

VISVESVARAYA TECHNOLOGICAL UNIVERSITY  
“JNANA SANGAMA”, BELAGAVI - 590 018



A MINI PROJECT REPORT  
on  
“LOGIC EXTRACTOR SYSTEM”

*Submitted by*

Shreeganesh	4SF20IS092
Vraj K Rabara	4SF20IS116

BACHELOR OF ENGINEERING  
in  
INFORMATION SCIENCE & ENGINEERING

*Under the Guidance of*

Mrs. Madhura N Hegde

Assistant Professor,

Department of ISE,

at



**SAHYADRI**

College of Engineering and Management  
Adyar, Mangaluru - 575 007

2022 - 23

**SAHYADRI**  
**College of Engineering and Management**  
**Adyar, Mangaluru - 575 007**

**Department of Information Science & Engineering**



**CERTIFICATE**

This is to certify that the mini project entitled “**Logic Extractor System**” has been carried out by **Shreeganesh (4SF20IS092)** and **Vraj K Rabara (4SF20IS116)** the bonafide students of Sahyadri College of Engineering and Management, Bachelor of Engineering in Information Science & Engineering of Visvesvaraya Technological University, Belagavi during the year 2022-23. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirements in respect of mini project work prescribed in File Structures Laboratory with Mini Project(18ISL67) for the said degree in sixth semester.

---

**Signature of the Guide1**  
Mrs. Madhura N Hegde

---

**Signature of the Guide2**  
Ms. Jayapadmini Kanchan

---

**Signature of the HOD**  
Dr. Mustafa Bhastikodi

**External Viva:**

Examiner's Name

Signature with Date

1. ....

.....

2. ....

.....

**SAHYADRI**  
**College of Engineering and Management**  
Adyar, Mangaluru - 575 007

Department of Information Science & Engineering



**DECLARATION**

We hereby declare that the entire work embodied in this Mini Project Report titled **“Logic Extractor System”** has been carried out by us at Sahyadri College of Engineering and Management, Mangaluru under the supervision of **Mrs. Madhura N Hegde**, for **Bachelor of Engineering in Information Science & Engineering**. This report has not been submitted to this or any other University for the award of any other degree.

Shreeganesh (4SF19IS092)

Vraj K Rabara (4SF19IS116)

Dept. of ISE, SCEM, Mangaluru

# Abstract

The logic extractor mini-project focuses on developing a tool capable of automatically extracting logical structures and expressions from C code. The project utilizes techniques from program analysis and formal methods to parse the code, identify control flow structures, and extract the underlying logic. The logic extractor tool employs lexical and syntactic analysis to tokenize and parse the C code. The tool identifies conditional statements, loops, and other control flow constructs. It applies data flow analysis techniques to infer variable dependencies and identifies logical expressions involving these variables. The mini-project aims to provide a user-friendly interface where developers can input C code snippets and obtain the extracted logic in a human-readable format. The logic extractor mini-project presents a tool that automates the extraction of logical structures and expressions from C code. The tool assists developers in understanding the underlying logic in their code, enabling effective debugging, optimization, and formal verification processes.

# Acknowledgement

It is with great satisfaction and euphoria that we are submitting the Mini Project Report on “**Logic Extractor System**”. We have completed it as a part of the curriculum of Visvesvaraya Technological University, Belagavi for the award of Bachelor of Engineering in Information Science & Engineering.

We are profoundly indebted to our guide, **Mrs. Madhura N Hegde**, Assistant Professor, Department of Information Science & Engineering for innumerable acts of timely advice, encouragement and We sincerely express our gratitude.

We express our sincere gratitude to **Dr. Mustafa Basthikodi**, Professor and Head, Department of Information Science & Engineering for his invaluable support and guidance.

We sincerely thank **Dr. Rajesha S**, Principal, Sahyadri College of Engineering and Management who have always been a great source of inspiration.

Finally, yet importantly, we express our heartfelt thanks to our family and friends for their wishes and encouragement throughout the work.

**Shreeganesh (4SF19IS092)**

**Vraj K Rabara (4SF19IS116)**

# Table of Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgement</b>	<b>ii</b>
<b>Table of Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	2
1.2 Scope and Purpose . . . . .	2
<b>2 Requirements Specification</b>	<b>3</b>
2.1 Hardware Specification . . . . .	3
2.2 Software Specification . . . . .	3
<b>3 System Design</b>	<b>4</b>
3.1 Architecture Diagram . . . . .	4
3.2 Appliation Modules . . . . .	5
3.2.1 User module . . . . .	5
<b>4 Implementation</b>	<b>6</b>
4.1 Languages Used . . . . .	6
4.1.1 Python . . . . .	6
4.1.2 Storage . . . . .	6
4.2 Pseudocode . . . . .	7
<b>5 Results and Discussion</b>	<b>10</b>
<b>6 Conclusion</b>	<b>13</b>
<b>References</b>	<b>14</b>

# List of Figures

3.1	System Architecture Diagram . . . . .	4
4.1	Pseudocode for reading input and output file . . . . .	7
4.2	Pseudocode for removing comments . . . . .	7
4.3	Pseudocode for removing scanf,printf,specific symbols and semicolon .	8
4.4	Pseudocode for removing comments . . . . .	9
5.1	Creation of input and output file . . . . .	10
5.2	Source code for the logic extraction . . . . .	11
5.3	Logic part extracted from the input file . . . . .	12

# Chapter 1

## Introduction

In modern software development, handling large-scale projects often involves working with complex file structures. Understanding the organization and relationships within these structures is crucial for effective maintenance, refactoring, and analysis of software systems. This mini-project focuses on developing a logic extractor tool that extracts file structure information from a given codebase, enabling developers to gain insights into the hierarchical relationships and dependencies among files. The tool starts by traversing the project directory and scanning all the relevant code files. It then applies various parsing and analysis techniques to identify key elements such as classes, functions, variables, and their relationships within and across files. The logic extractor takes into account factors such as imports, function calls, and variable references to establish connections between different files. The logic extractor for code files also aims to enhance the tool's usability by providing a user-friendly interface. Developers can input the project directory or specific code files to extract the logical structure. In conclusion, this mini-project aims to develop a logic extraction tool for C code that automates the extraction of loops, inputs, outputs, functions, and facilitates library removal and code transfer between files. The tool empowers developers with a deeper understanding of the code's structure, enables optimization and modularization, and enhances code maintenance and refactoring processes.



## 1.1 Overview

The logic extraction and code manipulation mini-project aims to develop a tool that can automatically extract and manipulate specific elements from C code, including loops, inputs, outputs, functions, as well as remove libraries from the code and facilitate the transfer of code snippets between different files while considering read and write command permissions. The project encompasses various techniques from program analysis, code transformation, and file management to achieve these goals. The tool will utilize techniques such as lexical and syntactic analysis to parse the C code and construct an abstract syntax tree .

## 1.2 Scope and Purpose

The purpose of this mini-project is to streamline the process of logic extraction, code manipulation, and file management for C code. By automating these tasks, the tool aims to improve developer productivity, code comprehension, and code organization. It provides developers with a convenient and efficient way to extract logic from loops, analyze inputs and outputs, understand functions, remove unnecessary libraries, and transfer code between files while adhering to proper permissions. The scope of the mini-project on logic extraction encompasses several key aspects, including loops, inputs, outputs, functions, removal of libraries from C code, and transferring code between different files while managing read and write command permissions. The purpose of the mini-project is to develop a tool that automates these tasks, providing developers with enhanced control and understanding of their code.

# Chapter 2

## Requirements Specification

### 2.1 Hardware Specification

- Processor : Intel(R) Core(TM) i5-1005G1 CPU @ 1.20GHz
- RAM : 8GB
- Hard Disk : 500GB
- Input Device : Standard keyboard and Mouse
- Output Device : Monitor

### 2.2 Software Specification

- Programming Language : Python 3.10.8
- IDE : Visual Studio Code 1.79.2

# Chapter 3

## System Design

### 3.1 Architecture Diagram

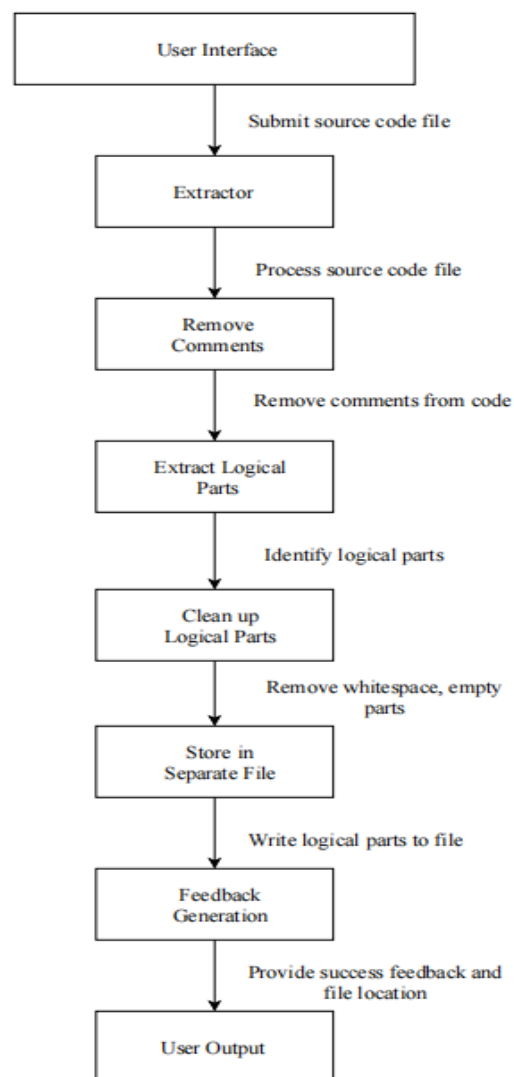


Figure 3.1: System Architecture Diagram

The Logic Extractor system architecture can be organized into several components that work together to extract logic from the C code. It all starts with a command-line interface (CLI) that accepts inputs, displays outputs, and provides options for logic extraction tasks. Then the input file parser component reads the C code from the input file provided by the user. It performs lexical and syntactic analysis to tokenize the code and generate an abstract syntax tree (AST). The logic extraction modules are responsible for extracting different aspects of the logic from the C code. The code transfer module facilitates the transfer of code snippets between files with read and write permissions. It handles reading from the source file, writing to the target file, and ensures the appropriate permissions are maintained during the transfer. The output generator component takes the extracted logic from the logic extraction modules and generates the desired output format.

## 3.2 Appliation Modules

### 3.2.1 User module

The user module in the Logic Extractor system serves as the interface between the user and the application. It includes a Command-Line Interface (CLI) that accepts user inputs, displays outputs, and provides options for logic extraction tasks. The user module interacts with other components, such as the Input File Parser, Logic Extraction, Code Transfer and Output Generator to facilitate the extraction of logic from C code. Overall, the user module enables users to interact with the Logic Extractor system, input tasks, and receive the extracted logic in the desired format through the CLI. It acts as a user-friendly interface, providing a seamless experience for users to perform logic extraction tasks on C code.

# Chapter 4

## Implementation

### 4.1 Languages Used

#### 4.1.1 Python

Python is a widely used high-level programming language known for its simplicity, readability, and versatility. It provides a powerful and easy-to-use syntax that enables developers to write efficient and expressive code. Python supports a wide range of applications, including web development, data analysis, scientific computing, machine learning, and automation. One of the key features of Python is its support for regular expressions, commonly referred to as regex. Python provides a built-in module called "re" that allows you to work with regular expressions seamlessly. Regex patterns can range from simple matching of characters to complex combinations involving quantifiers, character classes, grouping, and more. Python's regex capabilities are highly flexible, enabling you to handle a wide range of text-processing tasks efficiently.

#### 4.1.2 Storage

When transferring logic from an input file to an output file, you typically need to consider the file structure, storage, and read and write permissions. Understand the structure of the input file to determine how to extract the desired logic. We need to make sure we have the necessary read permission for the input file. Without the appropriate read access, we won't be able to extract the logic from the file. We need to verify that we have the required write permission for the output file. This permission allows us to save the extracted logic into the new file. Similar to the read permission, you may need to adjust the file's permissions if necessary.

## 4.2 Pseudocode

### Pseudocode for reading input and output file

The code you provided prompts the user to enter the paths for the input file and the resulting file. Once you have obtained these file paths, you can proceed with extracting the logical part from the input file and writing it to the output file.

```
def extract_logical_part():  
    # Prompt the user for input and output file paths  
    input_file = input("Enter the path to the input file: ")  
    output_file = input("Enter the path for the resulting file: ")
```

Figure 4.1: Pseudocode for reading input and output file

### Pseudocode for removing comments

The regular expression pattern `//.*` matches any text following the double forward slashes (`//`) and replaces it with an empty string, effectively removing the single-line comments. The regular expression pattern `/*.*?/` matches multi-line comments enclosed between `/*` and `*/`. The `.*?` matches any character (except a newline) in a non-greedy fashion. The `re.DOTALL` flag is used to ensure that the `.` in the regular expression also matches newline characters, allowing it to remove comments spanning multiple lines. Finally, the function returns the modified content string after stripping any leading or trailing whitespace.

```
def remove_comments(content):  
    # Remove single-line comments  
    content = re.sub(r'//.*', '', content)  
  
    # Remove multi-line comments  
    content = re.sub(r'/*.*?\*/', '', content, flags=re.DOTALL)  
  
    return content.strip()
```

Figure 4.2: Pseudocode for removing comments

### Pseudocode for removing scanf,printf,specific symbols and semicolon

The 'processline()' function takes a line of code as input and performs various transformations and checks on it. The line is stripped of leading and trailing whitespace using the 'strip()' method. If the line matches certain keywords ('int', 'double', 'string', 'float', or 'return') and does not contain the assignment operator '=', the line is set to 'None'. If the line is not 'None', it checks for the presence of 'scanf' or 'printf' statements. If 'scanf' is found, the line is modified to indicate it is an input. If 'printf' is found, the line is modified to indicate it is an output. If the line is equal to "return 0;" or "exit(0);", it is set to 'None'. If the line is not 'None' and ends with a semicolon (;), the semicolon is removed. The modified line is returned as the output of the function.

```
def process_line(line):
    line = line.strip()
    if line is None or line == "":
        return None
    if re.match(r'\b(int|double|string|float|return)\b', line) and '=' not in line:
        line = None
    if line is not None:
        # Check if the line contains cin or cout
        if "scanf" in line :
            line = "INPUT----->" + line.replace('\n', '\t').replace('\r', '\t')
            # Remove parentheses from scanf statement
            line = re.sub(r'scanf\s*\((.*?)\)', r'scanf \1', line)
            # Remove the first comma in scanf statement
            line = re.sub(r'scanf\s*([^\,]*)\s*', r'scanf \1', line)
            # Replace other specific symbols but keep << and >>
            line = re.sub(r'(scanf|"%d|%f|&|;)', '', line)
        elif "printf" in line :
            line = "OUTPUT----->" + line.replace('\n', '\t')
            # Remove parentheses from printf statement
            line = re.sub(r'printf\s*\((.*?)\)', r'printf \1', line)
            # Replace other specific symbols
            line = re.sub(r'(printf|"%s|&|;)', '', line)
            # Match format specifiers with variables
            format_specifiers = re.findall(r'%[a-zA-Z]', line)
            variables = re.findall(r'(?<=",)\s*([^\,]+)', line)
            for i in range(min(len(format_specifiers), len(variables))):
                line = line.replace(format_specifiers[i], variables[i])
            # Replace other specific symbols
            line = re.sub(r'(printf|"%[a-zA-Z]|&|;)', '', line)
        elif line == "return 0;" or line == "exit(0);":
            line = None
        # Remove semicolon at the end of the line
        if line is not None and line.endswith(';'):
            line = line[:-1]
    return line
```

Figure 4.3: Pseudocode for removing scanf,printf,specific symbols and semicolon

## Pseudocode for reading input and output file

The code snippet provided reads the contents of the input file, performs several transformations and checks on the content, extracts the logical part, and writes it to the output file. The input file is opened in read mode using `open(input-file, 'r')`. The regular expression `re.sub()` function is used to remove header files. The `remove-comments()` function is called to remove comments from the content. The content is split into lines using `strip()` and `split()`. Each line is processed using the `process-line()` function. The starting and ending brackets are removed if present. Indicators for the start and end of the logical part are added to the logical-part list. If the logical-part list is not empty, the output file is either created or rewritten to store the logical part. Finally, a message is printed to indicate whether the logical part was found and stored in the output file.

```
with open(input_file, 'r') as file:
    content = file.read()
    # Remove header files using regular expression
    content = re.sub(r'#include\s*<.*?>', '', content)
    # Remove comments from the content
    content = remove_comments(content)
    logical_part = []
    # Split the content into lines
    lines = content.strip().split('\n')
    # Process each line
    for line in lines:
        line = process_line(line)
        if line is not None:
            logical_part.append(line)
    if logical_part:
        # Remove the starting and ending brackets
        if logical_part[0] == "{":
            logical_part.pop(0)
        if logical_part[-1] == "}":
            logical_part.pop()
        logical_part.insert(0, "-----Logical Part Starts-----")
        logical_part.append("-----Logical Part Ends-----")
    if logical_part:
        # Check if the output file already exists
        if os.path.isfile(output_file):
            # Rewrite the output file by opening it in write mode and truncating the contents
            with open(output_file, 'w') as outfile:
                outfile.write('\n'.join(logical_part))
        else:
            # Create a new output file
            with open(output_file, 'w') as outfile:
                outfile.write('\n'.join(logical_part))
        print('Logical part extracted and stored in:', output_file)
    else:
        print('No logical part found in the input file.')
extract_logical_part()
```

Figure 4.4: Pseudocode for removing comments

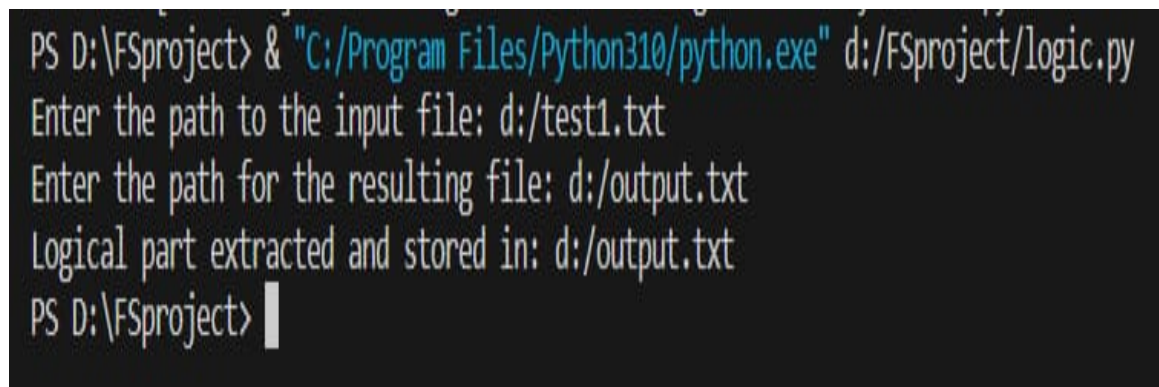


# Chapter 5

## Results and Discussion

### Creation of input and output file

The user is prompted to enter the path to the input file ('d:/test1.txt') and the path for the resulting file ('d:/output.txt'). The logical part of the input file is then extracted and stored in the specified output file ('d:/output.txt').



```
PS D:\FSproject> & "C:/Program Files/Python310/python.exe" d:/FSproject/logic.py
Enter the path to the input file: d:/test1.txt
Enter the path for the resulting file: d:/output.txt
Logical part extracted and stored in: d:/output.txt
PS D:\FSproject> █
```

Figure 5.1: Creation of input and output file

## Insertion of Source code into Input file

Reading an input file (test1.txt) containing C code and extract the logical part from it. It iterates through the file line by line, identifying the logical section based on specific conditions or patterns (e.g., if statements, loops). The logical part is stored in a variable and then written to an output file (output.txt). The code provides flexibility to adapt the conditions or patterns based on the requirements of the logical part extraction.

```
#include<stdio.h> // Include the standard input/output library
#include<stdio.h> // Include the standard input/output library
int main() // Main function
{
    int a, b, c; // Declare three integer variables

    printf("Enter the three sides of triangle"); // Print a message asking for input
    scanf("%d%d%d", &a, &b, &c); // Read three integers from the user

    if ((a < 1 || a > 10) || (b < 1 || b > 10) || (c < 1 || c > 10)) // Check if any side is out of the valid range
    {
        printf("Out of range values"); // Print an error message
        exit(0); // Terminate the program
    }

    if ((a < b + c) && (b < a + c) && (c < a + b)) // Check if the sides form a valid triangle
    {
        if ((a == b) && (b == c)) // Check if all sides are equal
            printf("Equilateral triangle"); // Print that it's an equilateral triangle
        else if ((a != b) && (b != c) && (a != c)) // Check if all sides are different
            printf("Scalene triangle"); // Print that it's a scalene triangle
        else
            printf("Isosceles triangle"); // Print that it's an isosceles triangle
    }
    else
    {
        printf("Triangle cannot be formed"); // Print that a triangle cannot be formed
    }

    return 0; // Return 0 to indicate successful execution
}
```

Figure 5.2: Source code for the logic extraction

### Logic Extracted into Output file

The code snippet reads an input file (test1.txt) containing C code and extracts the logical part based on specific conditions or patterns. It then writes the extracted logical part to an output file named output.txt.

```
-----Logical Part Starts-----  
OUTPUT-----> "Enter the three sides of triangle"  
INPUT-----> a, b, c  
if ((a < 1 || a > 10) || (b < 1 || b > 10) || (c < 1 || c > 10))  
{  
OUTPUT-----> "Out of range values"  
}  
if ((a < b + c) && (b < a + c) && (c < a + b))  
{  
if ((a == b) && (b == c))  
OUTPUT-----> "Equilateral triangle"  
else if ((a != b) && (b != c) && (a != c))  
OUTPUT-----> "Scalene triangle"  
else  
OUTPUT-----> "Isosceles triangle"  
}  
else  
{  
OUTPUT-----> "Triangle cannot be formed"  
}  
-----Logical Part Ends-----|
```

Figure 5.3: Logic part extracted from the input file

# Chapter 6

## Conclusion

The Logic Extractor System for C Code is a valuable tool that automates the extraction of logic from C code, providing developers with a deeper understanding of their codebase. By extracting and analyzing loops, inputs, outputs, functions, and allowing for the removal of libraries, the system enhances code comprehension and aids in code optimization and maintenance. The capability to transfer code snippets between files while managing read and write command permissions improves code organization and facilitates collaboration among developers. With the Logic Extractor System, developers can efficiently extract logic, optimize code, and improve software development processes. The system facilitates the transfer of code snippets between different files while managing read and write command permissions. Developers can specify the source file and the target file where they want to transfer the code. The system handles the necessary file operations, such as reading from the source file and writing to the target file, while ensuring proper permissions are maintained.

# References

- [1] Michael J. Folk, Bill Zoellick, Greg Riccardi: File Structures-An Object Oriented Approach with C++, 3rd Edition, Pearson Education, 1998.
- [2] K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File Structures Using C++, Tata McGraw-Hill, 2008.
- [3] Scot Robert Ladd: C++ Components and Algorithms, BPB Publications, 1993.
- [4] Raghu Ramakrishnan and Johannes Gehrke: Database Management Systems, 3rd Edition, McGraw Hill, 2003.