CAPESTONE PROJECT PRESENTATION

TOPIC : Developing a Lexical Analyzer for Source Code Tokenization

BY:

K.Gayathri (192221090)

Manoj Kumar(192211695)

WHAT IS A LEXICAL ANALYSER??

- Lexical analysis is the starting phase of the compiler.

- It gathers modified source code that is written in the form of sentences from the language preprocessor.

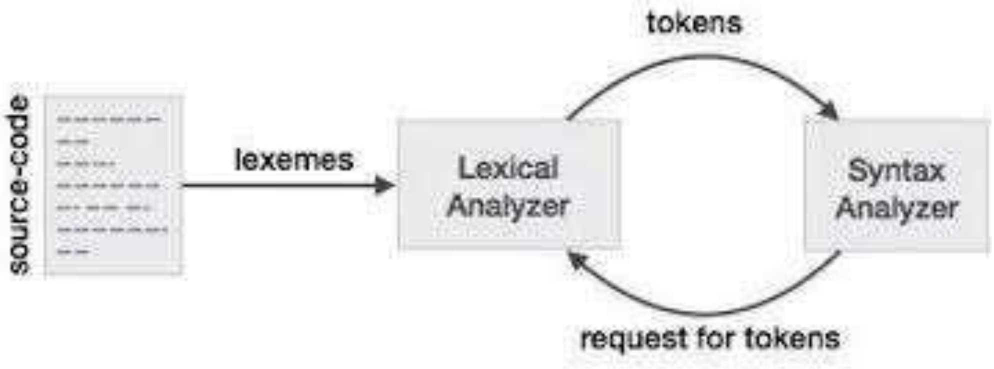
- Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

- The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code.

What is a Token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. **Example of tokens:**

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)



- Syntax analysis, also known as parsing, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language.
- This is typically the second stage of the compilation process, following lexical analysis.

- **Overview:** A lexical analyzer is a fundamental component of a compiler that plays a crucial role in the compilation process.
- Its primary function is to read the source code, break it into individual tokens, and classify them into their respective categories.

- **Key Components:**

1. **Tokenization:** Breaking the source code into individual tokens.
2. **Token Classification:** Classifying each token into keywords, identifiers, operators, literals, and symbols.
3. **Data Structures:** Using data structures such as token tables, symbol tables, and lexeme tables to store and manage tokens.

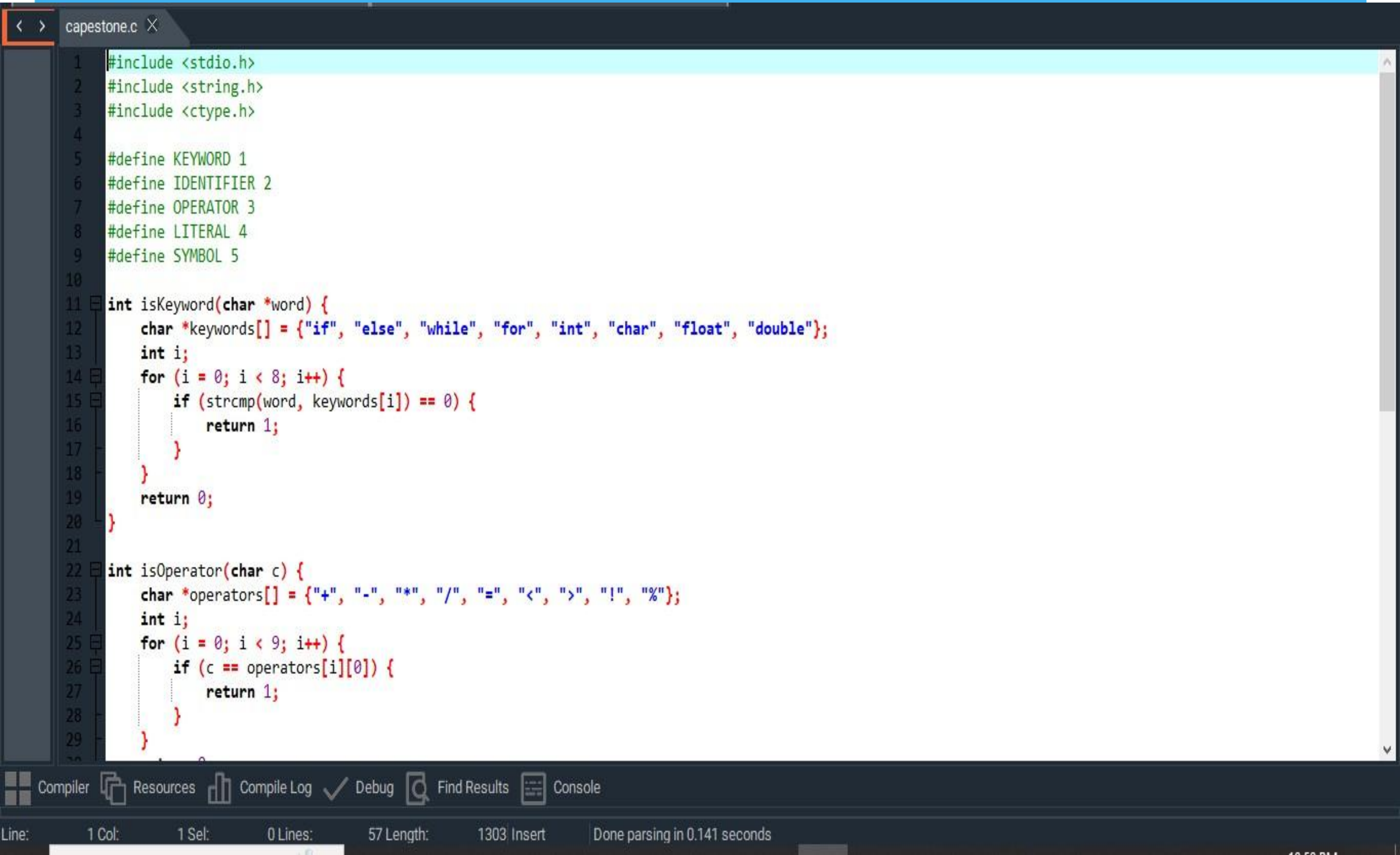
4. **Algorithms:** Implementing algorithms such as pattern matching and state transition diagrams to tokenize and classify tokens.

- **Implementation:**
- 1. Programming Language: C/C++
- 2. Tokenization Process: Reading source code, identifying tokens, classifying tokens, and storing tokens in token tables.
- 3. Parsing Logic: Using regular expressions and finite state machines to tokenize and classify tokens.

LEXICAL ANALYZER

- 1. **Error Handling:** Handling syntax errors and invalid tokens.
- 2. **Token Visualization:** Displaying tokens and their classifications.

- 3. **Code Optimization:** Minimizing tokenization time and memory usage.



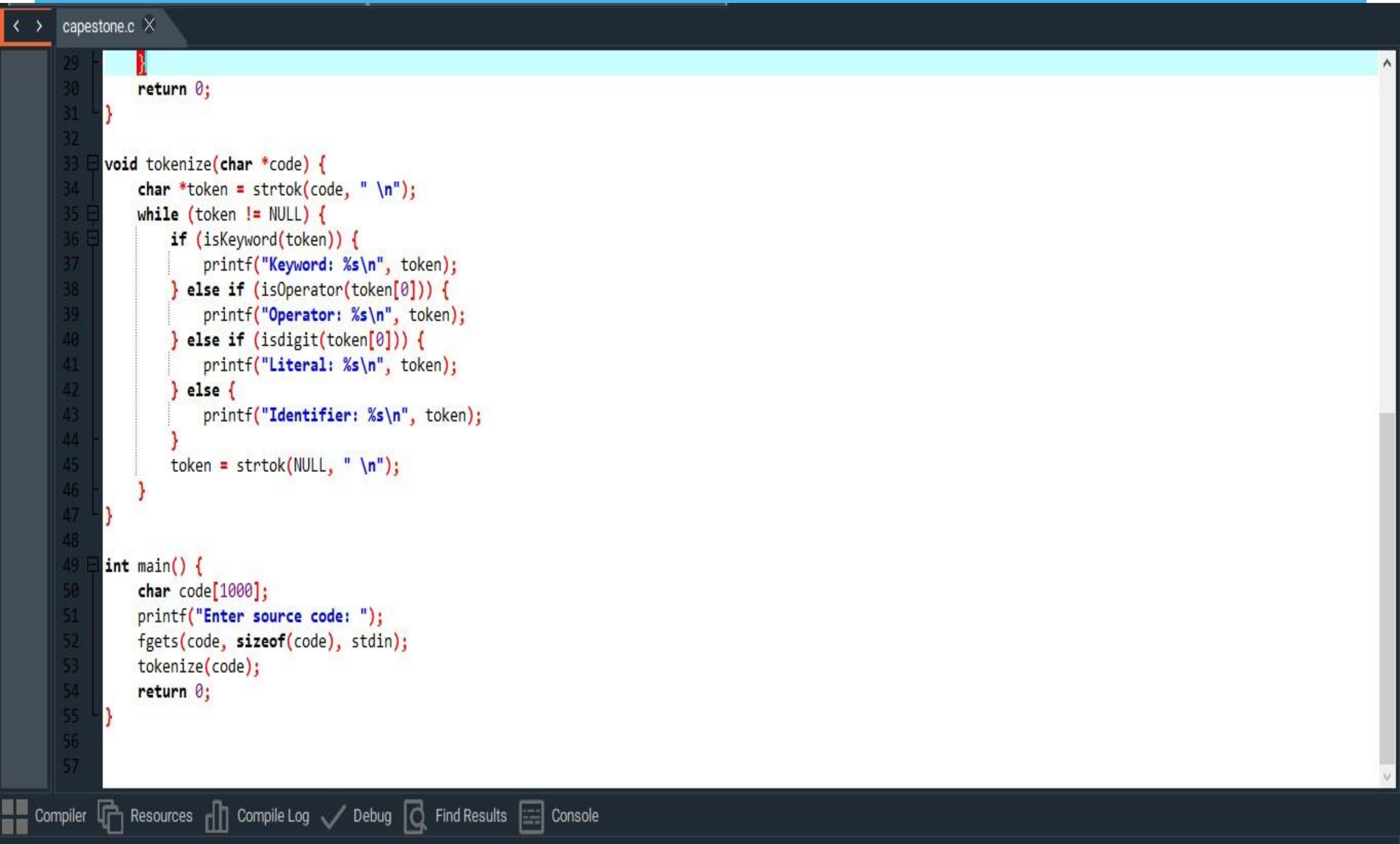
```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define KEYWORD 1
#define IDENTIFIER 2
#define OPERATOR 3
#define LITERAL 4
#define SYMBOL 5

int isKeyword(char *word) {
    char *keywords[] = {"if", "else", "while", "for", "int", "char", "float", "double"};
    int i;
    for (i = 0; i < 8; i++) {
        if (strcmp(word, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isOperator(char c) {
    char *operators[] = {"+", "-", "*", "/", "=", "<", ">", "!", "%"};
    int i;
    for (i = 0; i < 9; i++) {
        if (c == operators[i][0]) {
            return 1;
        }
    }
```

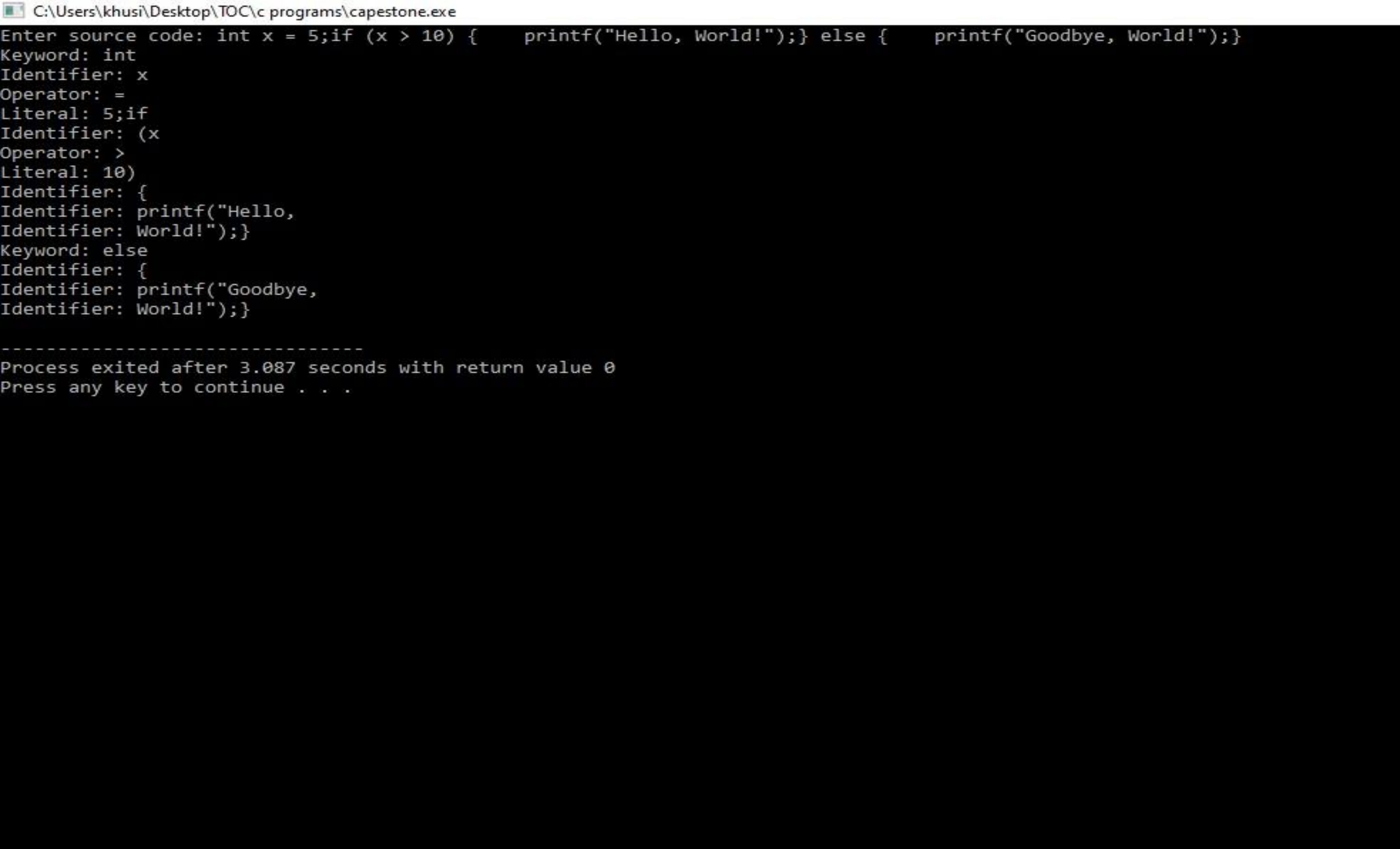
Line: 1 Col: 1 Sel: 0 Lines: 57 Length: 1303 Insert Done parsing in 0.141 seconds



```c
    }
    return 0;
}

void tokenize(char *code) {
    char *token = strtok(code, " \n");
    while (token != NULL) {
        if (isKeyword(token)) {
            printf("Keyword: %s\n", token);
        } else if (isOperator(token[0])) {
            printf("Operator: %s\n", token);
        } else if (isdigit(token[0])) {
            printf("Literal: %s\n", token);
        } else {
            printf("Identifier: %s\n", token);
        }
        token = strtok(NULL, " \n");
    }
}

int main() {
    char code[1000];
    printf("Enter source code: ");
    fgets(code, sizeof(code), stdin);
    tokenize(code);
    return 0;
}
```



```
Enter source code: int x = 5;if (x > 10) {    printf("Hello, World!");} else {    printf("Goodbye, World!");}
Keyword: int
Identifier: x
Operator: =
Literal: 5;if
Identifier: (x
Operator: >
Literal: 10)
Identifier: {
Identifier: printf("Hello,
Identifier: World!");}
Keyword: else
Identifier: {
Identifier: printf("Goodbye,
Identifier: World!");}

--------------------------------
Process exited after 3.087 seconds with return value 0
Press any key to continue . . .
```

CONCLUSION

- Developing a lexical analyzer is a crucial step in building a compiler.
- By understanding the key components, implementation, and features of a lexical analyzer, you can design and implement an efficient and effective tokenization system for source code.