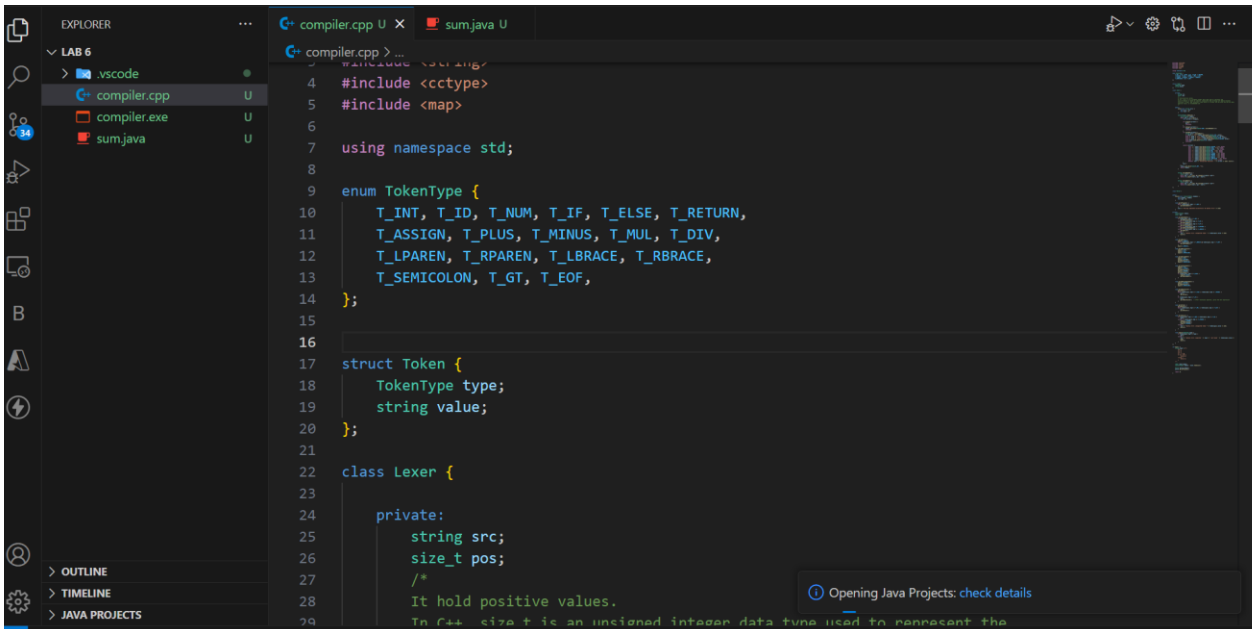
Compiler Construction Cs-471

Student Name: Fatiq Hussnain

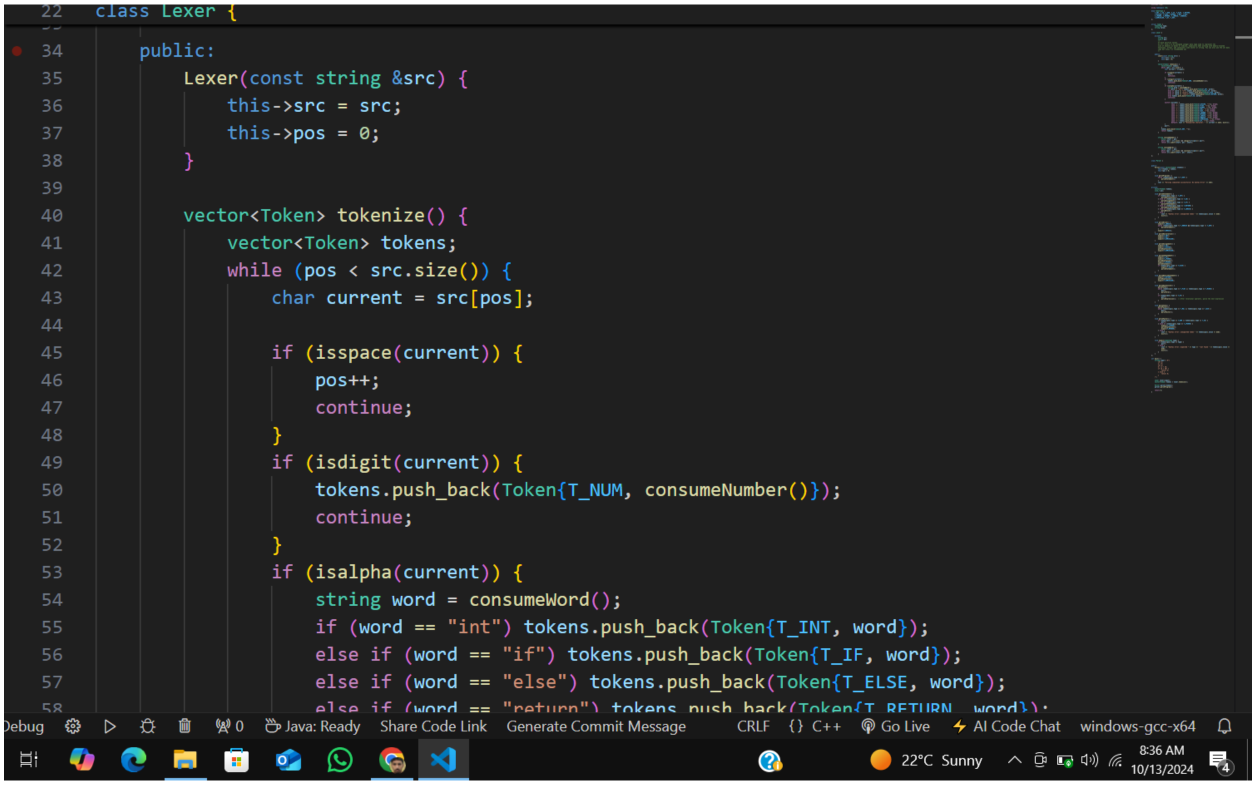
Instructor: Mr. Laeeq khan Niazi

Lab 6

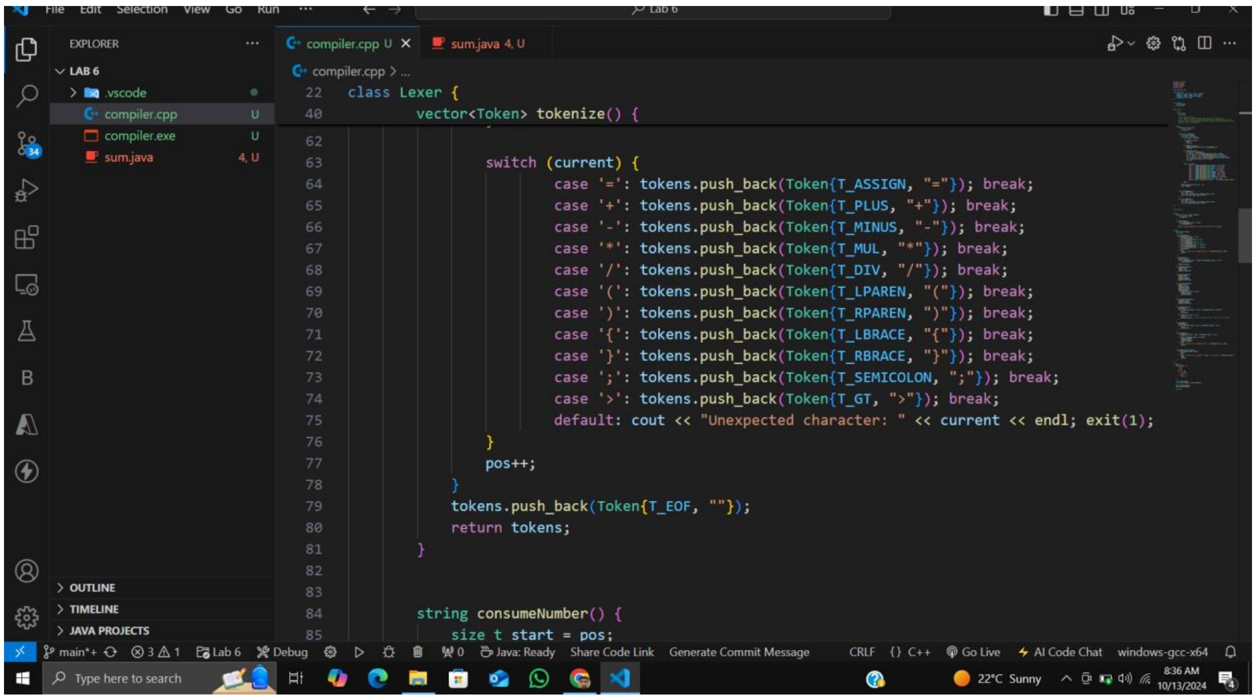
Code Snippets



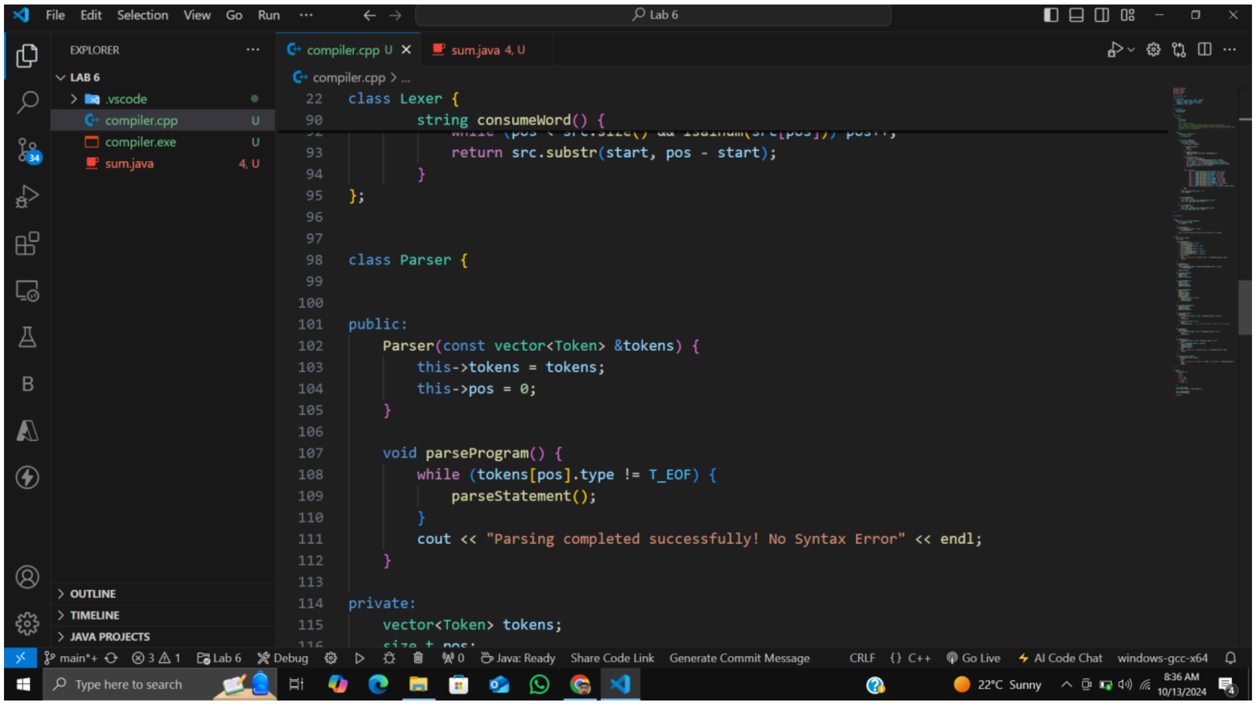
```cpp
#include <string>
#include <cctype>
#include <map>

using namespace std;

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_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE,
    T_SEMICOLON, T_GT, T_EOF,
};


struct Token {
    TokenType type;
    string value;
};

class Lexer {

    private:
        string src;
        size_t pos;
        /*
        It hold positive values.
        In C++, size_t is an unsigned integer data type used to represent the
```



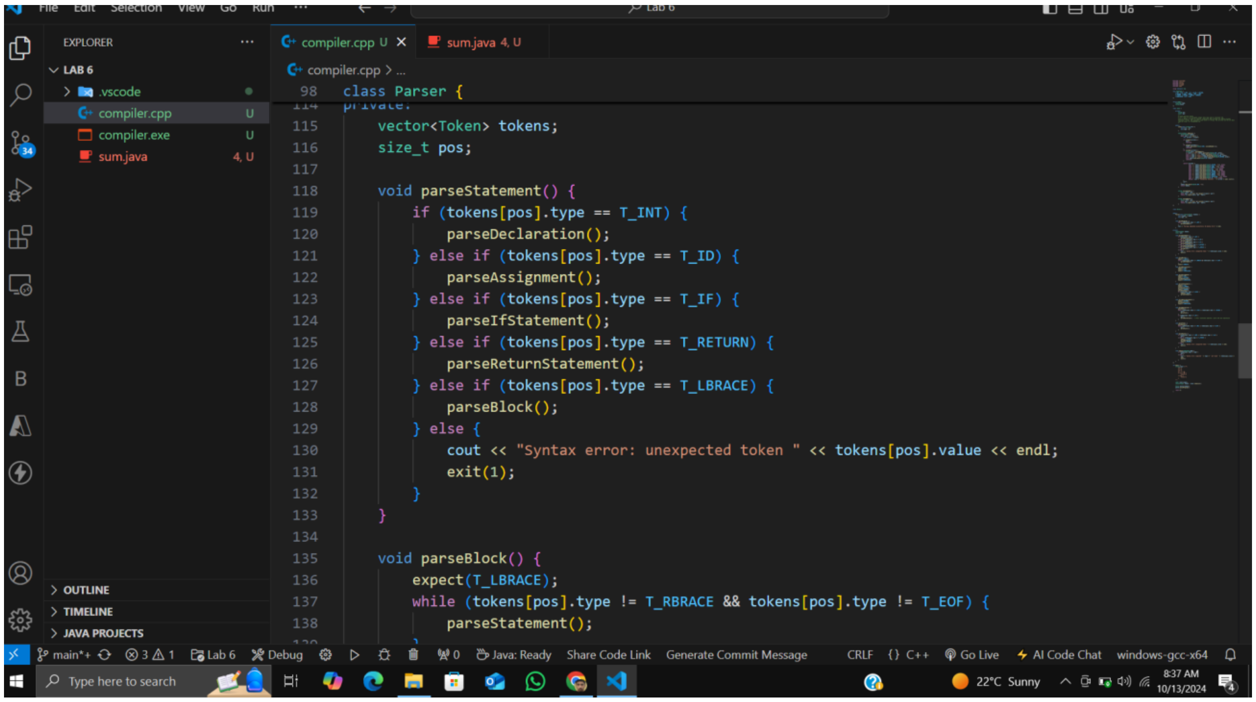
```cpp
class Lexer {
    public:
        Lexer(const string &src) {
            this->src = src;
            this->pos = 0;
        }

        vector<Token> tokenize() {
            vector<Token> tokens;
            while (pos < src.size()) {
                char current = src[pos];

                if (isspace(current)) {
                    pos++;
                    continue;
                }
                if (isdigit(current)) {
                    tokens.push_back(Token{T_NUM, consumeNumber()});
                    continue;
                }
                if (isalpha(current)) {
                    string word = consumeWord();
                    if (word == "int") tokens.push_back(Token{T_INT, word});
                    else if (word == "if") tokens.push_back(Token{T_IF, word});
                    else if (word == "else") tokens.push_back(Token{T_ELSE, word});
                    else if (word == "return") tokens.push_back(Token{T_RETURN, word});
```



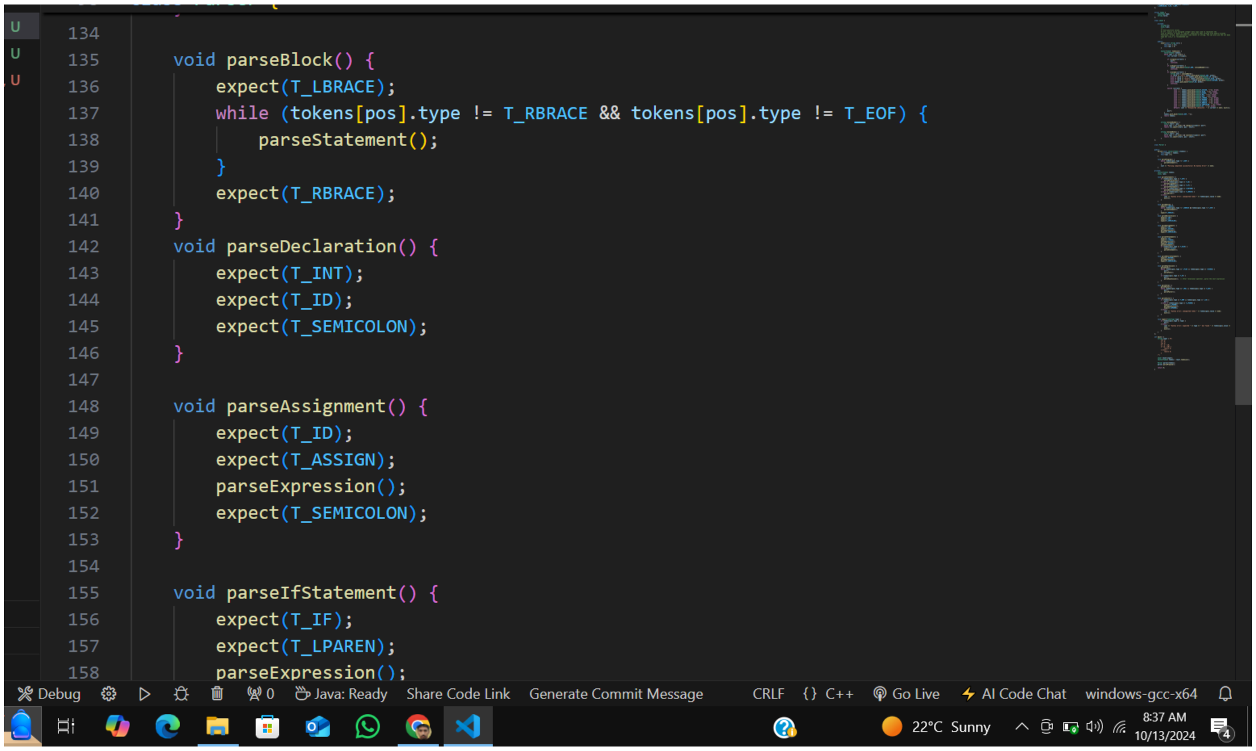
```cpp
class Lexer {
        vector<Token> tokenize() {

                switch (current) {
                        case '=': tokens.push_back(Token{T_ASSIGN, "="}); break;
                        case '+': tokens.push_back(Token{T_PLUS, "+"}); break;
                        case '-': tokens.push_back(Token{T_MINUS, "-"}); break;
                        case '*': tokens.push_back(Token{T_MUL, "*"}); break;
                        case '/': tokens.push_back(Token{T_DIV, "/"}); break;
                        case '(': tokens.push_back(Token{T_LPAREN, "("}); break;
                        case ')': tokens.push_back(Token{T_RPAREN, ")"}); break;
                        case '{': tokens.push_back(Token{T_LBRACE, "{"}); break;
                        case '}': tokens.push_back(Token{T_RBRACE, "}"}); break;
                        case ';': tokens.push_back(Token{T_SEMICOLON, ";"}); break;
                        case '>': tokens.push_back(Token{T_GT, ">"}); break;
                        default: cout << "Unexpected character: " << current << endl; exit(1);
                }
                pos++;
            }
            tokens.push_back(Token{T_EOF, ""});
            return tokens;
        }


        string consumeNumber() {
            size_t start = pos;
```



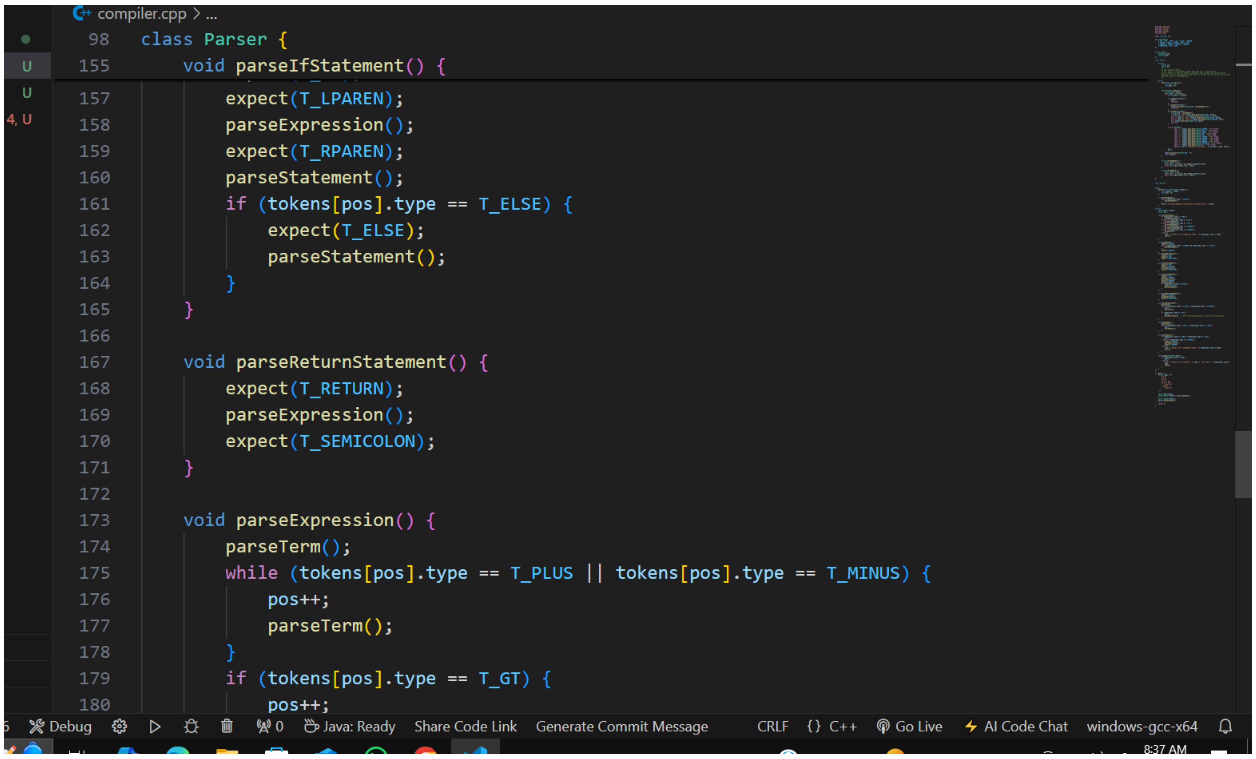
```cpp
class Lexer {
        string consumeWord() {
            while (pos < src.size() && isalnum(src[pos])) pos++;
            return src.substr(start, pos - start);
        }
};


class Parser {


public:
    Parser(const vector<Token> &tokens) {
        this->tokens = tokens;
        this->pos = 0;
    }

    void parseProgram() {
        while (tokens[pos].type != T_EOF) {
            parseStatement();
        }
        cout << "Parsing completed successfully! No Syntax Error" << endl;
    }

private:
    vector<Token> tokens;
    size_t pos;
```



```cpp
class Parser {
private:
    vector<Token> tokens;
    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_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 << endl;
            exit(1);
        }
    }

    void parseBlock() {
        expect(T_LBRACE);
        while (tokens[pos].type != T_RBRACE && tokens[pos].type != T_EOF) {
            parseStatement();
```

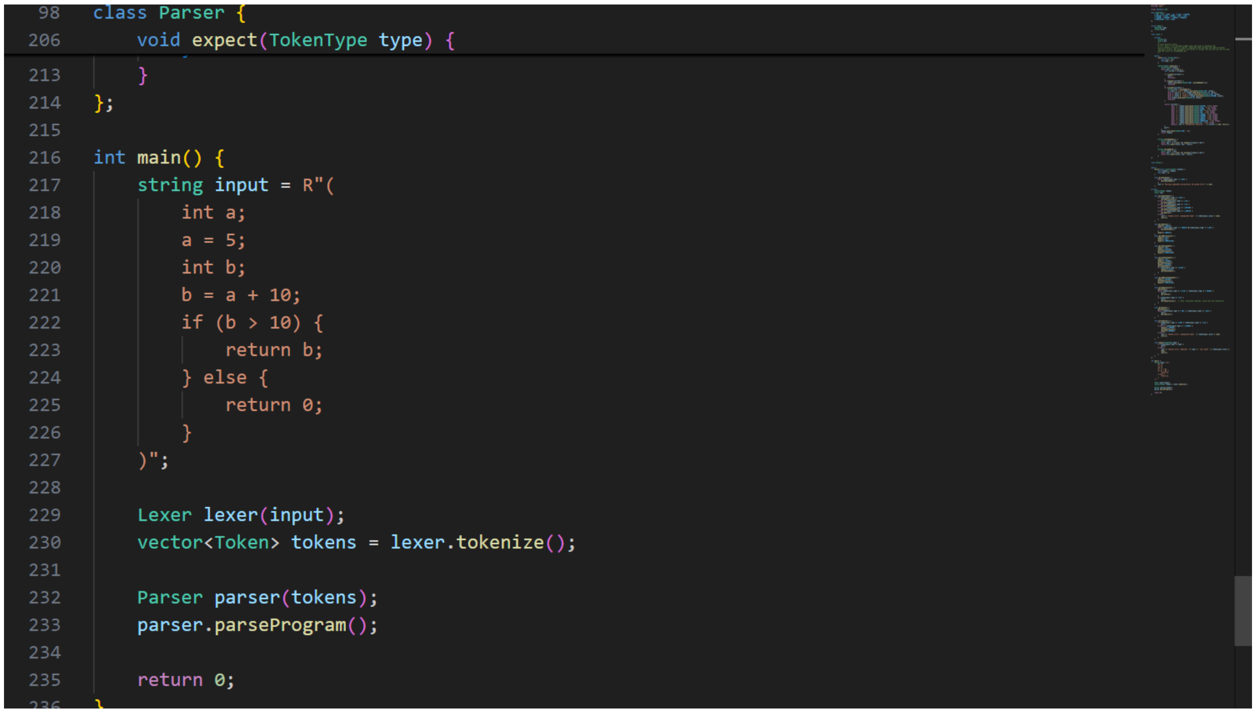


```
    void parseBlock() {
        expect(T_LBRACE);
        while (tokens[pos].type != T_RBRACE && tokens[pos].type != T_EOF) {
            parseStatement();
        }
        expect(T_RBRACE);
    }
    void parseDeclaration() {
        expect(T_INT);
        expect(T_ID);
        expect(T_SEMICOLON);
    }

    void parseAssignment() {
        expect(T_ID);
        expect(T_ASSIGN);
        parseExpression();
        expect(T_SEMICOLON);
    }

    void parseIfStatement() {
        expect(T_IF);
        expect(T_LPAREN);
        parseExpression();
```



compiler.cpp > ...

```
class Parser {
    void parseIfStatement() {
        expect(T_LPAREN);
        parseExpression();
        expect(T_RPAREN);
        parseStatement();
        if (tokens[pos].type == T_ELSE) {
            expect(T_ELSE);
            parseStatement();
        }
    }

    void parseReturnStatement() {
        expect(T_RETURN);
        parseExpression();
        expect(T_SEMICOLON);
    }

    void parseExpression() {
        parseTerm();
        while (tokens[pos].type == T_PLUS || tokens[pos].type == T_MINUS) {
            pos++;
            parseTerm();
        }
        if (tokens[pos].type == T_GT) {
            pos++;
```



```cpp
class Parser {
    void expect(TokenType type) {
    }
};

int main() {
    string input = R"(
        int a;
        a = 5;
        int b;
        b = a + 10;
        if (b > 10) {
            return b;
        } else {
            return 0;
        }
    )";

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

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

    return 0;
}
```