CC Lab 8

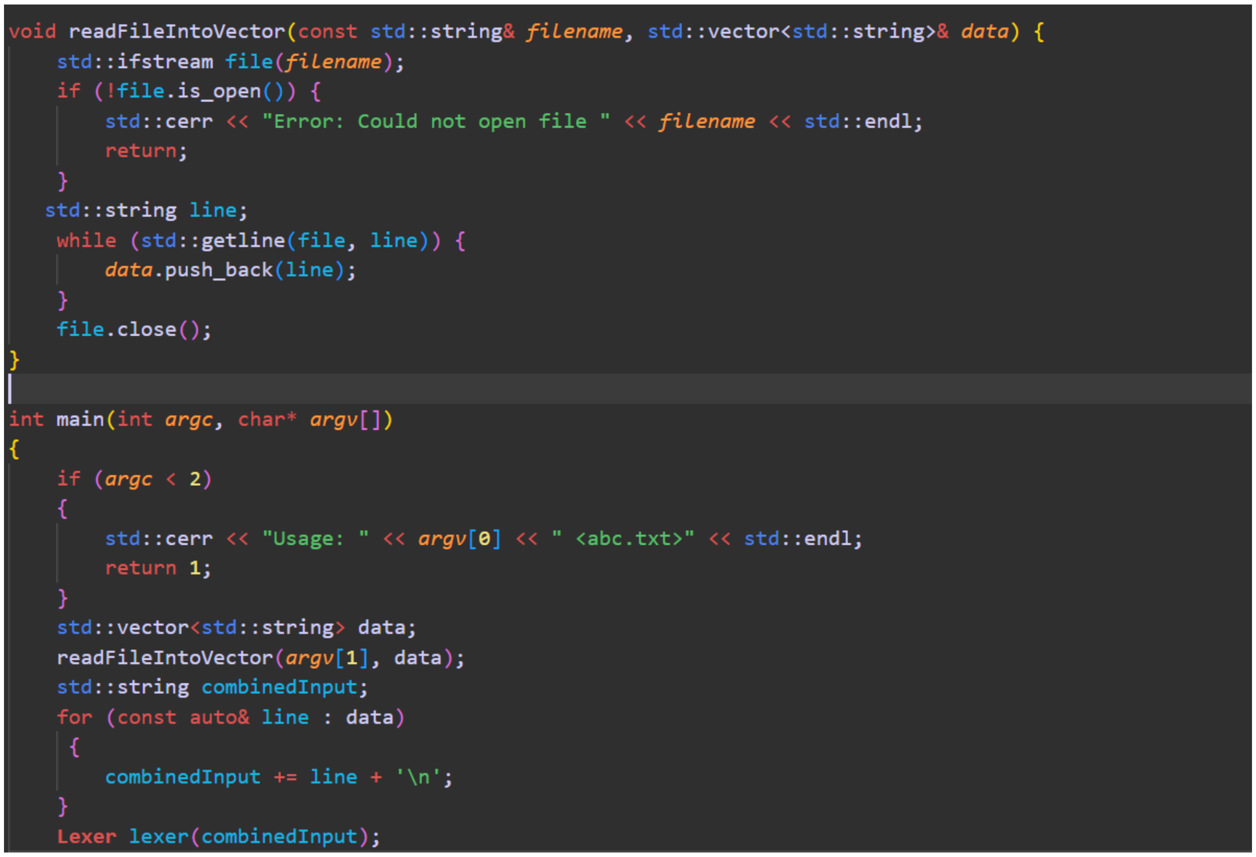
Session: 2021-2025

Submitted by:
Aleeza Shakeel   2021-CS-15

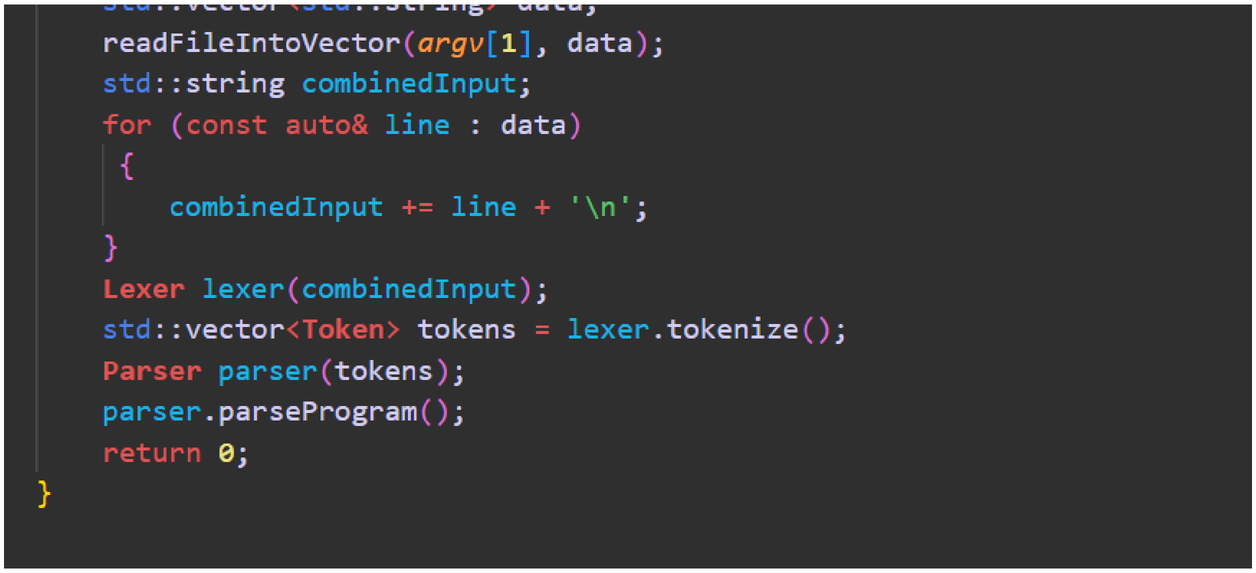
Submitted to :
Sir Laeeq khan Niazi

Department of Computer Science
University of Engineering and Technology
Lahore Pakistan

Task 1:
Turn this code like passing the file name from cmd (Done in lab1) and take that code and pass accordingly.

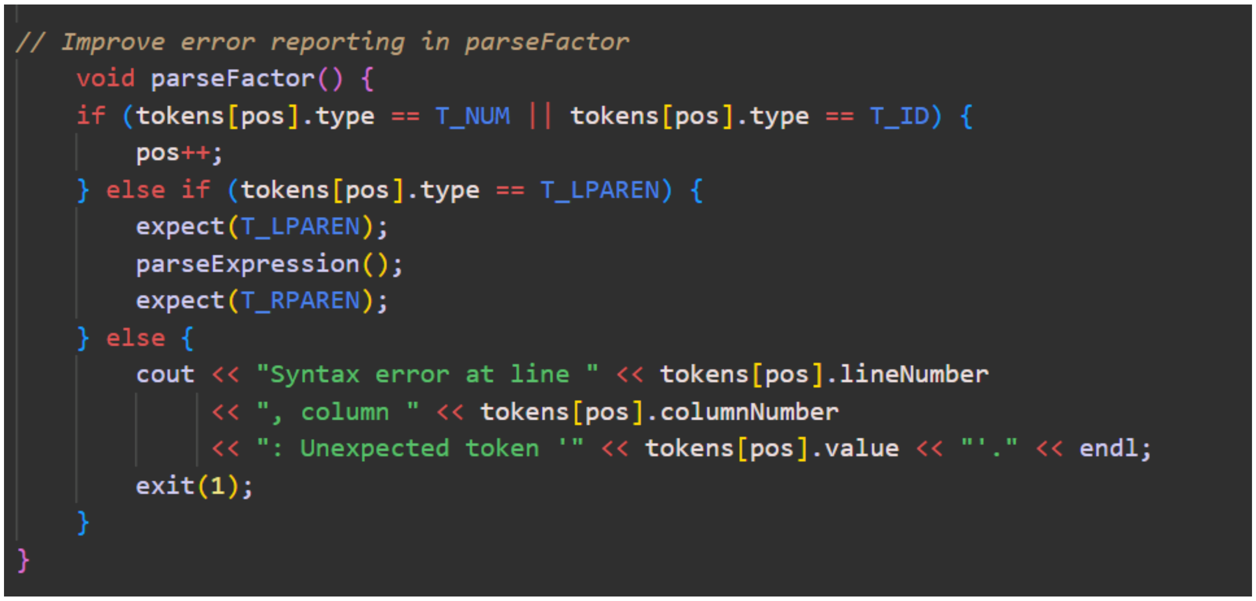


```cpp
void readFileIntoVector(const std::string& filename, std::vector<std::string>& data) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error: Could not open file " << filename << std::endl;
        return;
    }
   std::string line;
    while (std::getline(file, line)) {
        data.push_back(line);
    }
    file.close();
}

int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        std::cerr << "Usage: " << argv[0] << " <abc.txt>" << std::endl;
        return 1;
    }
    std::vector<std::string> data;
    readFileIntoVector(argv[1], data);
    std::string combinedInput;
    for (const auto& line : data)
     {
        combinedInput += line + '\n';
    }
    Lexer lexer(combinedInput);
```

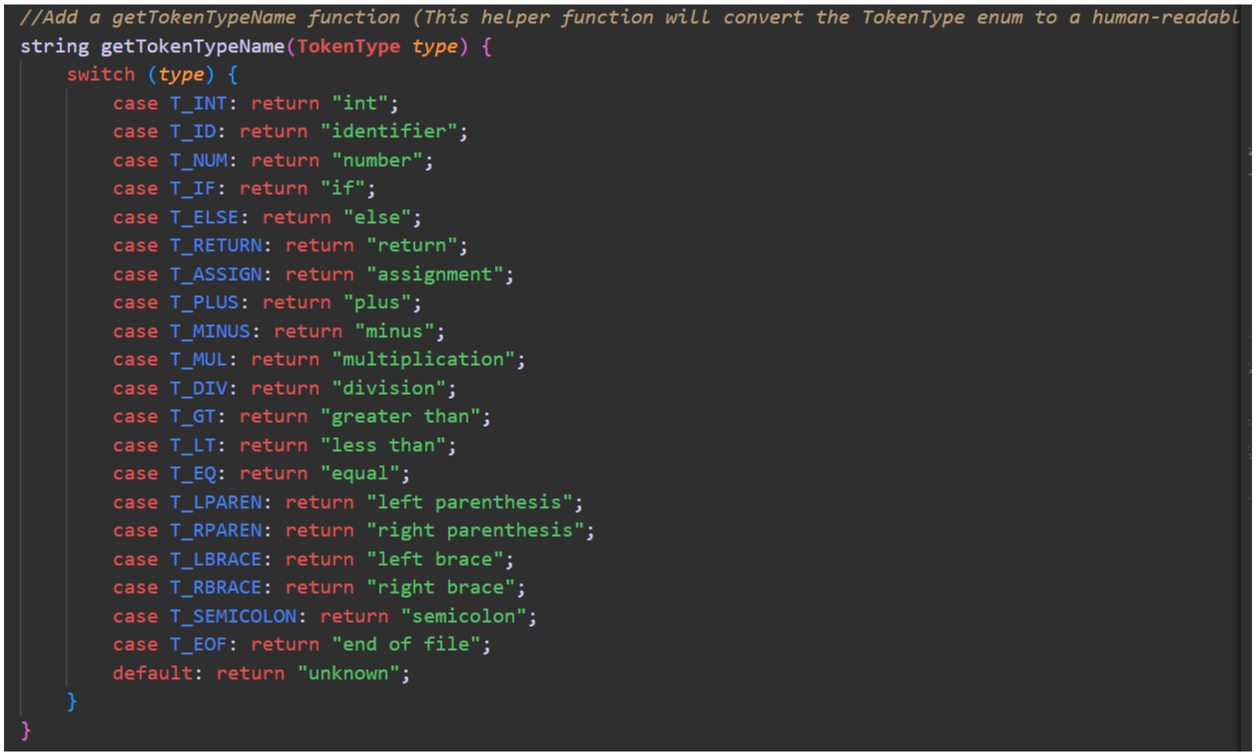


```cpp
    readFileIntoVector(argv[1], data);
    std::string combinedInput;
    for (const auto& line : data)
     {
        combinedInput += line + '\n';
    }
    Lexer lexer(combinedInput);
    std::vector<Token> tokens = lexer.tokenize();
    Parser parser(tokens);
    parser.parseProgram();
    return 0;
}
```

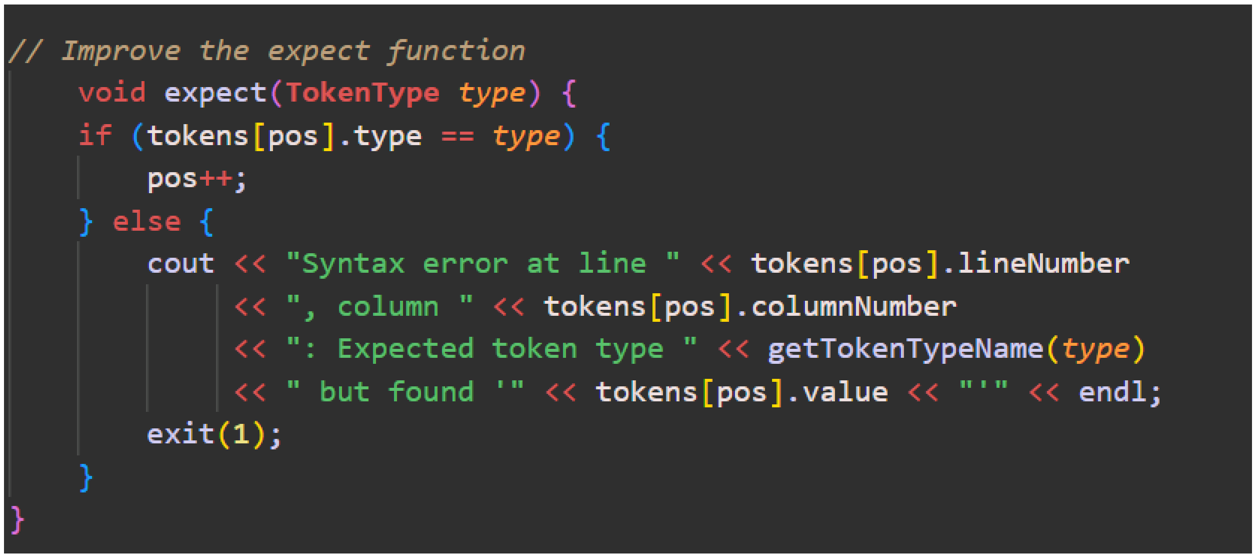
Task 2:
Making your language errors more human friendly to display the line number of the error.



```cpp
// Improve error reporting in parseFactor
    void parseFactor() {
    if (tokens[pos].type == T_NUM || tokens[pos].type == T_ID) {
        pos++;
    } else if (tokens[pos].type == T_LPAREN) {
        expect(T_LPAREN);
        parseExpression();
        expect(T_RPAREN);
    } else {
        cout << "Syntax error at line " << tokens[pos].lineNumber
             << ", column " << tokens[pos].columnNumber
             << ": Unexpected token '" << tokens[pos].value << "'." << endl;
        exit(1);
    }
}
```



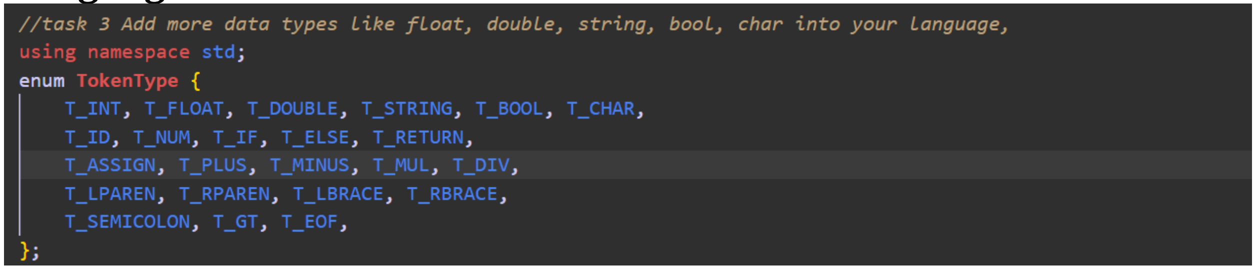
```cpp
//Add a getTokenTypeName function (This helper function will convert the TokenType enum to a human-readabl
string getTokenTypeName(TokenType type) {
    switch (type) {
        case T_INT: return "int";
        case T_ID: return "identifier";
        case T_NUM: return "number";
        case T_IF: return "if";
        case T_ELSE: return "else";
        case T_RETURN: return "return";
        case T_ASSIGN: return "assignment";
        case T_PLUS: return "plus";
        case T_MINUS: return "minus";
        case T_MUL: return "multiplication";
        case T_DIV: return "division";
        case T_GT: return "greater than";
        case T_LT: return "less than";
        case T_EQ: return "equal";
        case T_LPAREN: return "left parenthesis";
        case T_RPAREN: return "right parenthesis";
        case T_LBRACE: return "left brace";
        case T_RBRACE: return "right brace";
        case T_SEMICOLON: return "semicolon";
        case T_EOF: return "end of file";
        default: return "unknown";
    }
}
```



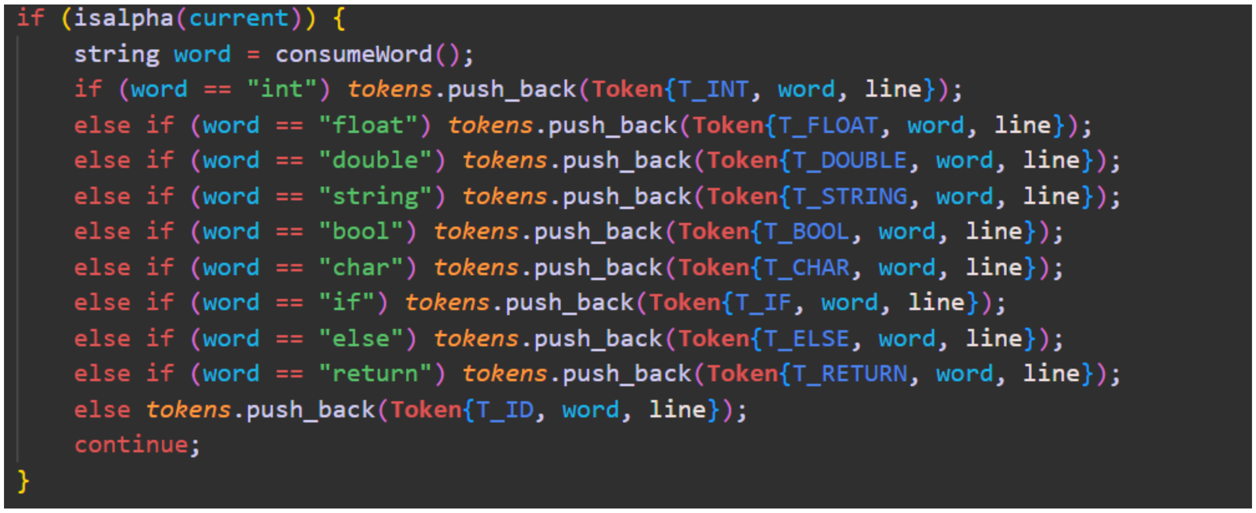
```cpp
// Improve the expect function
    void expect(TokenType type) {
    if (tokens[pos].type == type) {
        pos++;
    } else {
        cout << "Syntax error at line " << tokens[pos].lineNumber
             << ", column " << tokens[pos].columnNumber
             << ": Expected token type " << getTokenTypeName(type)
             << " but found '" << tokens[pos].value << "'" << endl;
        exit(1);
    }
}
```

Task 3:

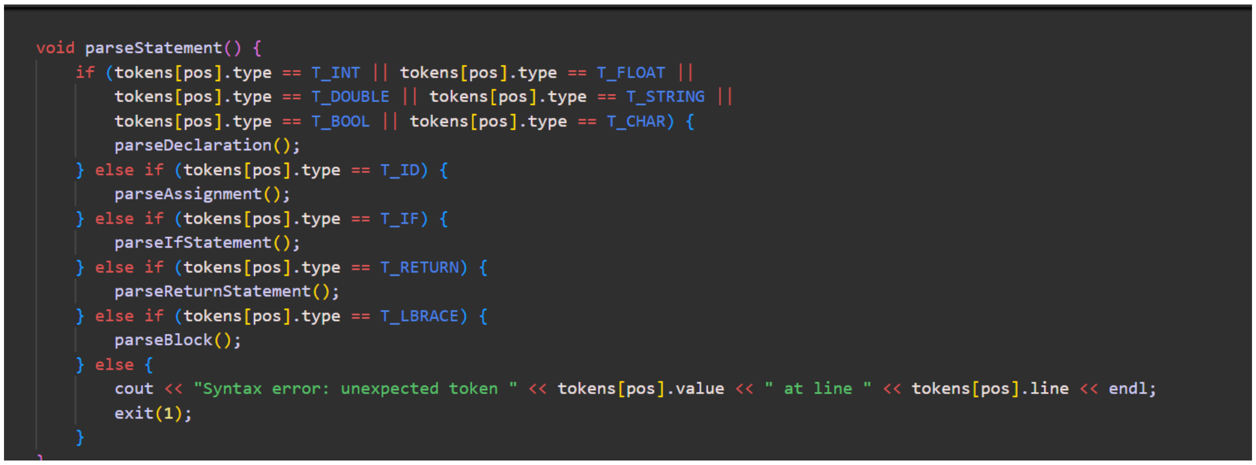
Add more data types like float, double, string, bool, char into your Language.



```cpp
//task 3 Add more data types like float, double, string, bool, char into your language,
using namespace std;
enum TokenType {
    T_INT, T_FLOAT, T_DOUBLE, T_STRING, T_BOOL, T_CHAR,
    T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE,
    T_SEMICOLON, T_GT, T_EOF,
};
```

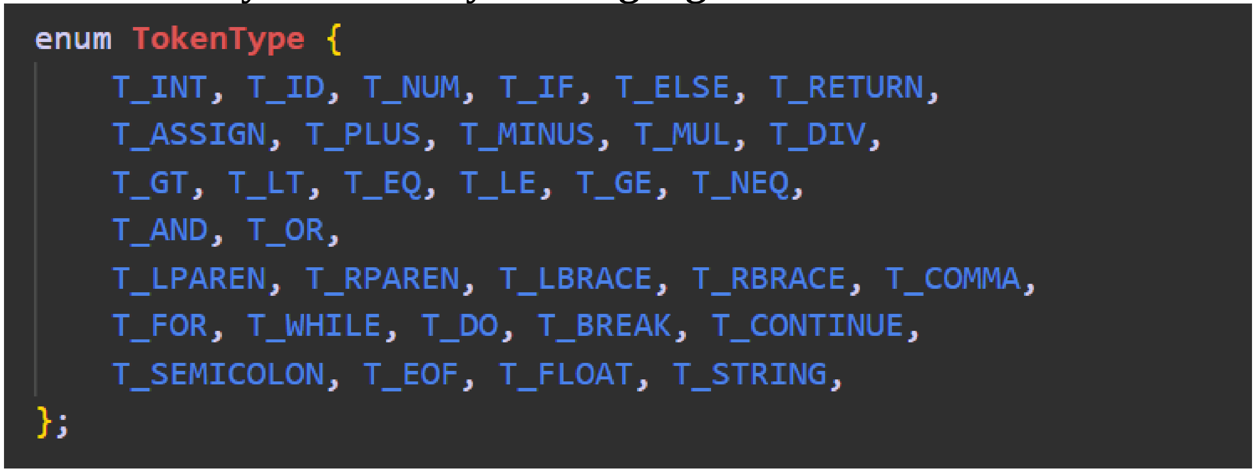


```cpp
if (isalpha(current)) {
    string word = consumeWord();
    if (word == "int") tokens.push_back(Token{T_INT, word, line});
    else if (word == "float") tokens.push_back(Token{T_FLOAT, word, line});
    else if (word == "double") tokens.push_back(Token{T_DOUBLE, word, line});
    else if (word == "string") tokens.push_back(Token{T_STRING, word, line});
    else if (word == "bool") tokens.push_back(Token{T_BOOL, word, line});
    else if (word == "char") tokens.push_back(Token{T_CHAR, word, line});
    else if (word == "if") tokens.push_back(Token{T_IF, word, line});
    else if (word == "else") tokens.push_back(Token{T_ELSE, word, line});
    else if (word == "return") tokens.push_back(Token{T_RETURN, word, line});
    else tokens.push_back(Token{T_ID, word, line});
    continue;
}
```

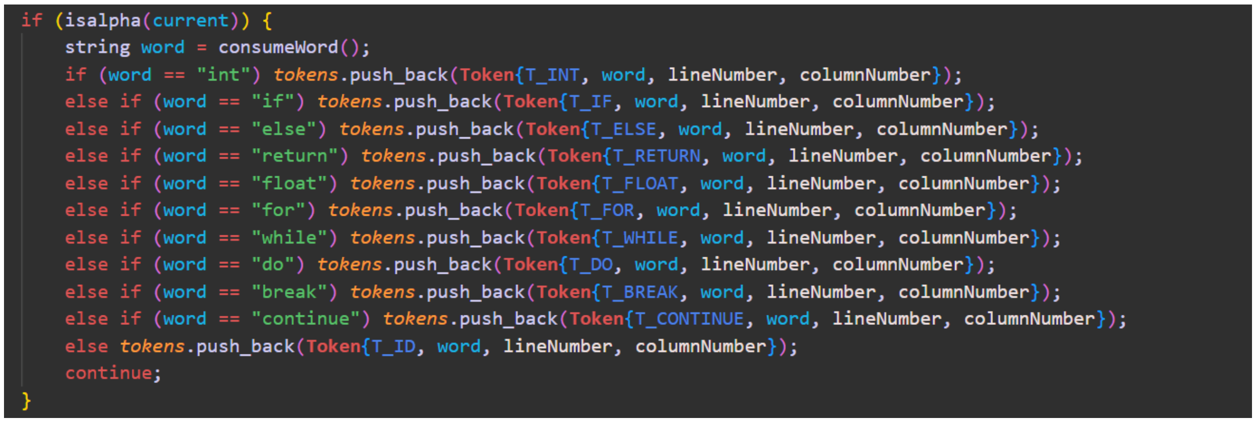


```
void parseStatement() {
    if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT ||
        tokens[pos].type == T_DOUBLE || tokens[pos].type == T_STRING ||
        tokens[pos].type == T_BOOL || tokens[pos].type == T_CHAR) {
        parseDeclaration();
    } else if (tokens[pos].type == T_ID) {
        parseAssignment();
    } else if (tokens[pos].type == T_IF) {
        parseIfStatement();
    } else if (tokens[pos].type == T_RETURN) {
        parseReturnStatement();
    } else if (tokens[pos].type == T_LBRACE) {
        parseBlock();
    } else {
        cout << "Syntax error: unexpected token " << tokens[pos].value << " at line " << tokens[pos].line << endl;
        exit(1);
    }
```

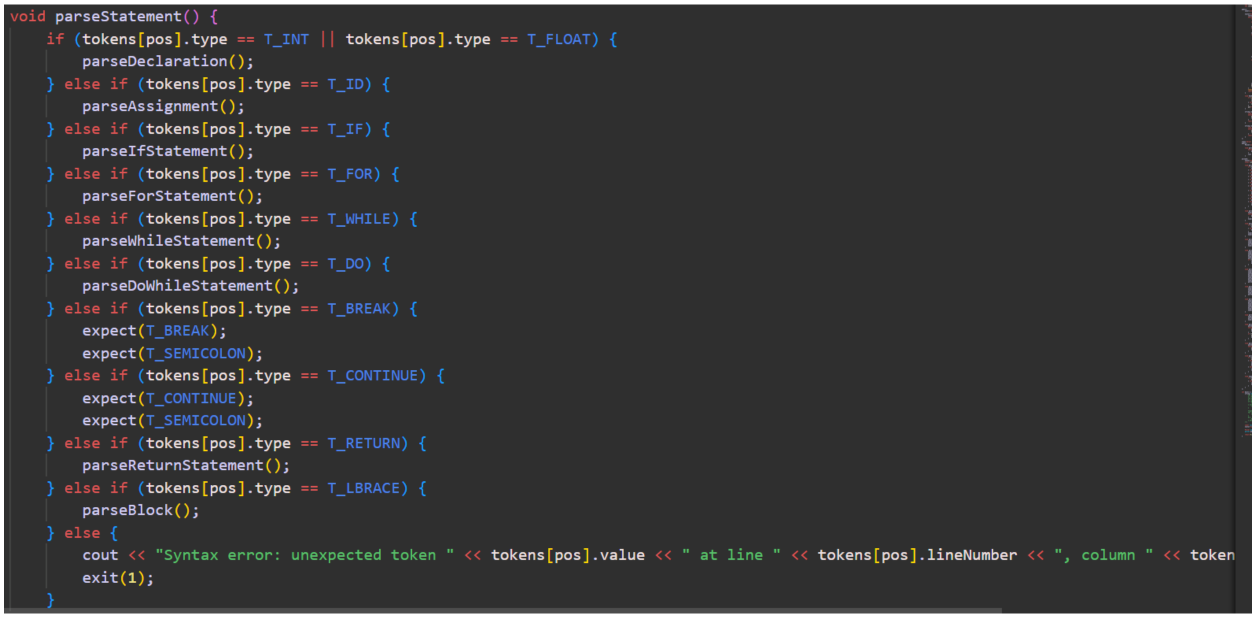
Task 4:
Add more keywords into your language



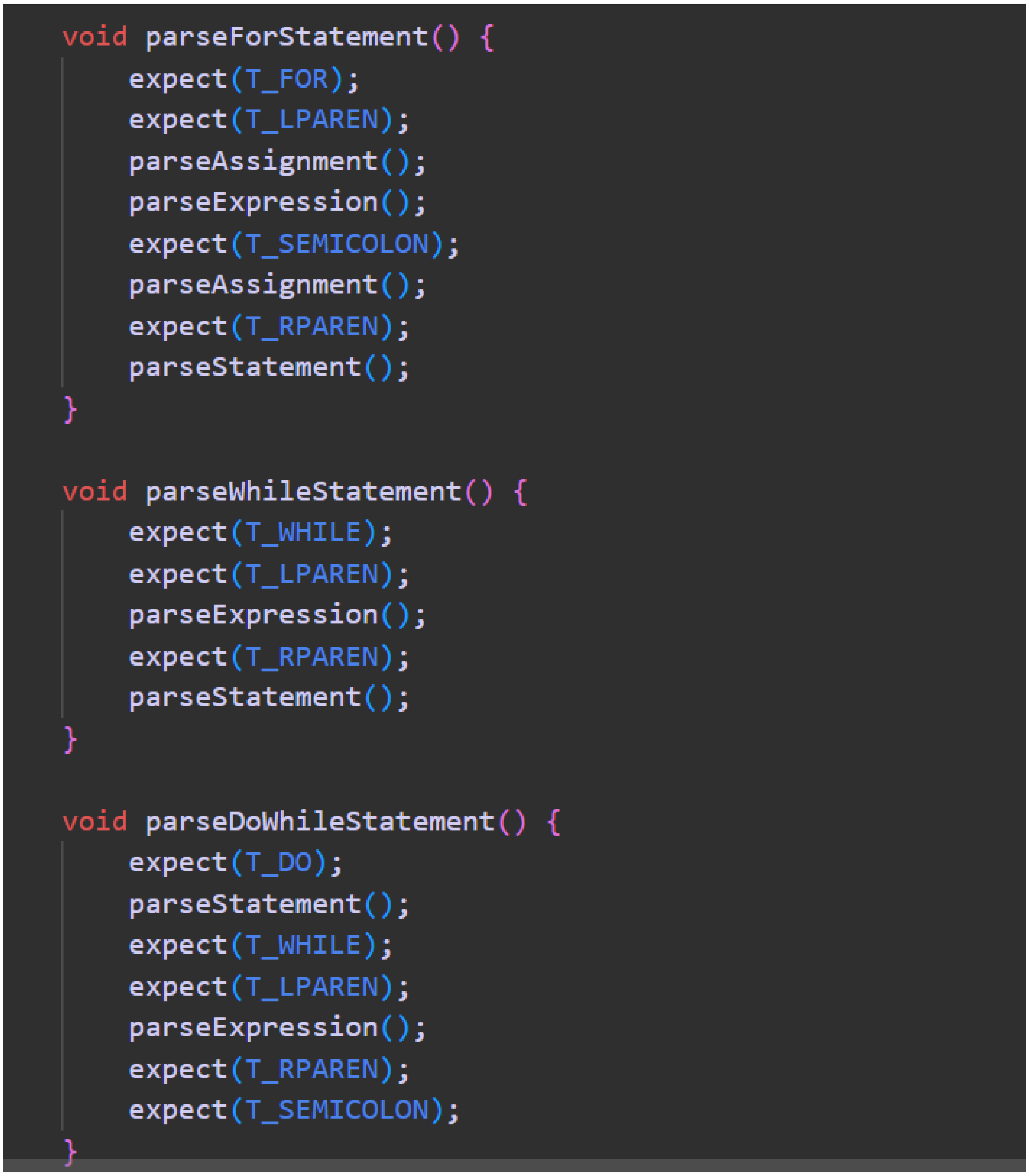
```
enum TokenType {
    T_INT, T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_GT, T_LT, T_EQ, T_LE, T_GE, T_NEQ,
    T_AND, T_OR,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE, T_COMMA,
    T_FOR, T_WHILE, T_DO, T_BREAK, T_CONTINUE,
    T_SEMICOLON, T_EOF, T_FLOAT, T_STRING,
};
```



```
if (isalpha(current)) {
    string word = consumeWord();
    if (word == "int") tokens.push_back(Token{T_INT, word, lineNumber, columnNumber});
    else if (word == "if") tokens.push_back(Token{T_IF, word, lineNumber, columnNumber});
    else if (word == "else") tokens.push_back(Token{T_ELSE, word, lineNumber, columnNumber});
    else if (word == "return") tokens.push_back(Token{T_RETURN, word, lineNumber, columnNumber});
    else if (word == "float") tokens.push_back(Token{T_FLOAT, word, lineNumber, columnNumber});
    else if (word == "for") tokens.push_back(Token{T_FOR, word, lineNumber, columnNumber});
    else if (word == "while") tokens.push_back(Token{T_WHILE, word, lineNumber, columnNumber});
    else if (word == "do") tokens.push_back(Token{T_DO, word, lineNumber, columnNumber});
    else if (word == "break") tokens.push_back(Token{T_BREAK, word, lineNumber, columnNumber});
    else if (word == "continue") tokens.push_back(Token{T_CONTINUE, word, lineNumber, columnNumber});
    else tokens.push_back(Token{T_ID, word, lineNumber, columnNumber});
    continue;
}
```



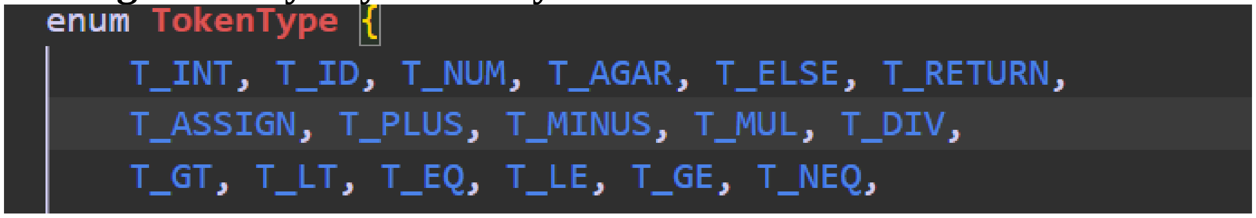
```cpp
void parseStatement() {
    if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT) {
        parseDeclaration();
    } else if (tokens[pos].type == T_ID) {
        parseAssignment();
    } else if (tokens[pos].type == T_IF) {
        parseIfStatement();
    } else if (tokens[pos].type == T_FOR) {
        parseForStatement();
    } else if (tokens[pos].type == T_WHILE) {
        parseWhileStatement();
    } else if (tokens[pos].type == T_DO) {
        parseDoWhileStatement();
    } else if (tokens[pos].type == T_BREAK) {
        expect(T_BREAK);
        expect(T_SEMICOLON);
    } else if (tokens[pos].type == T_CONTINUE) {
        expect(T_CONTINUE);
        expect(T_SEMICOLON);
    } else if (tokens[pos].type == T_RETURN) {
        parseReturnStatement();
    } else if (tokens[pos].type == T_LBRACE) {
        parseBlock();
    } else {
        cout << "Syntax error: unexpected token " << tokens[pos].value << " at line " << tokens[pos].lineNumber << ", column " << token
        exit(1);
    }
```



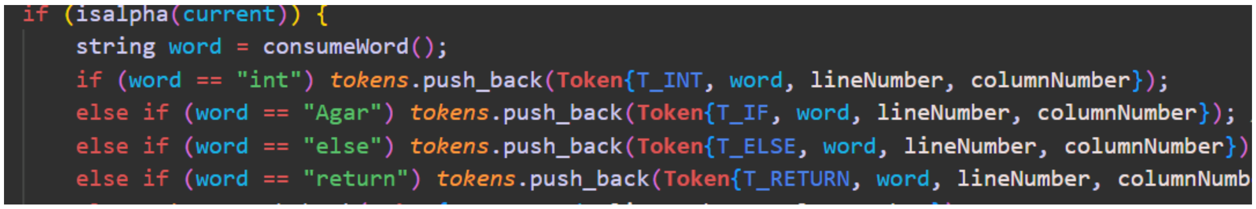
```
void parseForStatement() {
    expect(T_FOR);
    expect(T_LPAREN);
    parseAssignment();
    parseExpression();
    expect(T_SEMICOLON);
    parseAssignment();
    expect(T_RPAREN);
    parseStatement();
}

void parseWhileStatement() {
    expect(T_WHILE);
    expect(T_LPAREN);
    parseExpression();
    expect(T_RPAREN);
    parseStatement();
}

void parseDoWhileStatement() {
    expect(T_DO);
    parseStatement();
    expect(T_WHILE);
    expect(T_LPAREN);
    parseExpression();
    expect(T_RPAREN);
    expect(T_SEMICOLON);
}
```

Task 5:

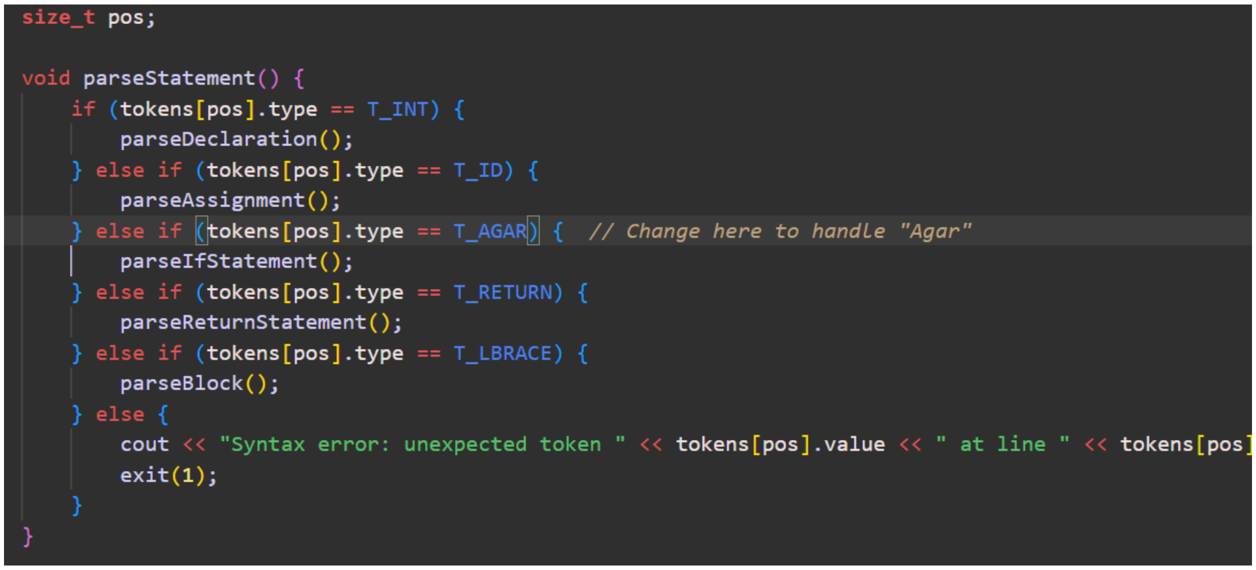
Change the structure of conditions like you can turn if statement into Agar or any keyword of your choice.



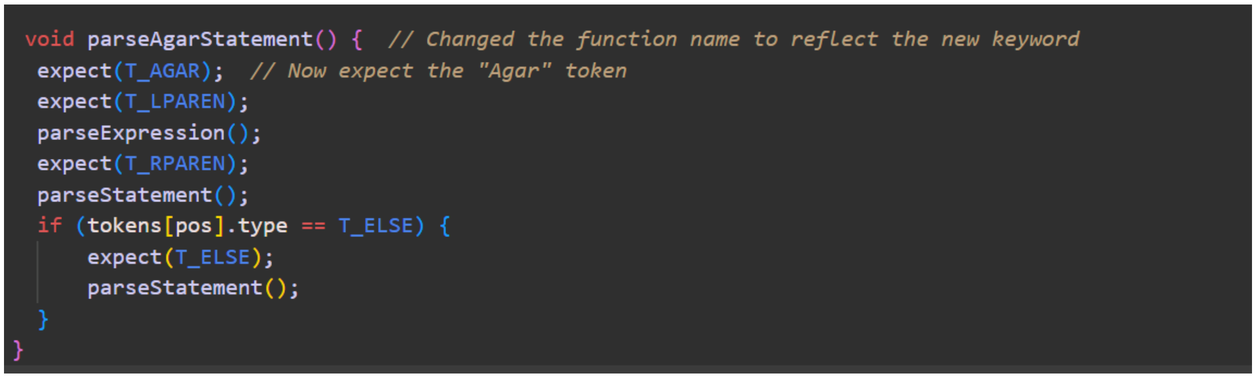
```
enum TokenType {
    T_INT, T_ID, T_NUM, T_AGAR, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_GT, T_LT, T_EQ, T_LE, T_GE, T_NEQ,
```



```
if (isalpha(current)) {
    string word = consumeWord();
    if (word == "int") tokens.push_back(Token{T_INT, word, lineNumber, columnNumber});
    else if (word == "Agar") tokens.push_back(Token{T_IF, word, lineNumber, columnNumber});
    else if (word == "else") tokens.push_back(Token{T_ELSE, word, lineNumber, columnNumber})
    else if (word == "return") tokens.push_back(Token{T_RETURN, word, lineNumber, columnNumb
```

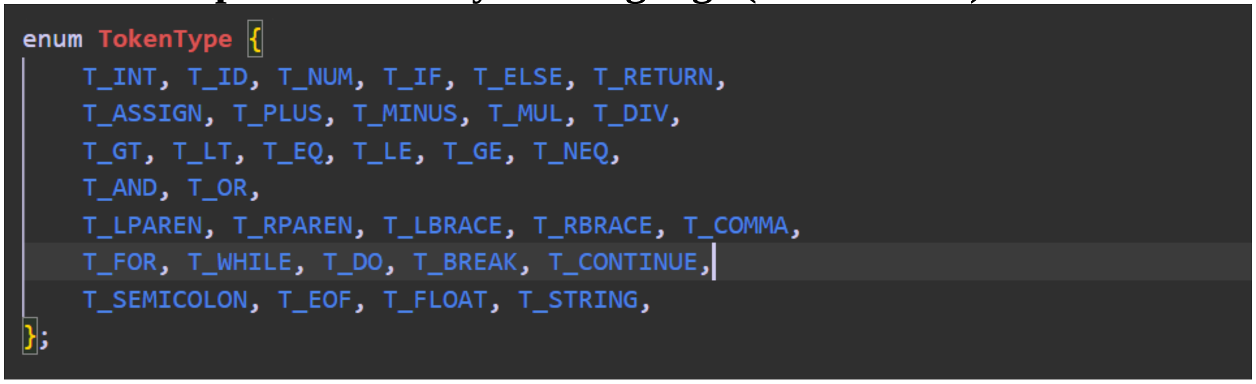


```
size_t pos;

void parseStatement() {
    if (tokens[pos].type == T_INT) {
        parseDeclaration();
    } else if (tokens[pos].type == T_ID) {
        parseAssignment();
    } else if (tokens[pos].type == T_AGAR) {  // Change here to handle "Agar"
        parseIfStatement();
    } else if (tokens[pos].type == T_RETURN) {
        parseReturnStatement();
    } else if (tokens[pos].type == T_LBRACE) {
        parseBlock();
    } else {
        cout << "Syntax error: unexpected token " << tokens[pos].value << " at line " << tokens[pos]
        exit(1);
    }
}
```

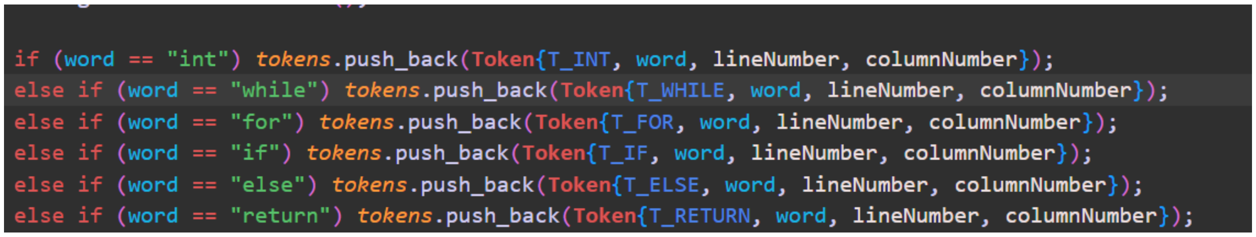


```
void parseAgarStatement() {  // Changed the function name to reflect the new keyword
 expect(T_AGAR);  // Now expect the "Agar" token
 expect(T_LPAREN);
 parseExpression();
 expect(T_RPAREN);
 parseStatement();
 if (tokens[pos].type == T_ELSE) {
     expect(T_ELSE);
     parseStatement();
 }
}
```

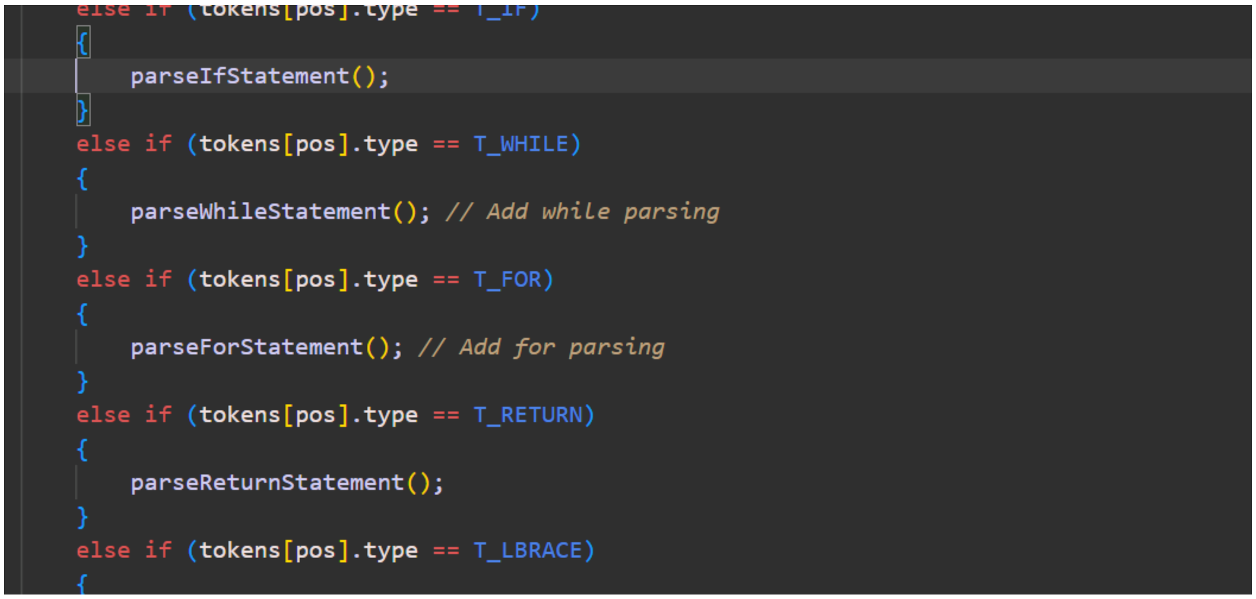
Task 6:
Add the loop feature into your language (while or for)



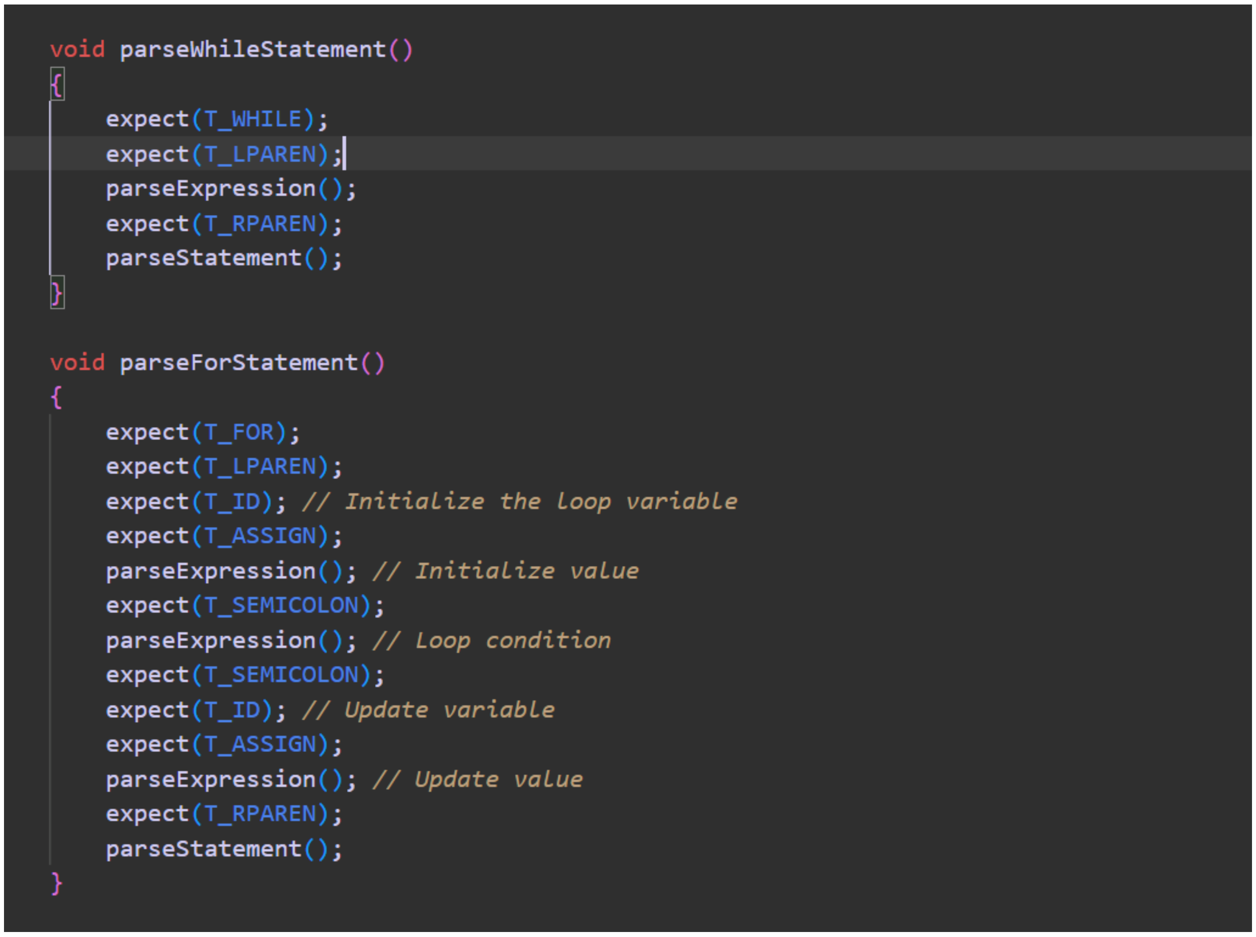
```
enum TokenType {
    T_INT, T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_GT, T_LT, T_EQ, T_LE, T_GE, T_NEQ,
    T_AND, T_OR,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE, T_COMMA,
    T_FOR, T_WHILE, T_DO, T_BREAK, T_CONTINUE,
    T_SEMICOLON, T_EOF, T_FLOAT, T_STRING,
};
```



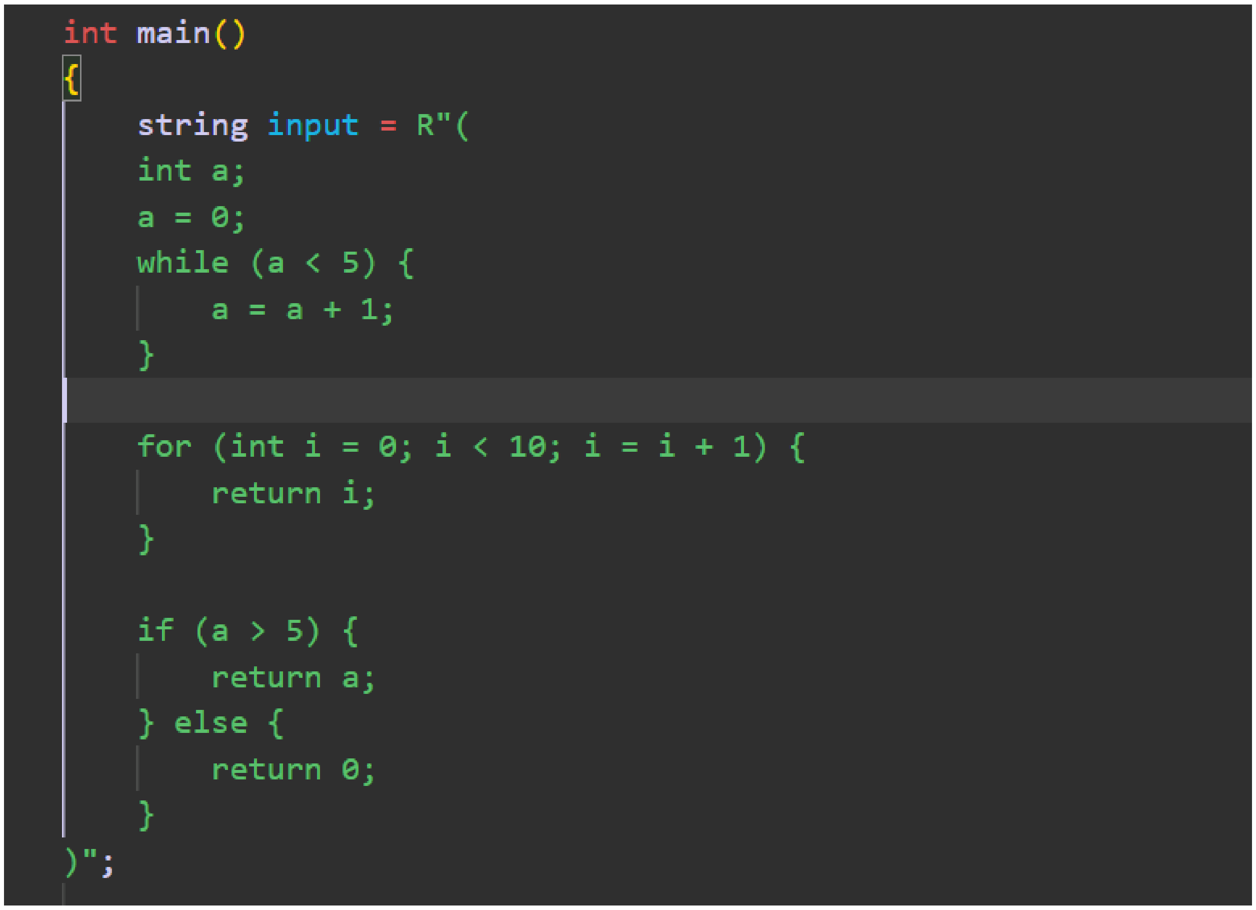
```
if (word == "int") tokens.push_back(Token{T_INT, word, lineNumber, columnNumber});
else if (word == "while") tokens.push_back(Token{T_WHILE, word, lineNumber, columnNumber});
else if (word == "for") tokens.push_back(Token{T_FOR, word, lineNumber, columnNumber});
else if (word == "if") tokens.push_back(Token{T_IF, word, lineNumber, columnNumber});
else if (word == "else") tokens.push_back(Token{T_ELSE, word, lineNumber, columnNumber});
else if (word == "return") tokens.push_back(Token{T_RETURN, word, lineNumber, columnNumber});
```



```
    else if (tokens[pos].type == T_IF)
    {
        parseIfStatement();
    }
    else if (tokens[pos].type == T_WHILE)
    {
        parseWhileStatement(); // Add while parsing
    }
    else if (tokens[pos].type == T_FOR)
    {
        parseForStatement(); // Add for parsing
    }
    else if (tokens[pos].type == T_RETURN)
    {
        parseReturnStatement();
    }
    else if (tokens[pos].type == T_LBRACE)
    {
```

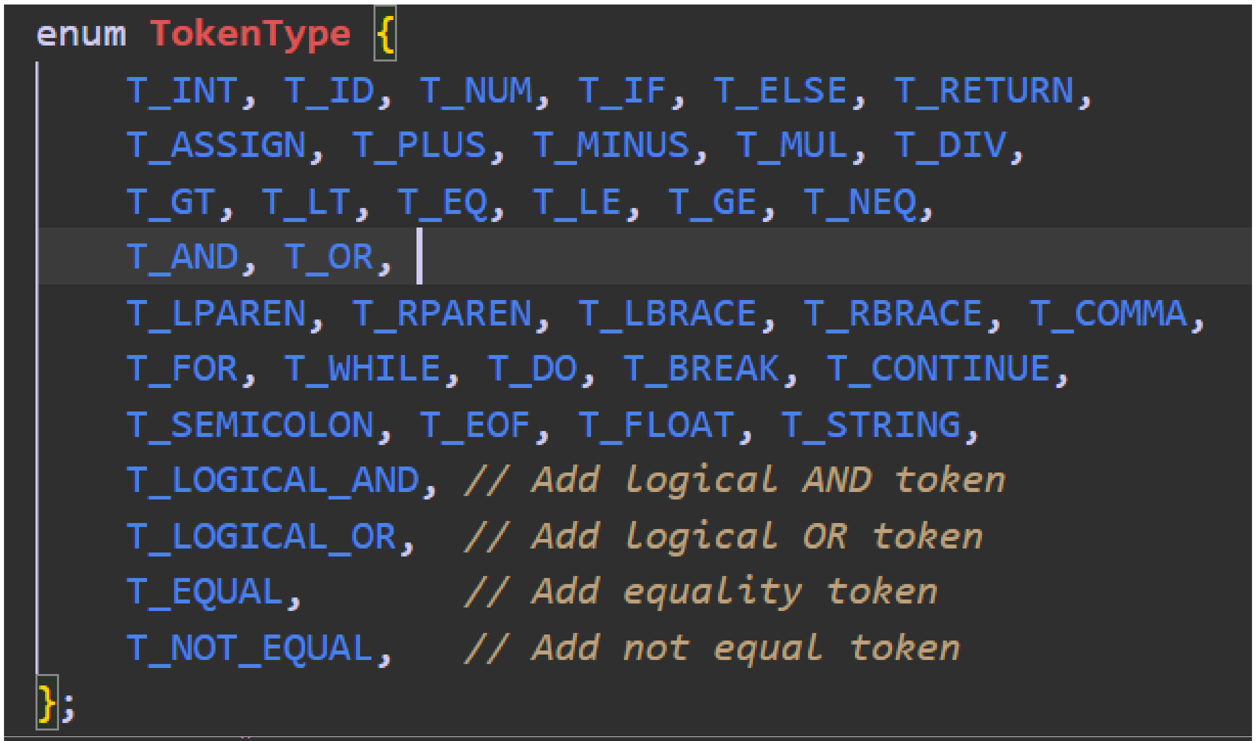


```
void parseWhileStatement()
{
    expect(T_WHILE);
    expect(T_LPAREN);
    parseExpression();
    expect(T_RPAREN);
    parseStatement();
}

void parseForStatement()
{
    expect(T_FOR);
    expect(T_LPAREN);
    expect(T_ID); // Initialize the loop variable
    expect(T_ASSIGN);
    parseExpression(); // Initialize value
    expect(T_SEMICOLON);
    parseExpression(); // Loop condition
    expect(T_SEMICOLON);
    expect(T_ID); // Update variable
    expect(T_ASSIGN);
    parseExpression(); // Update value
    expect(T_RPAREN);
    parseStatement();
}
```

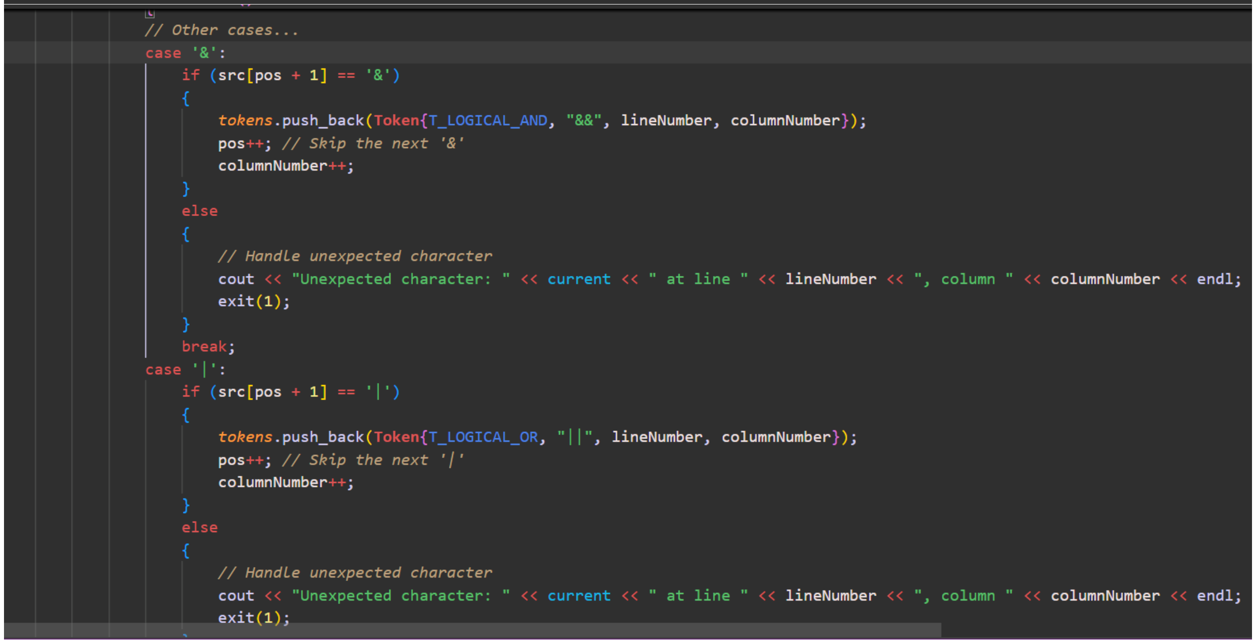


```
int main()
{
    string input = R"(
    int a;
    a = 0;
    while (a < 5) {
        a = a + 1;
    }

    for (int i = 0; i < 10; i = i + 1) {
        return i;
    }

    if (a > 5) {
        return a;
    } else {
        return 0;
    }
)";
```

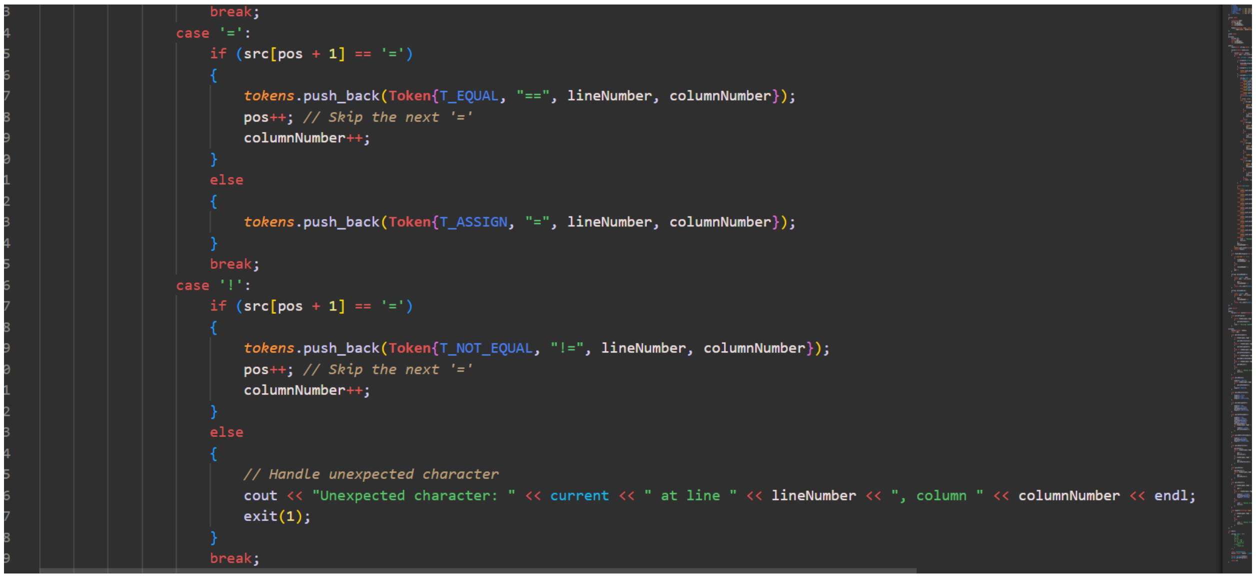
Task 7:
Add logical expressions into your language inside if conditions like &amp;&amp;, ||, == and !=



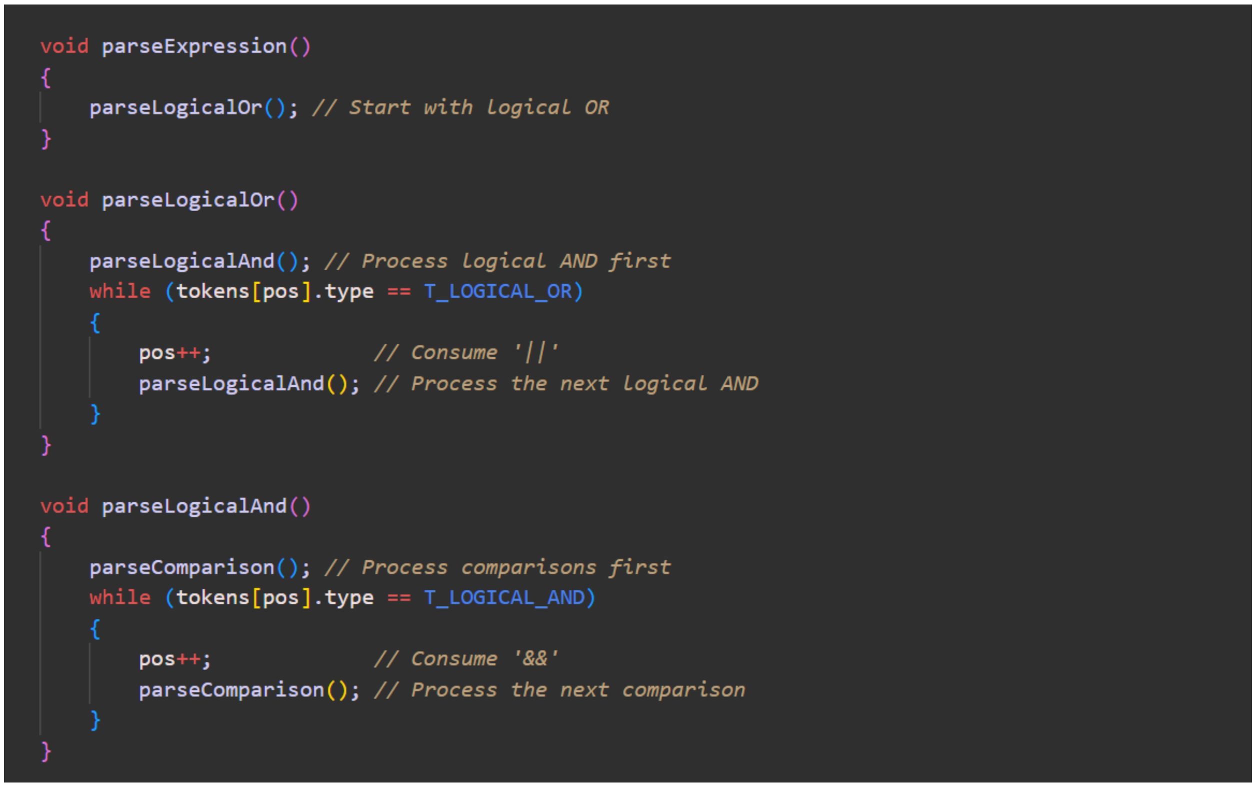
```cpp
enum TokenType {
    T_INT, T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_GT, T_LT, T_EQ, T_LE, T_GE, T_NEQ,
    T_AND, T_OR,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE, T_COMMA,
    T_FOR, T_WHILE, T_DO, T_BREAK, T_CONTINUE,
    T_SEMICOLON, T_EOF, T_FLOAT, T_STRING,
    T_LOGICAL_AND, // Add logical AND token
    T_LOGICAL_OR,  // Add logical OR token
    T_EQUAL,       // Add equality token
    T_NOT_EQUAL,   // Add not equal token
};
```



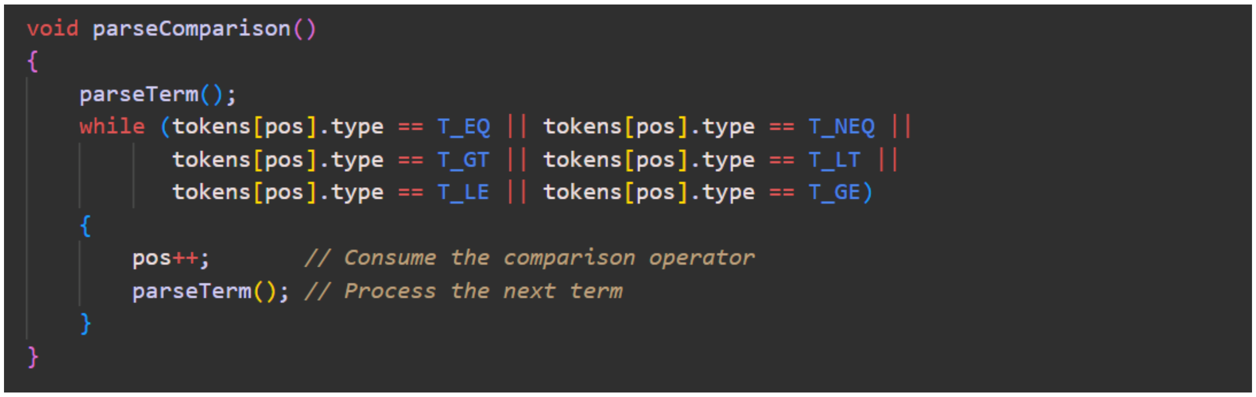
```cpp
// Other cases...
case '&':
    if (src[pos + 1] == '&')
    {
        tokens.push_back(Token{T_LOGICAL_AND, "&&", lineNumber, columnNumber});
        pos++; // Skip the next '&'
        columnNumber++;
    }
    else
    {
        // Handle unexpected character
        cout << "Unexpected character: " << current << " at line " << lineNumber << ", column " << columnNumber << endl;
        exit(1);
    }
    break;
case '|':
    if (src[pos + 1] == '|')
    {
        tokens.push_back(Token{T_LOGICAL_OR, "||", lineNumber, columnNumber});
        pos++; // Skip the next '|'
        columnNumber++;
    }
    else
    {
        // Handle unexpected character
        cout << "Unexpected character: " << current << " at line " << lineNumber << ", column " << columnNumber << endl;
        exit(1);
```



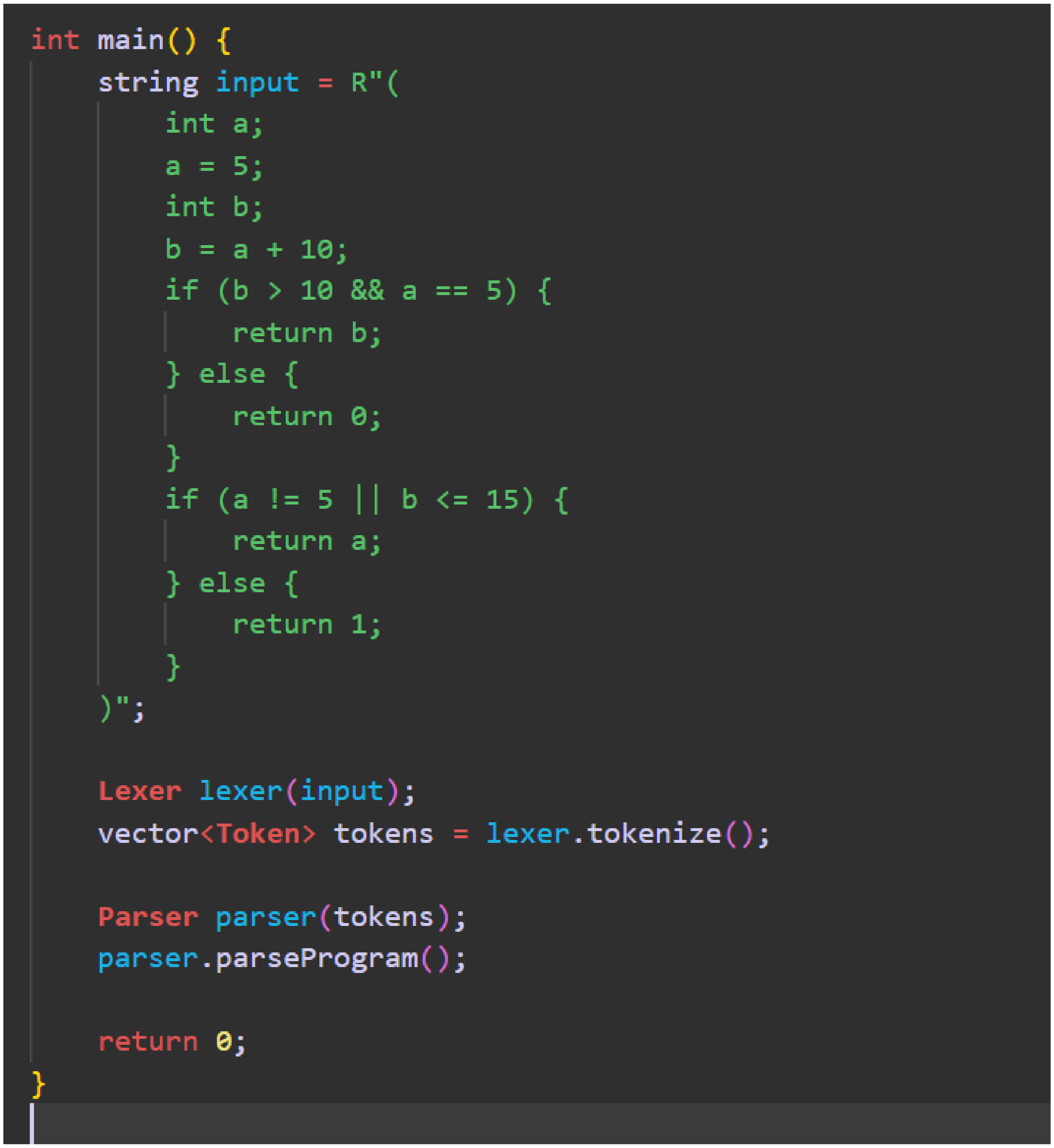
```
            break;
        case '=':
            if (src[pos + 1] == '=')
            {
                tokens.push_back(Token{T_EQUAL, "==", lineNumber, columnNumber});
                pos++; // Skip the next '='
                columnNumber++;
            }
            else
            {
                tokens.push_back(Token{T_ASSIGN, "=", lineNumber, columnNumber});
            }
            break;
        case '!':
            if (src[pos + 1] == '=')
            {
                tokens.push_back(Token{T_NOT_EQUAL, "!=", lineNumber, columnNumber});
                pos++; // Skip the next '='
                columnNumber++;
            }
            else
            {
                // Handle unexpected character
                cout << "Unexpected character: " << current << " at line " << lineNumber << ", column " << columnNumber << endl;
                exit(1);
            }
            break;
```



```
void parseExpression()
{
    parseLogicalOr(); // Start with logical OR
}

void parseLogicalOr()
{
    parseLogicalAnd(); // Process logical AND first
    while (tokens[pos].type == T_LOGICAL_OR)
    {
        pos++;              // Consume '||'
        parseLogicalAnd(); // Process the next logical AND
    }
}

void parseLogicalAnd()
{
    parseComparison(); // Process comparisons first
    while (tokens[pos].type == T_LOGICAL_AND)
    {
        pos++;              // Consume '&&'
        parseComparison(); // Process the next comparison
    }
}
```



```
void parseComparison()
{
    parseTerm();
    while (tokens[pos].type == T_EQ || tokens[pos].type == T_NEQ ||
           tokens[pos].type == T_GT || tokens[pos].type == T_LT ||
           tokens[pos].type == T_LE || tokens[pos].type == T_GE)
    {
        pos++;       // Consume the comparison operator
        parseTerm(); // Process the next term
    }
}
```



```cpp
int main() {
    string input = R"(
        int a;
        a = 5;
        int b;
        b = a + 10;
        if (b > 10 && a == 5) {
            return b;
        } else {
            return 0;
        }
        if (a != 5 || b <= 15) {
            return a;
        } else {
            return 1;
        }
    )";

    Lexer lexer(input);
    vector<Token> tokens = lexer.tokenize();

    Parser parser(tokens);
    parser.parseProgram();

    return 0;
}
```