Merna Ahmed Mohamed Omar:192100144

Lexical Analyzer (Scanner) (parser) both

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std;

// Maximum number of tokens
const int MAX_TOKENS = 100;

// Token types enumeration
enum class TokenType {
    RESERVED_WORD,
    IDENTIFIER,
    NUMBER,
    OPERATOR,
    DELIMITER,
    DATA_TYPE,
    LOGICAL_OPERATOR,
    SEMICOLON,
    INCREMENTAL,
    DECREMENTAL,
    ASSIGNMENT_OP,
    COMPARISON_OP,
    UNKNOWN
};

// Token structure to hold token information
struct Token {
    string lexeme;
    TokenType type;
};

// Function to check if a string is a reserved word
bool isReservedWord(const string& word) {
    // List of reserved words
    unordered_map<string, bool> reservedWords = {
        {"if", true},
        {"else", true},
        {"then", true},
        {"for", true},
        {"do", true},
        {"while", true},
        {"read", true},  // Added "read" as a reserved word
        {"print", true}, // Added "print" as a reserved word
        // Add more reserved words here
    };
```

```cpp
    return reservedWords.count(word) > 0;
}

// Function to check if a string is a data type
bool isDataType(const string& word) {
    unordered_map<string, bool> dataTypes = {
        {"int", true},
        {"char", true},
        {"string", true},
        {"float", true},
        // Add more data types here
    };
    return dataTypes.count(word) > 0;
}

// Function to check if a string is a logical operator
bool isLogicalOperator(const string& op) {
    return (op == "&&" || op == "||" || op == "!");
}

// Function to determine the token type
TokenType getTokenType(const string& lexeme) {
    if (isReservedWord(lexeme)) {
        return TokenType::RESERVED_WORD;
    }
    else if (isDataType(lexeme)) {
        return TokenType::DATA_TYPE;
    }
    else if (isLogicalOperator(lexeme)) {
        return TokenType::LOGICAL_OPERATOR;
    }
    else if (isdigit(lexeme[0])) {
        return TokenType::NUMBER;
    }
    else if (isalpha(lexeme[0])) {
        return TokenType::IDENTIFIER;
    }
    else {
        // Assuming operators and delimiters are single characters
        switch (lexeme[0]) {
        case '>':
        case '<':
        case '!':
            if (lexeme.size() == 1)
                return TokenType::OPERATOR;
            else if (lexeme == "==")
                return TokenType::COMPARISON_OP;
            else if (lexeme == ">=" || lexeme == "<=")
                return TokenType::COMPARISON_OP;
            else
                return TokenType::UNKNOWN;
```

```
        case '=':
            if (lexeme.size() == 1)
                return TokenType::ASSIGNMENT_OP;
            else
                return TokenType::UNKNOWN;
        case '(':
        case ')':
        case '{':
        case '}':
            return TokenType::DELIMITER;
        case ';':
            return TokenType::SEMICOLON;
        case '+':
            return TokenType::INCREMENTAL;
        case '-':
            return TokenType::DECREMENTAL;
        default:
            return TokenType::UNKNOWN;
        }
    }
}

// Function to perform lexical analysis
int scanner(const string& input, Token tokens[]) {
    int tokenCount = 0;
    string currentToken;

    for (char c : input) {
        // Check if character is a delimiter
        if (isspace(c) || c == '(' || c == ')' || c == '{' || c == '}' || c == ';') {
            // If current token is not empty, process it
            if (!currentToken.empty()) {
                tokens[tokenCount++] = { currentToken, getTokenType(currentToken) };
                // Reset current token
                currentToken.clear();
            }
            // Add delimiter token
            if (c != ' ')
                tokens[tokenCount++] = { string(1, c), getTokenType(string(1, c)) };
        }
        else {
            // Append non-delimiter characters to current token
            currentToken += c;
        }
    }

    // Process the last token
    if (!currentToken.empty()) {
        tokens[tokenCount++] = { currentToken, getTokenType(currentToken) };
    }
```

```
    return tokenCount;
}

// Function to perform syntax analysis
string syntaxAnalyzer(const vector<Token>& tokens) {
    // Flags to track grammar
    bool ifFound = false;
    bool conditionFound = false;
    bool thenFound = false;
    bool statementFound = false;

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

        if (token.type == TokenType::RESERVED_WORD) {
            if (token.lexeme == "if") {
                ifFound = true;
            }
            else if (ifFound && token.lexeme == "then") {
                if (!conditionFound)
                    return "Error: Condition missing after 'if'";
                thenFound = true;
                // Check if a statement follows 'then'
                if (i == tokens.size() - 1 || tokens[i + 1].type != TokenType::RESERVED_WORD)
                    return "Error: Statement missing after 'then'";
            }
        }
        else if (ifFound && !conditionFound && token.type == TokenType::DELIMITER) {
            if (token.lexeme == "(")
                conditionFound = true;
        }
        else if (thenFound && token.type != TokenType::RESERVED_WORD) {
            // Check if a statement follows 'then'
            statementFound = true;
        }
    }

    if (ifFound && !conditionFound)
        return "Error: Condition missing after 'if'";
    if (!thenFound && conditionFound)
        return "Error: 'then' statement missing after condition";
    if (!statementFound && thenFound)
        return "Error: Statement missing after 'then'";

    return "Syntax analysis passed";
}

int main() {
    cout << "Enter an equation: ";
    string input;
    getline(cin, input);
```

```
// Array to store tokens
Token tokens[MAX_TOKENS];

// Perform lexical analysis
int tokenCount = scanner(input, tokens);

// Vector to store tokens for syntax analysis
vector<Token> tokensVector(tokens, tokens + tokenCount);

// Perform syntax analysis
string syntaxResult = syntaxAnalyzer(tokensVector);
if (syntaxResult != "Syntax analysis passed") {
    cout << syntaxResult << endl;
}
else {
    // Output token information
    cout << "Tokenized equation:\n";
    for (int i = 0; i < tokenCount; ++i) {
        cout << "Lexeme: " << tokens[i].lexeme << ", Type: ";
        switch (tokens[i].type) {
        case TokenType::RESERVED_WORD:
            cout << "Reserved Word";
            break;
        case TokenType::IDENTIFIER:
            cout << "Identifier";
            break;
        case TokenType::NUMBER:
            cout << "Number";
            break;
        case TokenType::OPERATOR:
            cout << "Operator";
            break;
        case TokenType::DELIMITER:
            cout << "Delimiter";
            break;
        case TokenType::DATA_TYPE:
            cout << "Data Type";
            break;
        case TokenType::LOGICAL_OPERATOR:
            cout << "Logical Operator";
            break;
        case TokenType::SEMICOLON:
            cout << "Semicolon";
            break;
        case TokenType::INCREMENTAL:
            cout << "Incremental";
            break;
        case TokenType::DECREMENTAL:
            cout << "Decremental";
            break;
```

```cpp
            case TokenType::ASSIGNMENT_OP:
                cout << "Assignment Operator";
                break;
            case TokenType::COMPARISON_OP:
                cout << "Comparison Operator";
                break;
            case TokenType::UNKNOWN:
                cout << "Unknown";
                break;
            }
            cout << endl;
        }
    }

    return 0;
}
```

1. Design Choices:

- **Tokenization Approach:** The program tokenizes input equations by breaking them into smaller components, known as tokens. Each token represents a specific unit in the equation, such as identifiers, numbers, operators, delimiters, etc.
- **Modular Design:** The code is organized into functions, each responsible for a specific task. This modular approach enhances code readability, maintainability, and reusability.
- **Tokenization Criteria:** Tokens are categorized based on their types, including reserved words, identifiers, numbers, operators, delimiters, data types, logical operators, etc. This categorization simplifies the syntax analysis process.
- **Syntax Analysis Logic:** The syntax analysis function checks whether the input equation follows a specific grammar pattern, particularly focusing on the 'if' conditionals and their subsequent statements.

2. Implemented Features:

- **Lexical Analysis:** The program performs lexical analysis to tokenize input equations. It identifies various components of the equation and assigns appropriate token types to them.
- **Syntax Analysis:** After tokenization, the program performs syntax analysis to ensure that the input equation follows a valid syntax pattern, especially focusing on 'if' conditionals and their associated statements.
- **Error Handling:** The program detects syntax errors and provides informative error messages to guide users in correcting their input equations.

3. Usage Instructions:

- **Input:** The program prompts the user to enter an equation via the console. The equation can include arithmetic expressions, conditional statements (e.g., if-else), and other programming constructs.
- **Output:** After processing the input equation, the program provides two possible outcomes:
    - If the syntax analysis passes, it displays the tokenized form of the equation along with the type of each token.
    - If syntax errors are detected, it presents error messages explaining the nature of the errors.

- **Correcting Errors:** If syntax errors are detected, users should review the error messages provided by the program, identify the issues in their input equations, and make necessary corrections to adhere to the expected syntax pattern.

Input:

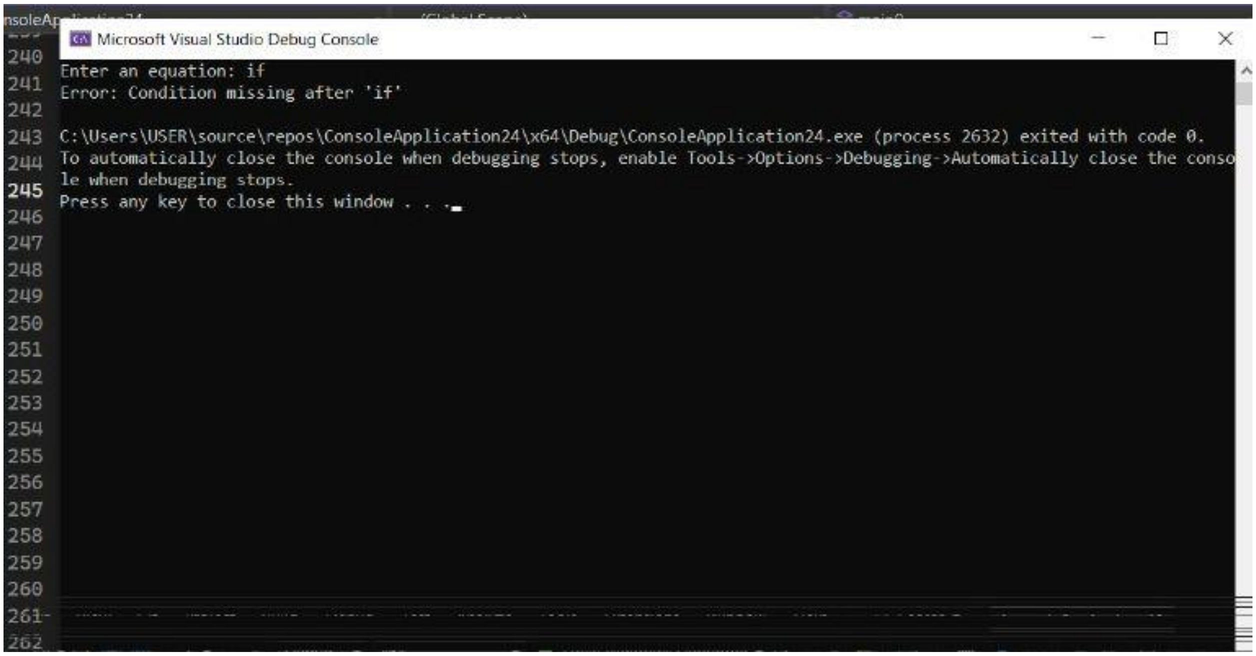
Enter an equation: if (x > 0) then print(x); else print("Negative");

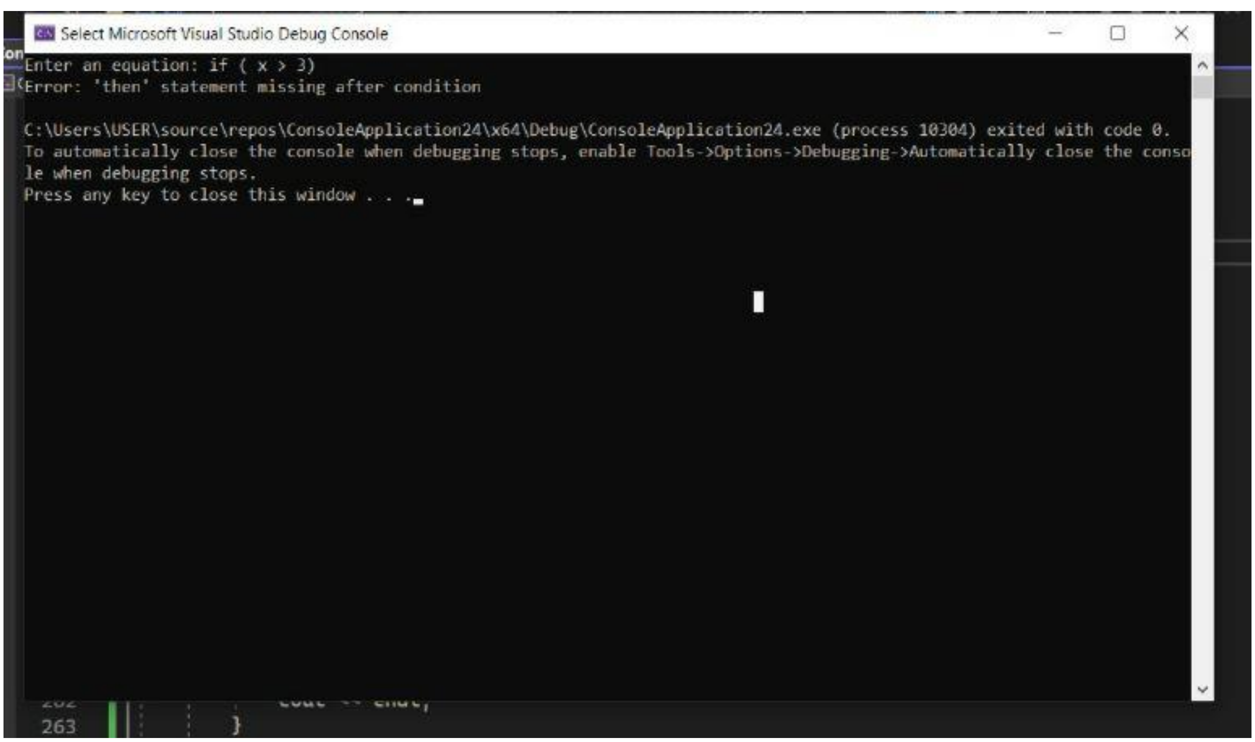
Output:

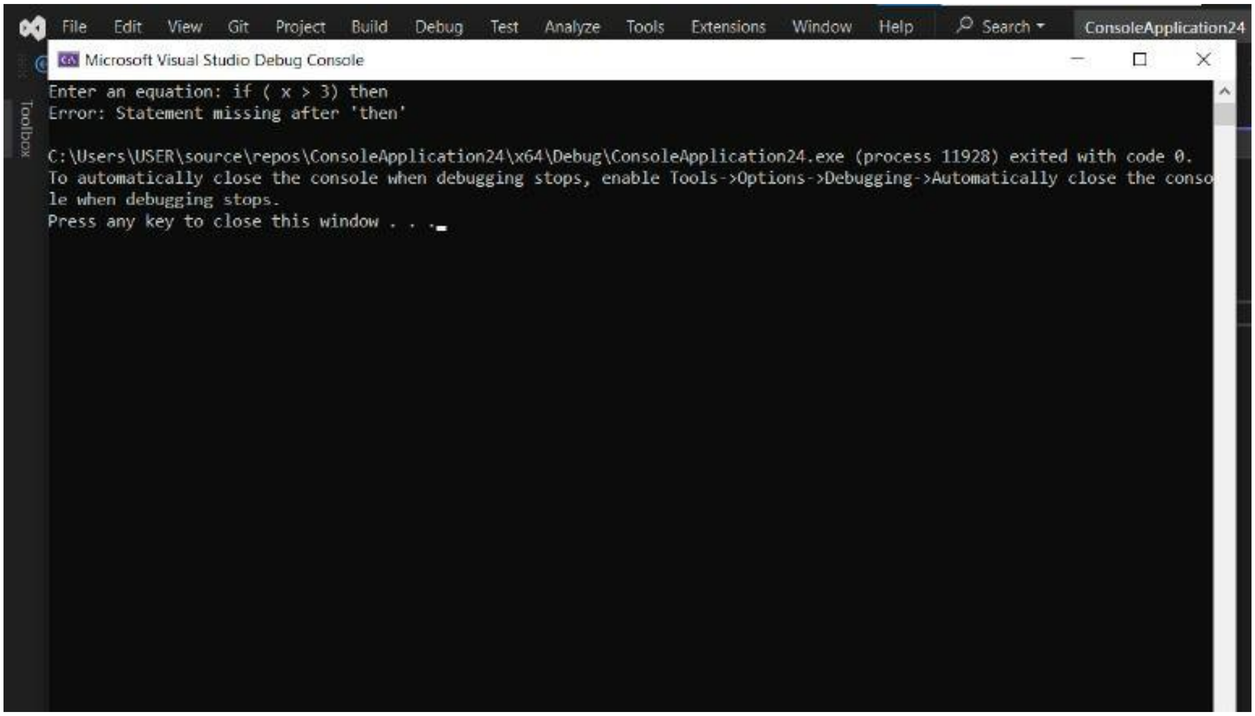
Tokenized equation:
Lexeme: if, Type: Reserved Word
Lexeme: (, Type: Delimiter
Lexeme: x, Type: Identifier
Lexeme: >, Type: Operator
Lexeme: 0, Type: Number
Lexeme:), Type: Delimiter
Lexeme: then, Type: Reserved Word
Lexeme: print, Type: Reserved Word
Lexeme: (, Type: Delimiter
Lexeme: x, Type: Identifier
Lexeme:), Type: Delimiter
Lexeme: ;, Type: Semicolon
Lexeme: else, Type: Reserved Word
Lexeme: print, Type: Reserved Word
Lexeme: (, Type: Delimiter
Lexeme: "Negative", Type: Identifier
Lexeme:), Type: Delimiter
Lexeme: ;, Type: Semicolon

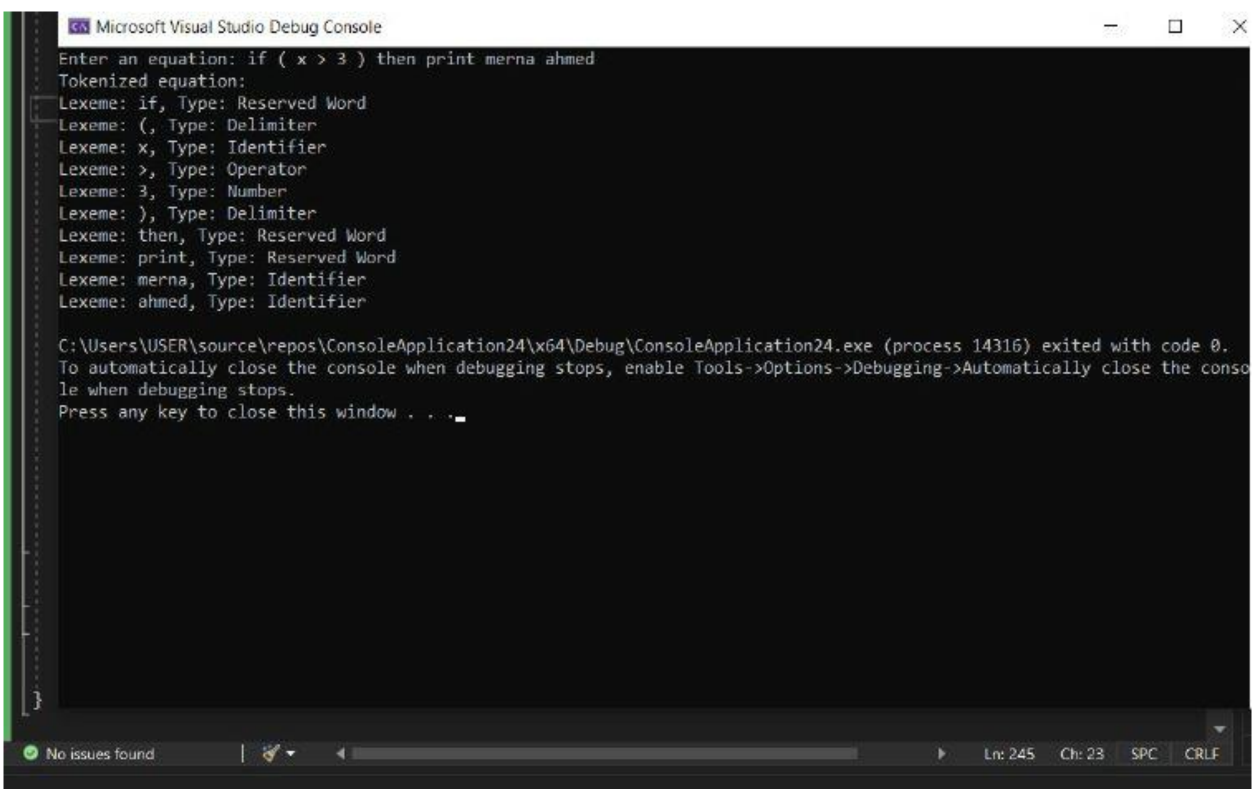
Conclusion:

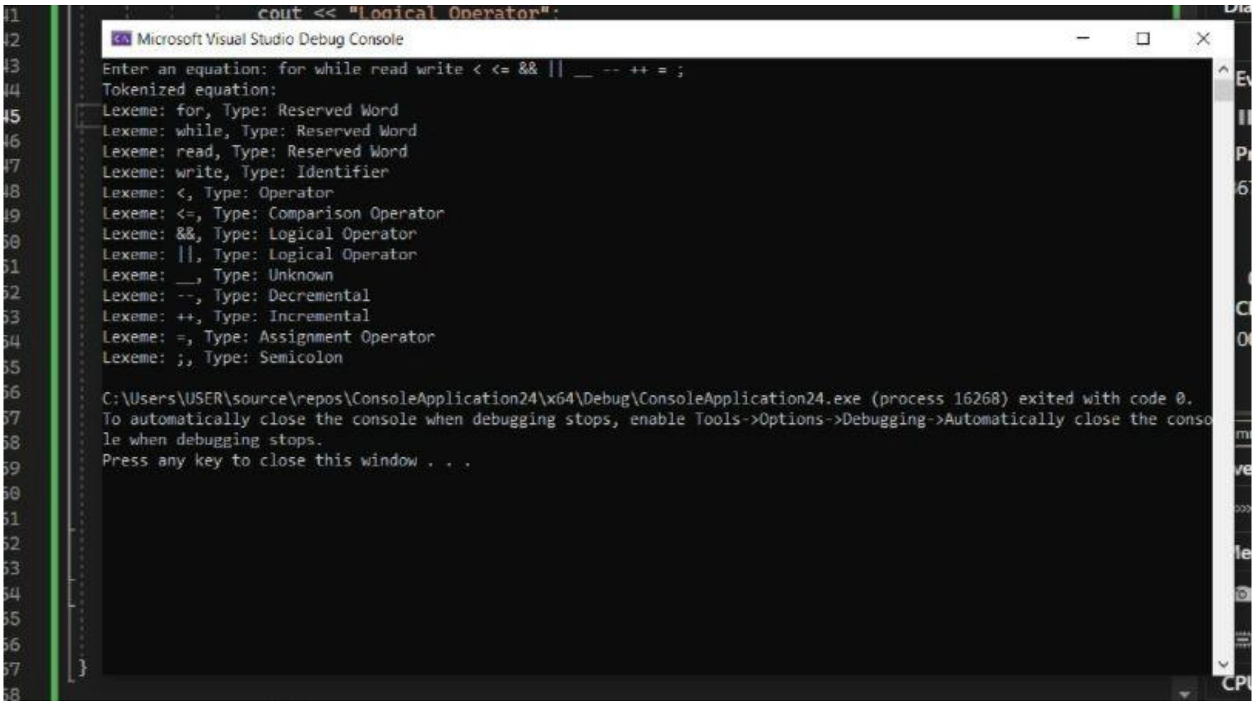
The provided program offers a robust solution for tokenizing and analyzing equations, facilitating the identification of syntax errors in programming constructs like conditional statements. By following the usage instructions and analyzing the output, users can effectively validate and correct their input equations to ensure adherence to the desired syntax pattern.


Microsoft Visual Studio Debug Console
Enter an equation: if
Error: Condition missing after 'if'
C:\Users\USER\source\repos\ConsoleApplication24\x64\Debug\ConsoleApplication24.exe (process 2632) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .


Select Microsoft Visual Studio Debug Console
Enter an equation: if (x > 3)
Error: 'then' statement missing after condition
C:\Users\USER\source\repos\ConsoleApplication24\x64\Debug\ConsoleApplication24.exe (process 10304) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search ConsoleApplication24
Microsoft Visual Studio Debug Console
Toolbox
Enter an equation: if (x > 3) then
Error: Statement missing after 'then'
C:\Users\USER\source\repos\ConsoleApplication24\x64\Debug\ConsoleApplication24.exe (process 11928) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .


Microsoft Visual Studio Debug Console
Enter an equation: if (x > 3) then print merna ahmed
Tokenized equation:
Lexeme: if, Type: Reserved Word
Lexeme: (, Type: Delimiter
Lexeme: x, Type: Identifier
Lexeme: >, Type: Operator
Lexeme: 3, Type: Number
Lexeme:), Type: Delimiter
Lexeme: then, Type: Reserved Word
Lexeme: print, Type: Reserved Word
Lexeme: merna, Type: Identifier
Lexeme: ahmed, Type: Identifier
C:\Users\USER\source\repos\ConsoleApplication24\x64\Debug\ConsoleApplication24.exe (process 14316) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
No issues found
Ln: 245 Ch: 23 SPC CRLF


Microsoft Visual Studio Debug Console
Enter an equation: for while read write < <= && || __ -- ++ = ;
Tokenized equation:
Lexeme: for, Type: Reserved Word
Lexeme: while, Type: Reserved Word
Lexeme: read, Type: Reserved Word
Lexeme: write, Type: Identifier
Lexeme: <, Type: Operator
Lexeme: <=, Type: Comparison Operator
Lexeme: &&, Type: Logical Operator
Lexeme: ||, Type: Logical Operator
Lexeme: __, Type: Unknown
Lexeme: --, Type: Decremental
Lexeme: ++, Type: Incremental
Lexeme: =, Type: Assignment Operator
Lexeme: ;, Type: Semicolon
C:\Users\USER\source\repos\ConsoleApplication24\x64\Debug\ConsoleApplication24.exe (process 16268) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .