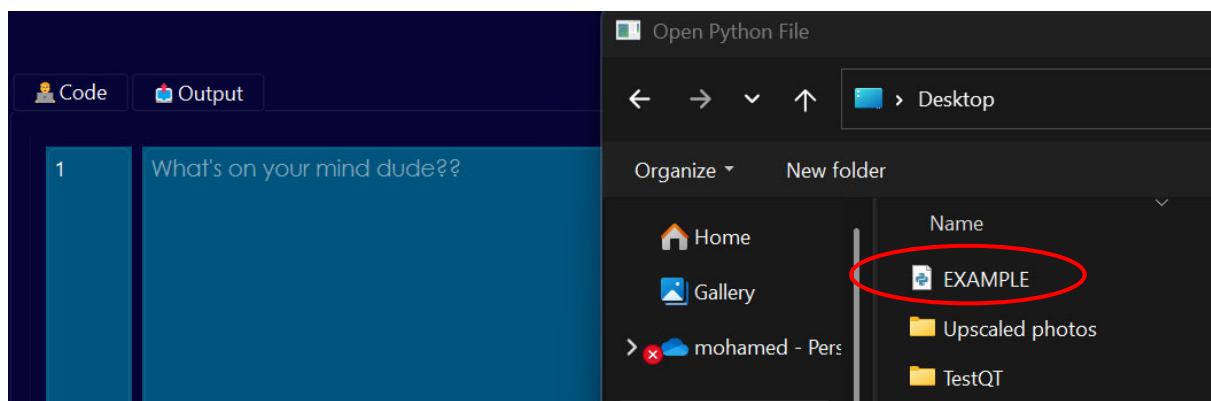
```

1 # This is an example Python program to test the lexical analyzer.
2
3 def factorial(n):
4     if n < 0:
5         return None # Negative number, no factorial exists.
6     elif n == 0:
7         return 1
8     else:
9         result = 1
10        for i in range(1, n + 1):
11            result *= i
12        return result
13
14 def fibonacci(n):
15     a = 0
16     b = 1
17     if n < 0:
18         return None # Negative index not allowed.
  
```

**-load file:** the **Load File** functionality simplifies file management by enabling users to quickly open and view external files within the application, streamlining workflows and boosting productivity.

This function takes only python files, copies content and pastes in the code textarea

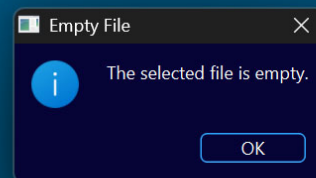
Additionally, it checks if the file is empty and displays error indictading there's no content to compile.



File is empty-----

-->

```
Code Output
1 # This is an example Python program to test the lexical analyzer.
2
3 def factorial(n):
4     if n < 0:
5         return None # Negative number, no factorial exists.
6     elif n == 0:
7         return 1
8     else:
9         result = 1
10        for i in range(1, n + 1):
11            result *= i
12        return result
13    def fibonacci(n):
14
15        a = 0
16        b = 1
17        if n < 0:
18            return None # Negative index not allowed.
19        elif n == 0:
20            return a
```



←-----File is found and copied

```

void MainWindow::loadFile()
{
    // Get file path from dialog
    QString filePath = QFileDialog::getOpenFileName(
        this,
        tr("Open Python File"),
        QDir::homePath(),
        tr("Python Files (*.py);;All Files (*)")
    );

    // If user canceled
    if (filePath.isEmpty()) {
        return;
    }

    // Check if it's a Python file
    if (!filePath.endsWith(".py", Qt::CaseInsensitive)) {
        QMessageBox::warning(this, tr("Invalid File"),
            tr("Please select a Python (.py) file.));
        return;
    }

    QFile file(filePath);

    // Check if file can be opened
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QMessageBox::critical(this, tr("Error"),
            tr("Could not open file for reading.));
        return;
    }

    // Read file content
    QTextStream in(&file);
    QString content = in.readAll();
    file.close();

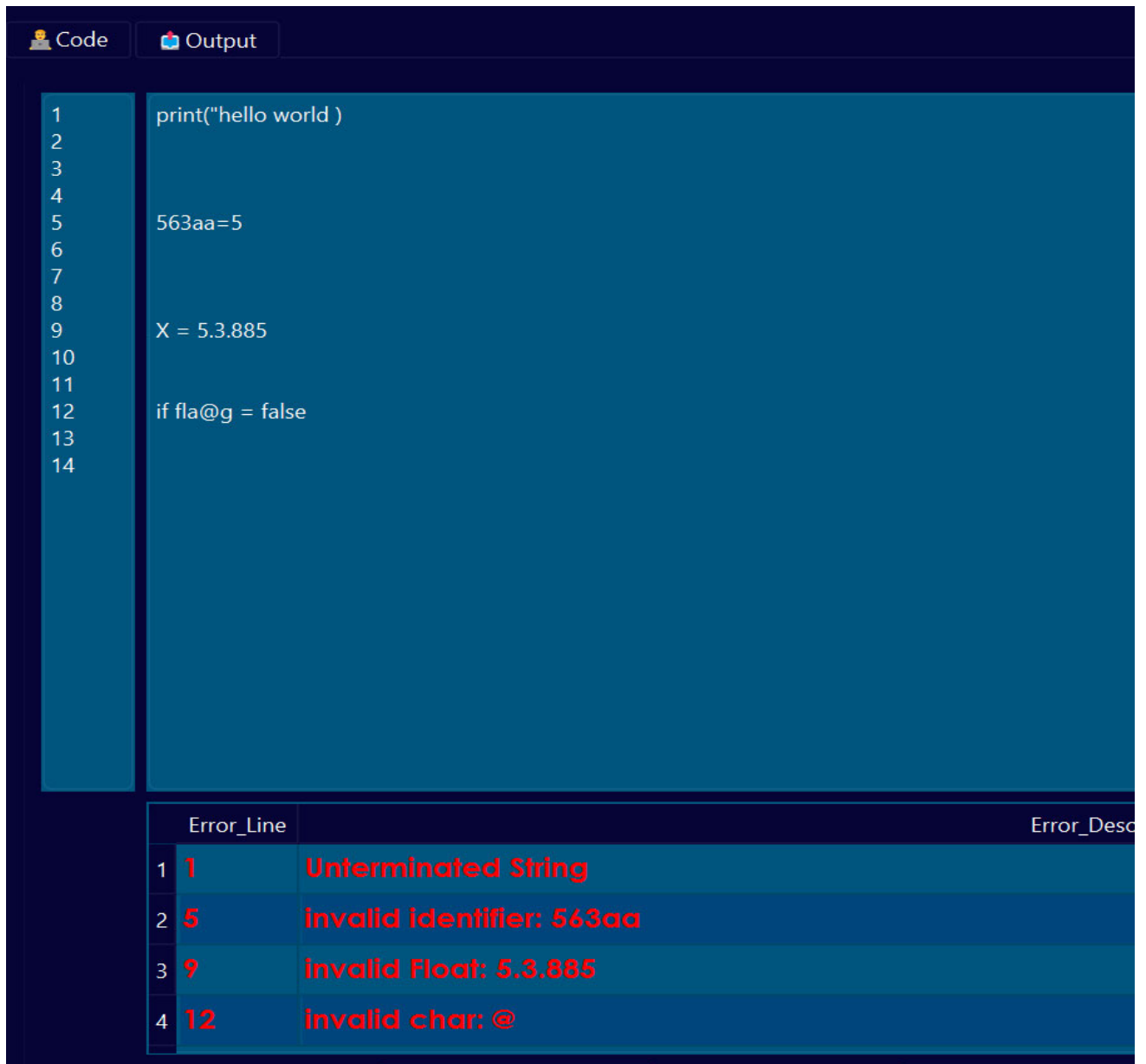
    // Check if file is empty
    if (content.isEmpty()) {
        QMessageBox::information(this, tr("Empty File"),
            tr("The selected file is empty.));
        return;
    }

    // Set text in the editor
    ui->textarea->setPlainText(content);
}

```

## 10.3 GUI error handling:

This output shows the displaying of the 4 kinds of error implemented in the code logic:



The screenshot displays a code editor with a dark theme. The top bar has two tabs: 'Code' (active) and 'Output'. The code editor shows a Python script with four lines of code, each with a line number on the left:

```
1 print("hello world )  
2  
3  
4  
5 563aa=5  
6  
7  
8  
9 X = 5.3.885  
10  
11  
12 if fla@g = false  
13  
14
```

Below the code editor, there is a table with two columns: 'Error\_Line' and 'Error\_Desc'. It lists four errors:

Error_Line	Error_Desc
1 1	Unterminated String
2 5	invalid Identifier: 563aa
3 9	invalid Float: 5.3.885
4 12	invalid char: @

# 11. Test Scenario: -

Here is a test python code and its output tokens and symbol table:

```
Code Output
1 # This is an example Python program to test the lexical analyzer.
2
3 def factorial(n):
4     if n < 0:
5         return None # Negative number, no factorial exists.
6     elif n == 0:
7         return 1
8     else:
9         result = 1
10        for i in range(1, n + 1):
11            result *= i
12        return result
13
14 def fibonacci(n):
15     a = 0
16     b = 1
17     if n < 0:
18         return None # Negative index not allowed.
19     elif n == 0:
20         return a
21     elif n == 1:
22         return b
23     else:
24         for i in range(2, n + 1):
25             a, b = b, a + b
26         return b
27
28 a = 42
29 b = 3.14
30 c = 10.
31 d = .5
32
33 s1 = "Hello, world!"
34 s2 = "Line1\nLine2"
35 s3 = "Quote: \"inner\""
36 s4 = 'C:\path\to\file'
37 s5 = 'It\'s fine'
38
39 def main():
40     print("Factorial of 5:", factorial(5))
41     print("Fibonacci of 10:", fibonacci(10))
42
43 Error_Line
```

```
Code Output
40 print("Fibonacci of 10:", fibonacci(10))
41
42 # Testing with floating point numbers.
43 pi = 3.14159
44 e = 2.71828
45 print("Sum of pi and e:", pi + e)
46
47 utils = MathUtils()
48 nums = [1, 2, 3, 4, 5]
49 print("Sum of squares:", utils.sum_of_squares(nums))
50
51 # A complex arithmetic expression:
52 result = (factorial(5) + fibonacci(10)) * (pi - e) / (3.0 * 2.0)
53 print("Complex expression result:", result)
54
55 # Additional loops and conditions for testing:
56 for i in range(3):
57     if i % 2 == 0:
58         print("Even index:", i)
59     else:
60         print("Odd index:", i)
61
62 # Multiline string for testing
63
64 print("multiline_str")
65
66 res5 = "cdk" + "ddnl"
67 res6 = a
68 res7 = None
69 print("Value of x after operations:", x)
70
71 if __name__ == "__main__":
72     main()
73 x = 123
74 y = 0.001
75 msg = "Test\nEscape"
76 ch = \"
77
78
79 if_flag = True # if_flag should be ID, True currently ID (consider adding it as keyword)
80 for_count = 0
81
82 Error_Line
```

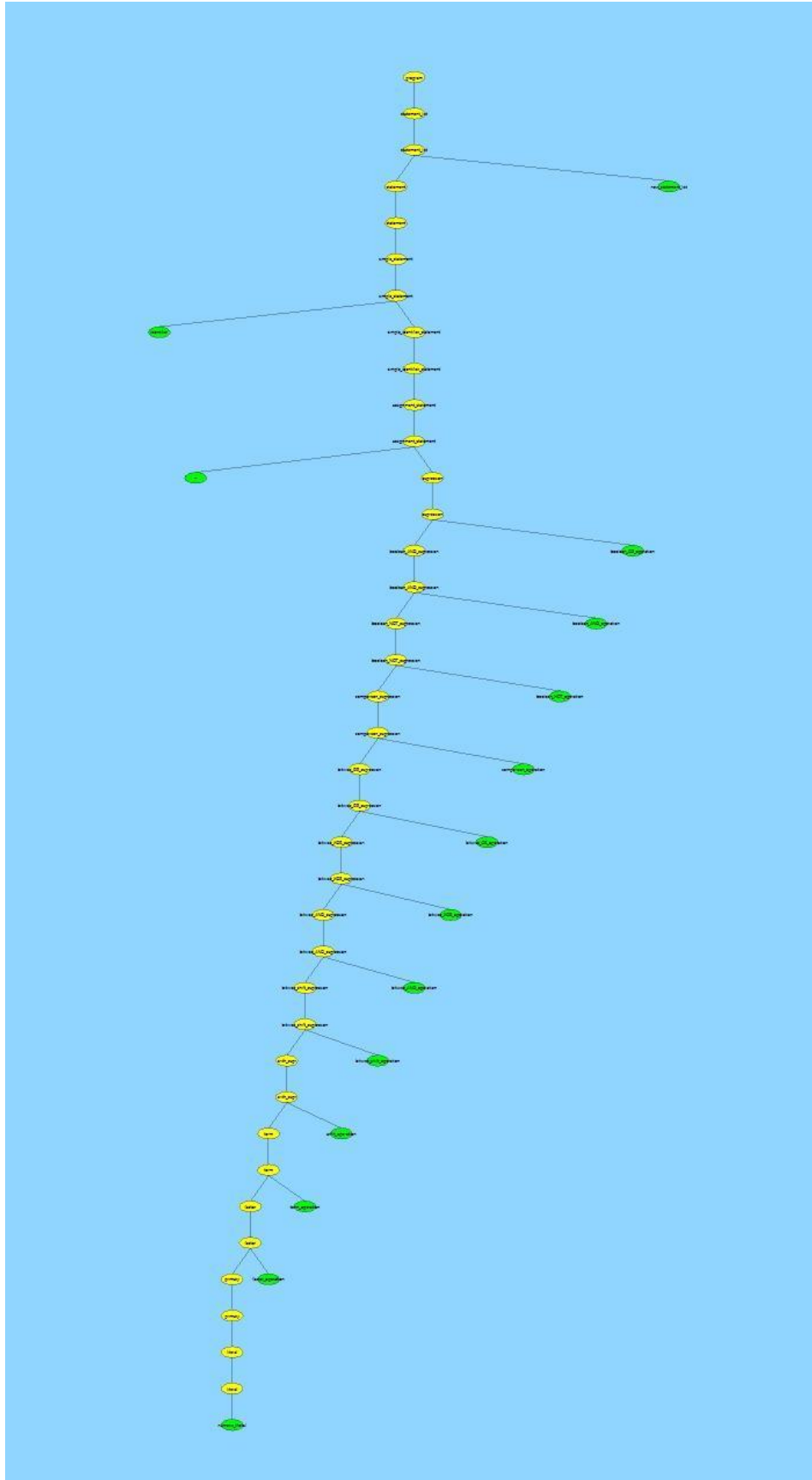


Line	Token	Value	Line	Token	Value	Type
1	IDENT	1	9	keyword		
3	IDENT	0	9	id	result	int
3	KEYWORD	def	10	id		
3	ID	0	14	keyword	fibonacci	
3	BRACKET	[	15	id	a	42
3	ID	1	16	id	b	3.14
3	BRACKET	]	16	id	c	10.
3	punctuation	:	29	id	d	5.
4	IDENT	4	30	id		
4	KEYWORD	return	32	id	s1	Hello, world!
4	ID	1	33	id	s2	Line1\nline2
4	OPERATOR	<	34	id	s3	Quote: "inner"
4	INTEGER	0	35	id	s4	C:\path\to\file
4	punctuation	:	36	id	s5	It's fine
5	IDENT	8	38	main		
5	KEYWORD	return	43	pi		3.14159
5	KEYWORD	None	44	e		2.71828
6	IDENT	4	47	utils		
6	KEYWORD	def	47	MathUtils		
6	ID	1	48	nums		
6	OPERATOR	==	49	sum_of_squares		
6	INTEGER	0	46	res5		
6	punctuation	:	47	res6		
7	IDENT	8	48	res7		None
7	KEYWORD	return	49	x		123
7	INTEGER	1	74	y		0.001
8	IDENT	4	75	msg		Test\nescape
8	KEYWORD	else	76	ch		.

Line	Token	Value	Line	Token	Value	Type
9	punctuation	:	9	result	1	int
9	IDENT	0	10	id		
9	ID	2	14	keyword	fibonacci	
9	OPERATOR	=	15	id	a	42
9	INTEGER	1	16	id	b	3.14
10	KEYWORD	for	16	id	c	10.
10	ID	3	20	id	d	5.
10	KEYWORD	in	32	id	s1	Hello, world!
10	KEYWORD	range	33	id	s2	Line1\nline2
10	BRACKET	[	34	id	s3	Quote: "inner"
10	INTEGER	1	35	id	s4	C:\path\to\file
10	punctuation	:	36	id	s5	It's fine
10	ID	1	38	main		
10	OPERATOR	+	43	pi		3.14159
10	INTEGER	1	44	e		2.71828
10	BRACKET	]	47	utils		
10	punctuation	:	47	MathUtils		
11	IDENT	12	48	nums		
11	ID	2	49	sum_of_squares		
11	OPERATOR	~	46	res5		
11	ID	3	47	res6		
12	IDENT	8	48	res7		None
12	KEYWORD	return	49	x		123
12	ID	2	74	y		0.001
14	IDENT	0	75	msg		Test\nescape
14	IDENT	0	76	ch		.
14	KEYWORD	def	79	if_flag		True
14	ID	4	80	for_count		0

Line	Token	Value	Line	Token	Value	Type
14	ID	4	9	result	1	int
14	BRACKET	[	10	id		
14	ID	1	14	keyword	fibonacci	
14	BRACKET	]	15	id	a	42
14	punctuation	:	16	id	b	3.14
16	IDENT	4	16	id	c	10.
16	ID	5	20	id	d	5.
16	OPERATOR	=	32	id	s1	Hello, world!
16	INTEGER	0	33	id	s2	Line1\nline2
16	ID	4	34	id	s3	Quote: "inner"
16	OPERATOR	=	35	id	s4	C:\path\to\file
16	INTEGER	1	36	id	s5	It's fine
17	KEYWORD	return	38	main		
17	ID	1	43	pi		3.14159
17	OPERATOR	<	44	e		2.71828
17	INTEGER	0	47	utils		
17	punctuation	:	47	MathUtils		
18	IDENT	8	48	nums		
18	KEYWORD	return	49	sum_of_squares		
18	KEYWORD	None	46	res5		
19	IDENT	4	47	res6		
19	KEYWORD	def	48	res7		None
19	ID	1	49	x		123
19	OPERATOR	==	74	y		0.001
19	INTEGER	0	75	msg		Test\nescape
19	punctuation	:	76	ch		.
20	IDENT	8	79	if_flag		True
20	KEYWORD	return	80	for_count		0

The output token is continued to 380 lines of tokens... it can be found in the .txt file attached to the project folder



## **12. Conclusion: -**

In conclusion, this project successfully demonstrates the foundational stage of compiler construction through a fully functional lexical analyzer for the Python programming language. By accurately identifying and categorizing tokens, managing indentation levels, maintaining a dynamic symbol table, and effectively capturing lexical errors, the analyzer provides a structured and reliable representation of the source code. These elements are essential for enabling the next stages of compilation, such as syntax and semantic analysis. Moreover, the integration of a **Qt-based graphical user interface (GUI)** enhances user interaction by providing a clear and intuitive platform for entering code, visualizing tokens, and reviewing errors in real time. This combination of backend functionality and frontend usability reflects a comprehensive understanding of both compiler theory and practical software development. Overall, the project serves as a solid starting point in the journey of building a full-fledged compiler.

### **13. References: -**

- 1) **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006).** *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley
- 2) **Appel, A. W. (2004).** *Modern Compiler Implementation in C/Java/ML* (2nd ed.). Cambridge University Press.
- 3) **Sethi, R. (1996).** *Programming Languages: Concepts and Constructs* (2nd ed.). Addison-Wesley.
- 4) **Blanchette, J., & Summerfield, M. (2008).** *C++ GUI Programming with Qt 4* (2nd ed.). Prentice Hall.
- 5) **The Qt Company. (n.d.).** *Qt Documentation*.  
<https://doc.qt.io/>
- 6) **GeeksforGeeks. (n.d.).** *Compiler Design - Lexical Analysis*.  
<https://www.geeksforgeeks.org/compiler-design-lexical-analysis/>