

COMPLEX COMPILER (LEXICAL)

Submitted by

**SHAIK IRFAN [RA2011026010080]
HARSHITH B [RA2011026010079]**

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

**BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING**

with specialization in Artificial Intelligence & Machine Learning



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

**SCHOOL OF COMPUTING
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603203**

May 2023



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University - U/s 3 of UGC Act, 1956

**SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203**

BONAFIDE CERTIFICATE

Certified that 18CSC304J – COMPILER DESIGN project report titled “COMPLEX COMPILER -LEXIAL” is the bonafide work of SHAIK IRFAN [RA2011026010080] and ARSHITH B [RA2011026010079] who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion in this or any other candidate.

SIGNATURE

Faculty In-Charge
Dr. J. Jeyasudha
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

SIGNATURE

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and HOD,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai



COMPLEX COMPILER (LEXICAL)

Submitted by

SHAIK IRFAN [RA2011026010080]

HARSHITH B [RA2011026010079]

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY in COMPUTER SCIENCE ENGINEERING

with specialization in Artificial Intelligence & Machine Learning



SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

May 2023



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled “**COMPLEX COMPILER -LEXIAL**” is the bonafide work of **SHAIK IRFAN [RA2011026010080]** and **HARSHITH B [RA2011026010079]** who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Faculty In-Charge

Dr. J. Jeyasudha

Assistant Professor

Department of Computational Intelligence

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

SIGNATURE

HEAD OF THE DEPARTMENT

Dr. R Annie Uthra

Professor and HOD,

Department of Computational Intelligence,

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

TABLE OF CONTENTS

Chapter No.	Title	Page No.
1	CHAPTER 1 - INTRODUCTION	4
1.1	Introduction	4
1.2	Problem statement	5
1.3	Objectives	6
1.4	Hardware & Software Specifications	7
2	CHAPTER 2 - ANATOMY OF A COMPILER	8
2.1	Lexical Analysis	8
2.2	Intermediate Code Generation	9
2.3	Three address code	10
3	CHAPTER 3 – ARCHITECTURE AND COMPONENTS	11
3.1	Architecture Diagram	11
3.2	Component Diagrams	12
3.2.1	Lexical Analysis	12
3.2.2	Intermediate Code Generation	14
3.2.3	Quadruple	15
3.2.4	Triple	16
4	CHAPTER 4 – CODING AND TESTING	17
4.1	Lexical Analysis	17
4.2	Intermediate Code Generation	20
5	CHAPTER 5 - RESULT	25
6	CHAPTER 6 - CONCLUSIONS	26

CHAPTER 1

1.1 INTRODUCTION

The development of a website that allows developers to input their code and see the output of what the compiler would produce is an excellent tool for developers. As software development becomes increasingly complex, the ability to test code and identify potential errors before deploying it is critical. This tool provides developers with a convenient way to quickly and efficiently test their code without the need to install and configure a complete development environment.

Consider a scenario where a developer is working on a new feature for an application. The feature requires the use of a complex algorithm that the developer has not worked with before. The developer writes the code and attempts to run it, but immediately receives an error message from the compiler. Without the ability to test the code in isolation, the developer must spend significant time trying to isolate the error and determine how to fix it.

However, with the use of the tool we have developed, the developer can simply input the code into the website and select the appropriate part of the compiler to test it. In this case, the developer could select the lexical analyzer to identify and correct any syntax errors. The tool quickly identifies the error and provides a clear message to the developer on how to fix it. With the error corrected, the developer can then run the code again and see the output of what the compiler would produce. This tool saves the developer a significant amount of time and effort in the debugging process.

Additionally, this tool is also useful for teaching programming. Beginners can use the tool to learn the basics of programming without the need to install and configure a development environment. The tool provides an easy-to-use interface that allows users to write code, see the output of what the compiler would produce, and make changes as needed. The feedback provided by the tool is immediate and clear, which helps beginners quickly identify and correct errors.

1.2 PROBLEM STATEMENT

As a software developer, we might have encountered situations where you want to test your code against different compilers, or we might have to compile your code on different platforms. But it can be time-consuming and challenging to set up different compilers and platforms to compile your code manually. This is where the tool you have developed comes in handy. The tool allows developers to input their code and see the output of what the compiler would produce without worrying about installing and configuring different compilers and platforms. With your tool, developers can quickly test their code against different compilers and platforms without leaving their development environment. This tool can also be beneficial for developers who are just starting with programming, as they can see how their code is being compiled and understand the different stages of the compilation process, such as lexical analysis and intermediate code generation. Moreover, the tool can help developers to identify and fix errors in their code during the development phase, making it easier for them to deliver bug-free code. Hence, the tool can save developers a lot of time and effort by providing them with a convenient and efficient way to test their code against different compilers and platforms, and help them deliver high-quality code.

1.3 OBJECTIVES

- Developed a website that allows developers to input their code and see the output of what the compiler would produce.
- Implemented a lexical analyzer, intermediate code generation (Three address code).
- Used HTML,CSS as frontend and NodeJS with as backend, python as source code.
- Used FLASK to connect source code with the frontend.

1.4 HARDWARE AND SOFTWARE SPECIFICATIONS

1.4.1 HARDWARE REQUIREMENTS

- A server or cloud infrastructure to host the website and the backend logic.
- Sufficient RAM and CPU power to handle multiple user requests simultaneously.
- Sufficient disk space to store the code files and other resources.

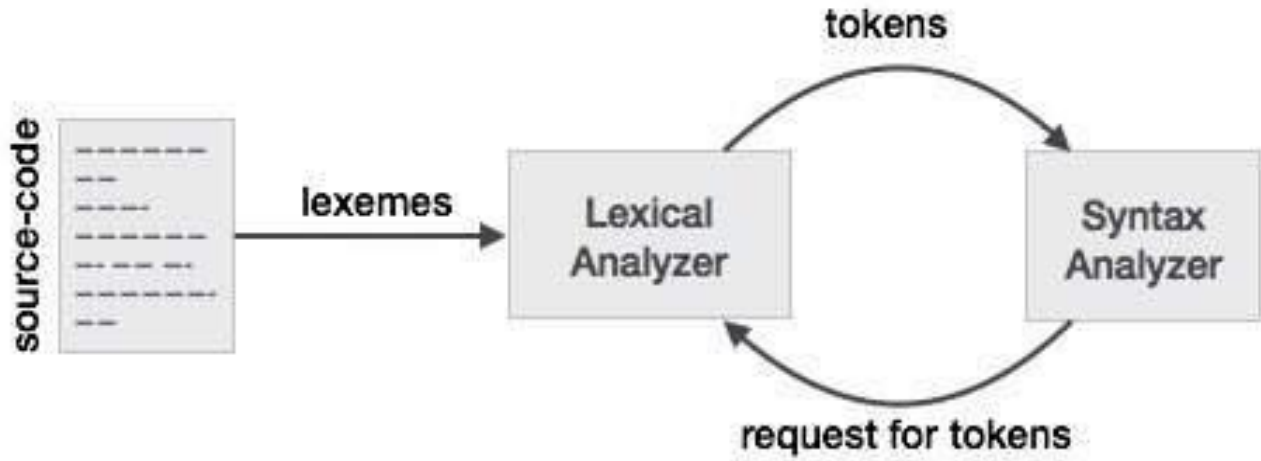
1.4.2 SOFTWARE REQUIREMENTS

- Operating system: Linux or compatible OS.
- Python interpreter installed on the server.
- NodeJS , FLASK installed for the backend.
- A web browser to access the website.

CHAPTER 2

ANATOMY OF A COMPILER

2.1 LEXICAL ANALYSIS



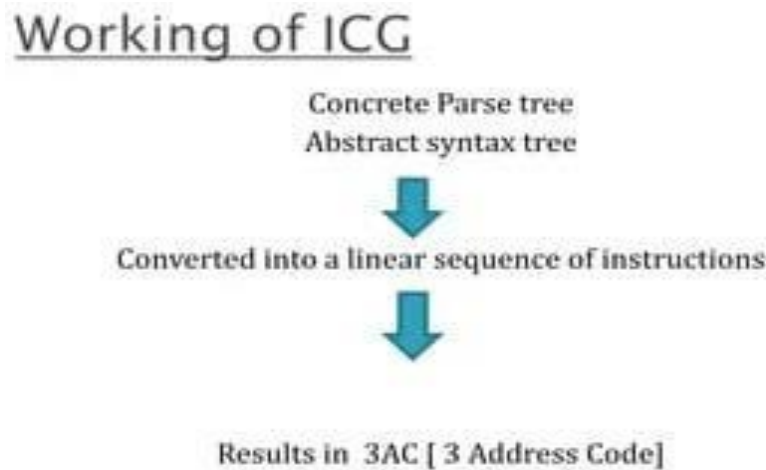
Lexical analysis is the first phase of the compilation process in which the input source code is scanned and analysed to generate a stream of tokens that are subsequently used by the compiler in the later stages of the compilation process.

The process of lexical analysis is also known as scanning, and the component of the compiler that performs this task is called a lexer or a tokenizer. The primary goal of lexical analysis is to identify the individual lexical units (tokens) of the source code, such as keywords, identifiers, literals, operators, and punctuation marks, and produce a stream of tokens that can be processed by the compiler's parser.

The process of lexical analysis involves several steps. The first step is to read the source code character by character and group them into lexemes, which are the smallest meaningful units of the programming language. Next, the lexer applies a set of rules or regular expressions to identify the lexemes and classify them into different token types.

During this process, the lexer also discards any comments or white spaces that are not significant to the language's syntax. For example, the lexer would ignore any spaces, tabs, or newlines in the source code and focus only on the meaningful tokens that make up the language's syntax.

2.2 INTERMEDIATE CODE GENERATION



Intermediate code generation is a crucial phase in the process of compiling a programming language. Its purpose is to translate the source code into an intermediate language representation that is closer to machine code and can be easily optimized and translated into executable code.

During intermediate code generation, the compiler analyses the source code and creates a simplified version of it, typically in the form of a set of instructions or statements in a lower-level language. This intermediate code serves as a bridge between the high-level source code and the low-level machine code.

The intermediate code is usually designed to be independent of the hardware and operating system on which the code will eventually run, making it easier to port the code to different platforms. It also allows the compiler to perform optimizations that can improve the performance of the generated code, such as dead code elimination and constant folding.

Some examples of intermediate code representations are Three-Address Code (TAC), Quadruple Code, and Intermediate Representation (IR). These representations are usually simpler and more concise than the original source code, making it easier for the compiler to analyse and optimize them. Once the intermediate code is generated, the compiler can then proceed to the next phase, which is code optimization and code generation.

2.3 THREE ADDRESS CODE

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

Three address code is used in compiler applications:

Optimization: Three address code is often used as an intermediate representation of code during optimization phases of the compilation process. The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation: Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process. The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.

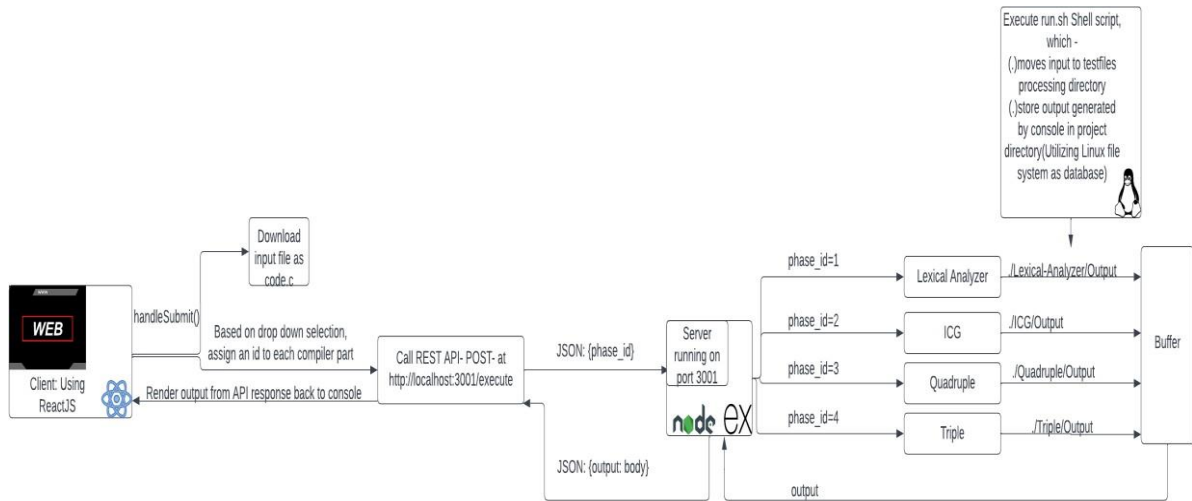
Debugging: Three address code can be helpful in debugging the code generated by the compiler. Since three address code is a low-level language, it is often easier to read and understand than the final generated code. Developers can use the three address code to trace the execution of the program and identify errors or issues that may be present.

Language translation: Three address code can also be used to translate code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to translate the code to multiple target languages.

CHAPTER 3

ARCHITECTURE AND COMPONENTS

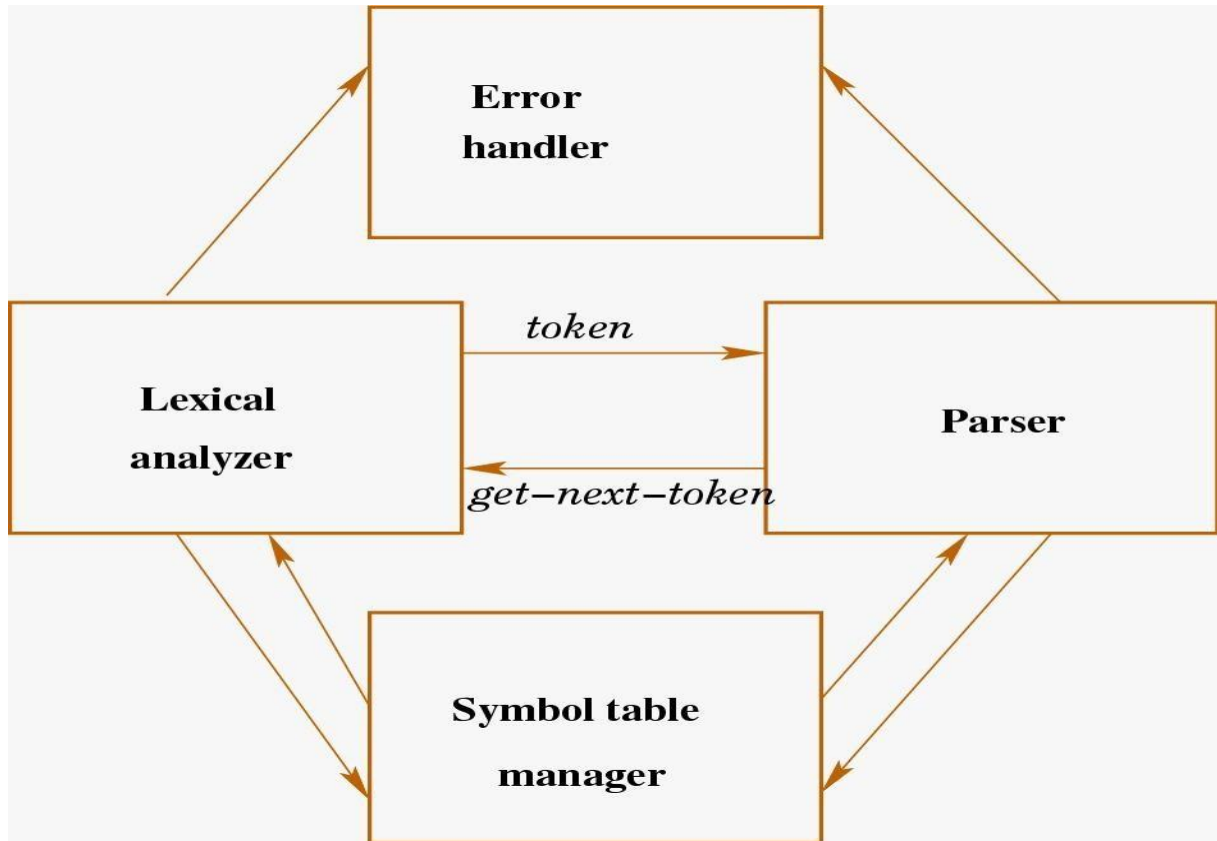
3.1 ARCHITECTURE DIAGRAM



The project consists of several components that work together to achieve its goal. The Front-end UI is responsible for presenting the user interface to the end-user, and it is built using NodeJS, which is a popular JavaScript library for building user interfaces. The FLASK serves as the communication interface between the front-end UI and the back-end logic. On the other hand, the back-end logic consists of several components, including the Lexical Analyzer and the Intermediate Code Generator (ICG). The Lexical Analyzer is responsible for analysing the input source code and generating a stream of tokens, while the ICG is responsible for generating an intermediate code representation of the input source code. Three Address code data structures are used to represent the intermediate code generated by the ICG, and they are also implemented as a part of the back-end logic. Overall, the project architecture involves the front-end UI, the FLASK, and the back-end logic components, including the Lexical Analyzer, ICG, and data structures. Each component has a specific role and works together to achieve the project's objective.

3.2 COMPONENTS DIAGRAMS

3.2.1 LEXICAL ANALYSIS

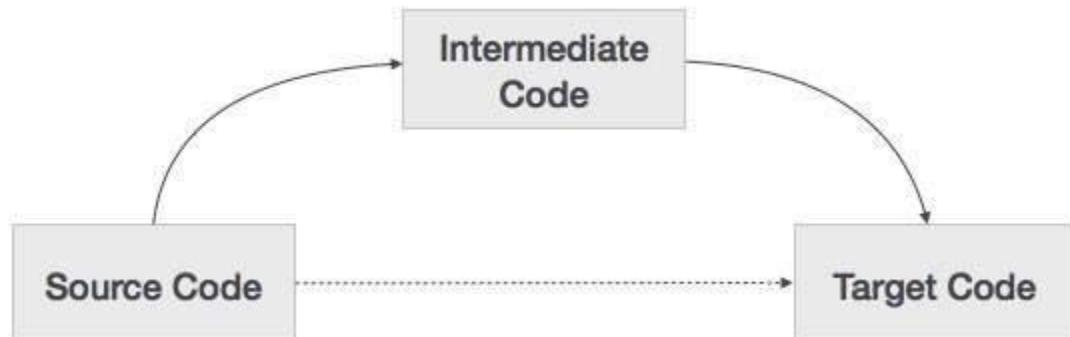


Here are the key components of a lexical compiler:

1. **Input source code:** The input to the lexer is the source code of the program that needs to be compiled. It is usually a text file written in a high-level programming language.
2. **Regular expressions:** The lexer uses regular expressions to define the patterns of the language's syntax. Each regular expression corresponds to a specific token type.
3. **Tokenizer:** The tokenizer is the core component of the lexer. It reads the input source code and identifies the tokens by matching the regular expressions against the input. It then creates a stream of tokens, which is passed on to the next stage of the compiler.

4. Token types: Each token has a type associated with it. Examples of token types include keywords, identifiers, operators, literals, and punctuation.
5. Symbol table: The symbol table is a data structure that keeps track of the identifiers used in the program. It stores information such as the identifier's name, type, and location in memory.
6. Error handling: The lexer also performs error handling by identifying and reporting lexical errors. For example, if the input contains an unrecognized character, the lexer will report an error.

3.2.2 INTERMEDIATE CODE GENERATION

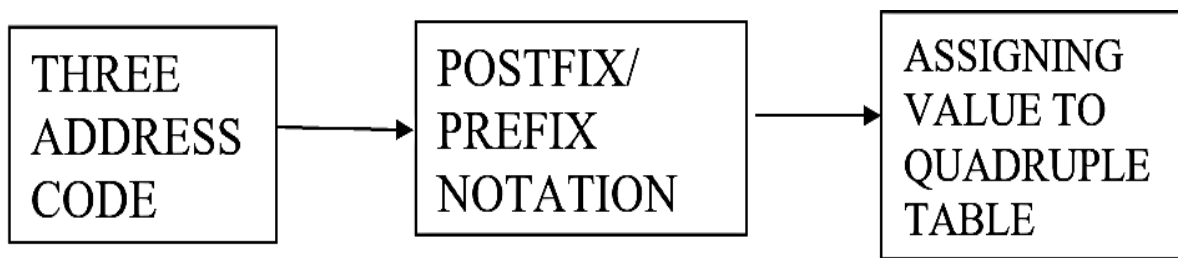


A component diagram is a type of UML diagram that depicts the system's components and their relationships. In the context of intermediate code generation in a compiler, a component diagram can be used to illustrate the flow of information between the different components of the system, namely the source code, intermediate code, and target code.

The source code is the original program written in a high-level language that the compiler receives as input. The intermediate code is an intermediate representation of the program that is generated during the compilation process. The target code is the final output of the compiler, which is often in the form of machine code or assembly language.

In the component diagram, the source code component would be depicted as the input to the system, with an arrow pointing towards the intermediate code component. The intermediate code component would be the central component of the system, as it represents the intermediate representation of the program that is generated during the compilation process. It would have arrows pointing towards both the source code component and the target code component. This indicates that the intermediate code is generated from the source code and is used to generate the target code.

3.2.3 QUADRUPLER

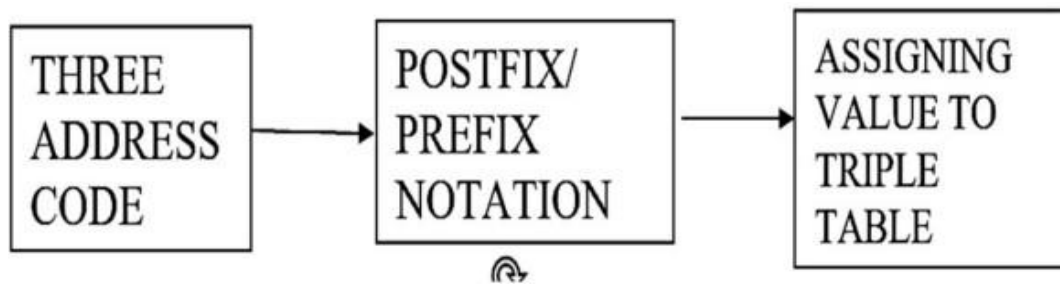


Three-address code would be shown as the system's input on the component diagram, with arrows pointing in the direction of the postfix/prefix expression component and the quadruple component. The three-address code's arithmetic and logical expressions would be translated into postfix/prefix notation by the postfix/prefix expression component. The postfix/prefix expressions would be used to create the quadruples by the quadruple component.

Each sub-component of the quadruple production process, such as parsing the postfix/prefix expressions, constructing the quadruples, and optimising the quadruples, is in charge of a specific step in the quadruple generation process.

The output of the system would be shown as the quadruple component, with an arrow pointing in the direction of the three-address code component. This illustrates how the quadruples are produced.

3.2.4 TRIPLE



The operation code, the source operand, and the destination operand would each be shown as a separate component in a component diagram for a triple, joined by lines that show their relationships. A line linking the operation code component to the source operand component, for instance, would show that the operation code defines the kind of operation to be carried out on the source operand.

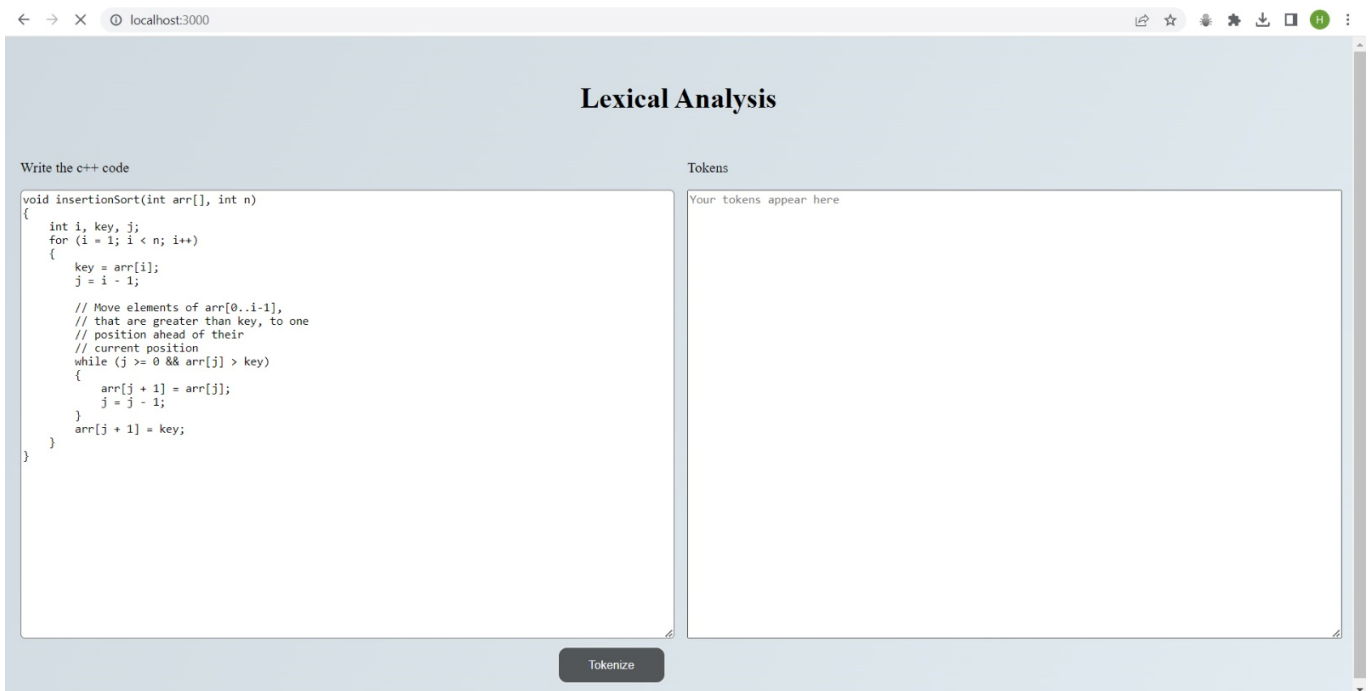
A component diagram for a triple might additionally include additional elements that are utilised across the entire programme, such as variables, constants, and control flow structures, in addition to the three components of the triple. Lines connecting these elements would also show how they relate to one another

CHAPTER 4

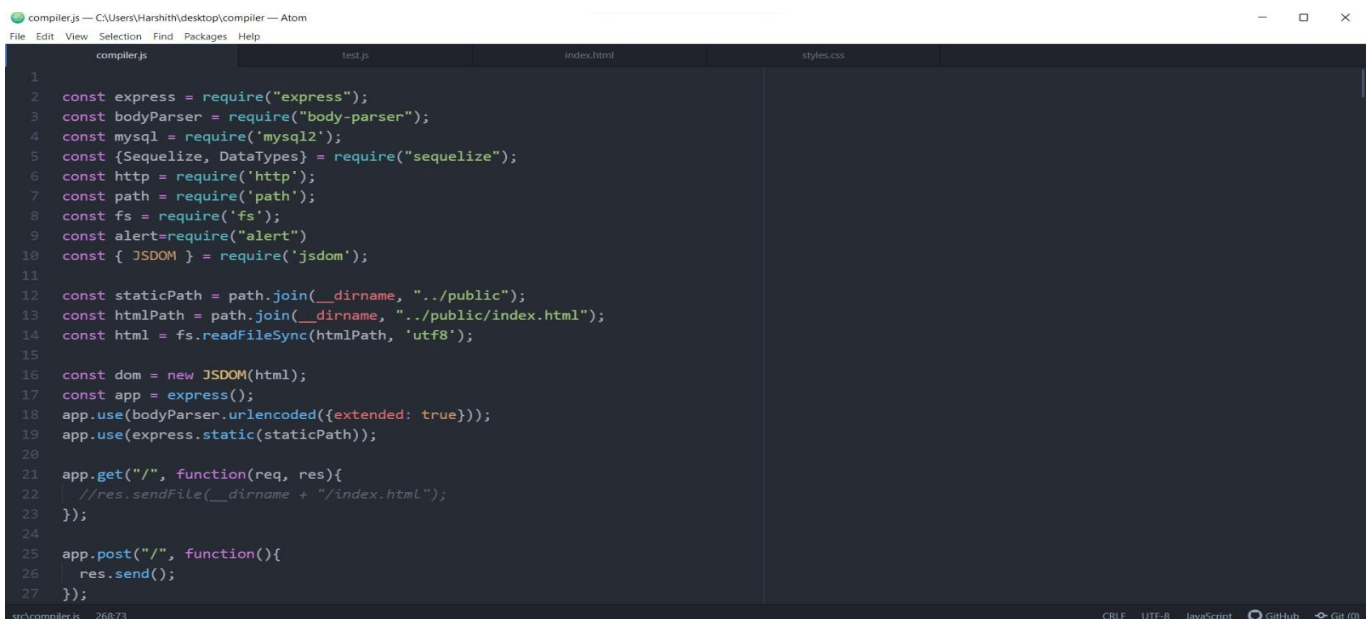
CODING AND TESTING

4.1 LEXICAL ANALYSIS

4.1.1 FRONTEND



4.1.2 BACKEND





```
57
58 function tokenize(code) {
59     const tokens = [];
60     let currentToken = "";
61
62     for (let i = 0; i < code.length; i++) {
63         const char = code[i];
64         const nextChar = code[i + 1];
65
66         if (char === "/" ) {
67             if (nextChar === "/" ) {
68                 // Single-line comment, skip to end of line
69                 i++;
70                 while (code[i] !== "\n" && i < code.length - 1) {
71                     i++;
72                 }
73                 continue;
74             } else if (nextChar === "*") {
75                 // Multi-line comment, skip until end of comment
76                 i += 2;
77                 while (!(code[i] === "*" && code[i + 1] === "/" ) && i < code.length - 1) {
78                     i++;
79                 }
80                 i += 2; // Skip past end of comment
81                 continue;
82             }
83         }
84     }
85 }
```

```
207
208 const connection = mysql.createConnection({
209     host: 'localhost',
210     user: 'root',
211     password: 'Compiler@2023',
212     database: 'tokens_of_code'
213 });
214
215 function insertDataIntoTable(form, o_identifiers, o_keywords, o_numbers, o_operators, o_punctuations, o_strings,code) {
216
217     connection.connect(function(error) {
218         if (error) {
219             console.error('Error connecting to the database: ' + error.stack);
220             return;
221         }
222
223         console.log('Connected to the database with ID ' + connection.threadId);
224     });
225
226     //if (err) throw err;
227     //console.log('Connected to MySQL database!');
228
229     // perform a query to insert data into the database
230     //const to_identifiers = o_identifiers;
231     //const to_keywords = o_keywords;
232     //const to_numbers = o_numbers;
233     //const to_operators = o_operators;
```



```
index.html — C:\Users\Harshith\desktop\compiler — Atom
File Edit View Selection Find Packages Help
compiler.js test.js index.html styles.css

1 <!DOCTYPE html>
2 <html lang="en" dir="ltr">
3   <head>
4     <meta charset="utf-8">
5     <title>Lexical Analysis</title>
6     <link rel="stylesheet" type="text/css" href="css/styles.css">
7   </head>
8   <body>
9
10    <h1>Lexical Analysis</h1>
11
12    <form class="container_parent" action="index.html" method="post">
13
14      <div class="container_1">
15        <p>Write the c++ code</p>
16        <textarea type="text" name="code" rows=5 id = "t_input" placeholder="Write your code here..."></textarea>
17        <button name="submit" type="submit">Tokenize</button>
18      </div>
19      <div class="container_2">
20        <p>Tokens</p>
21        <textarea type="text" name="token_output" id="t_output" placeholder="Your tokens appear here"></textarea>
22      </div>
23    </form>
24
25  </body>
26
27 </html>

public/index.html 22:9 CRLF UTF-8 HTML GitHub Git (0)
```

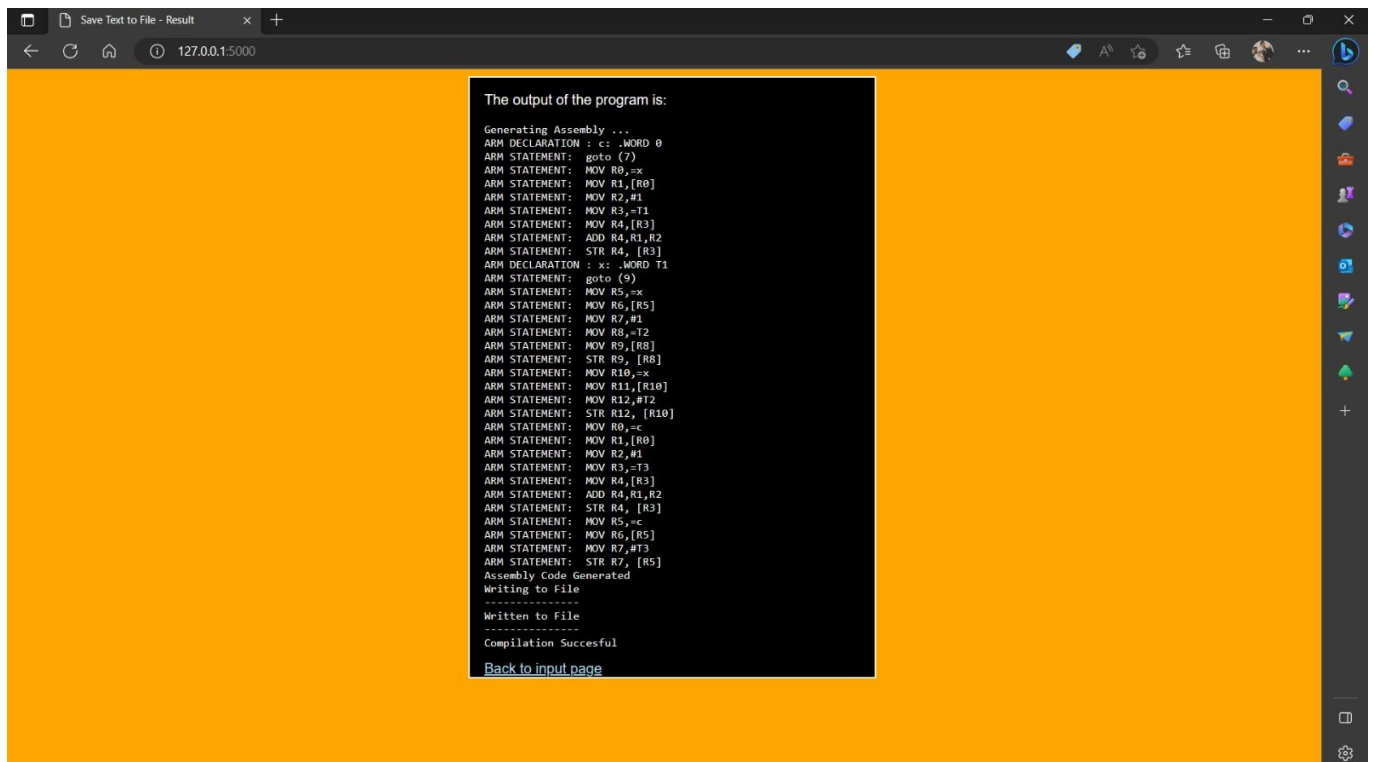
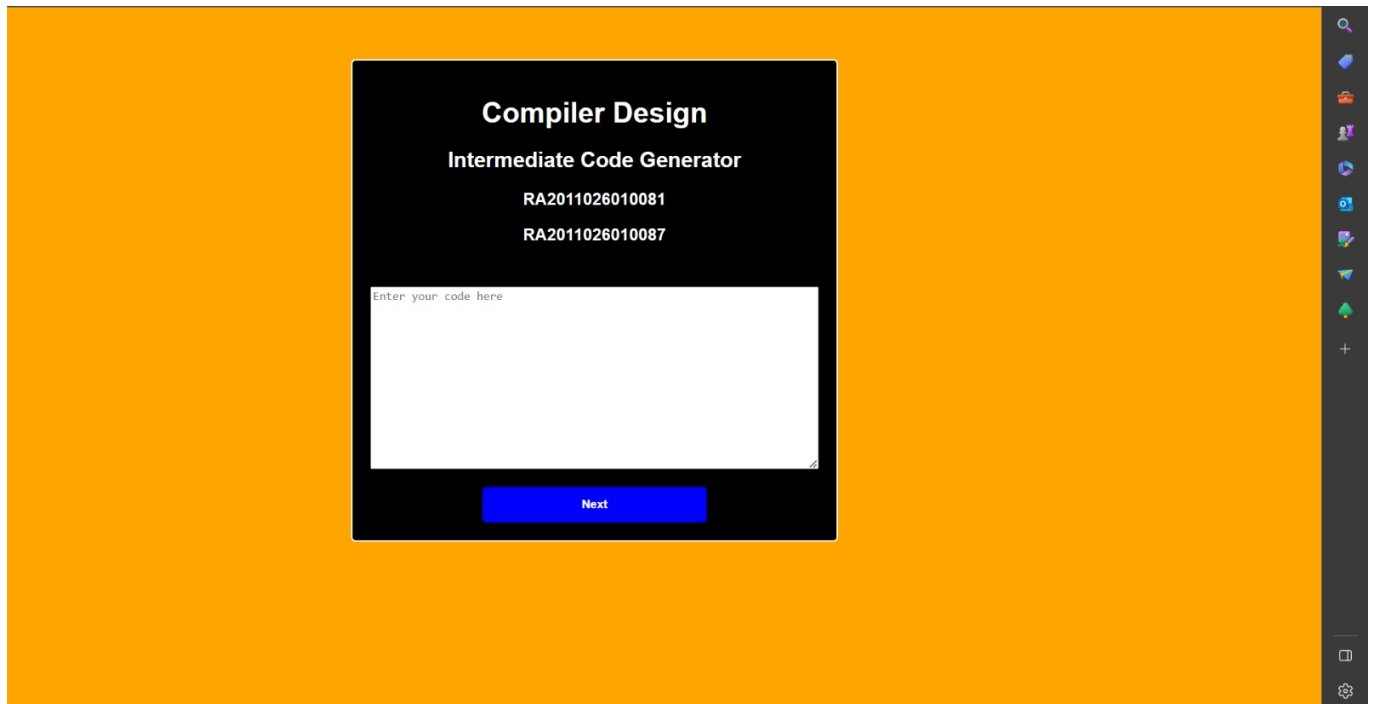
```
styles.css — C:\Users\Harshith\desktop\compiler — Atom
File Edit View Selection Find Packages Help
compiler.js test.js index.html styles.css

1 body{
2   background-image: linear-gradient(120deg, #cfd9df 0%, #e2ebf0 100%);
3 }
4
5
6 h1{
7   text-align: center;
8   margin-top: 50px;
9   margin-bottom: 25px;
10 }
11
12 .container_1 textarea{
13   width: 100%;
14   border-radius: 5px;
15   height: 500px;
16 }
17
18 .container_2 textarea{
19   width: 100%;
20   height: 500px;
21   resize: vertical;
22 }
23
24 button{
25   background-color: #525659;
26   cursor: pointer;
27   margin: 5px;
28 }

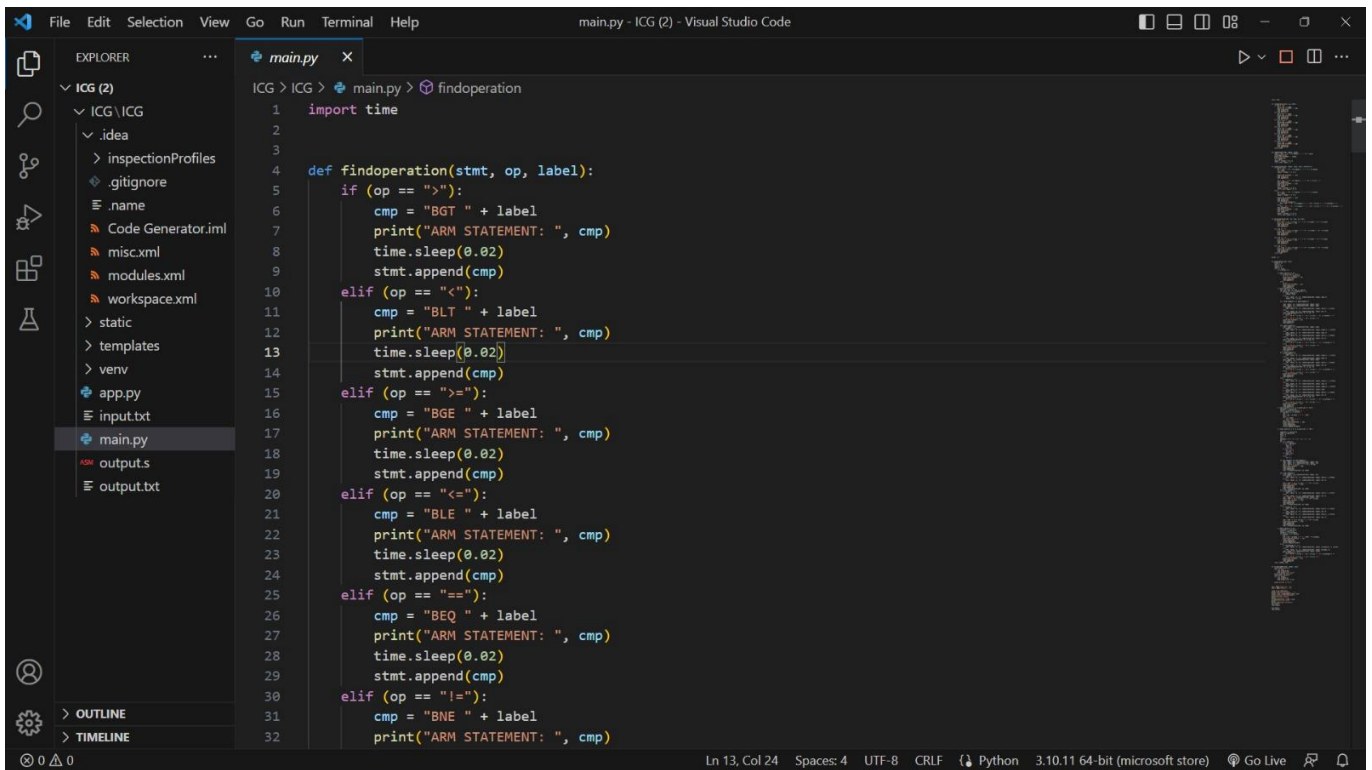
public/css/styles.css 28:22 CRLF UTF-8 CSS GitHub Git (0)
```

4.2 INTERMEDIATE CODE GENERATION

4.2.1 FRONTEND



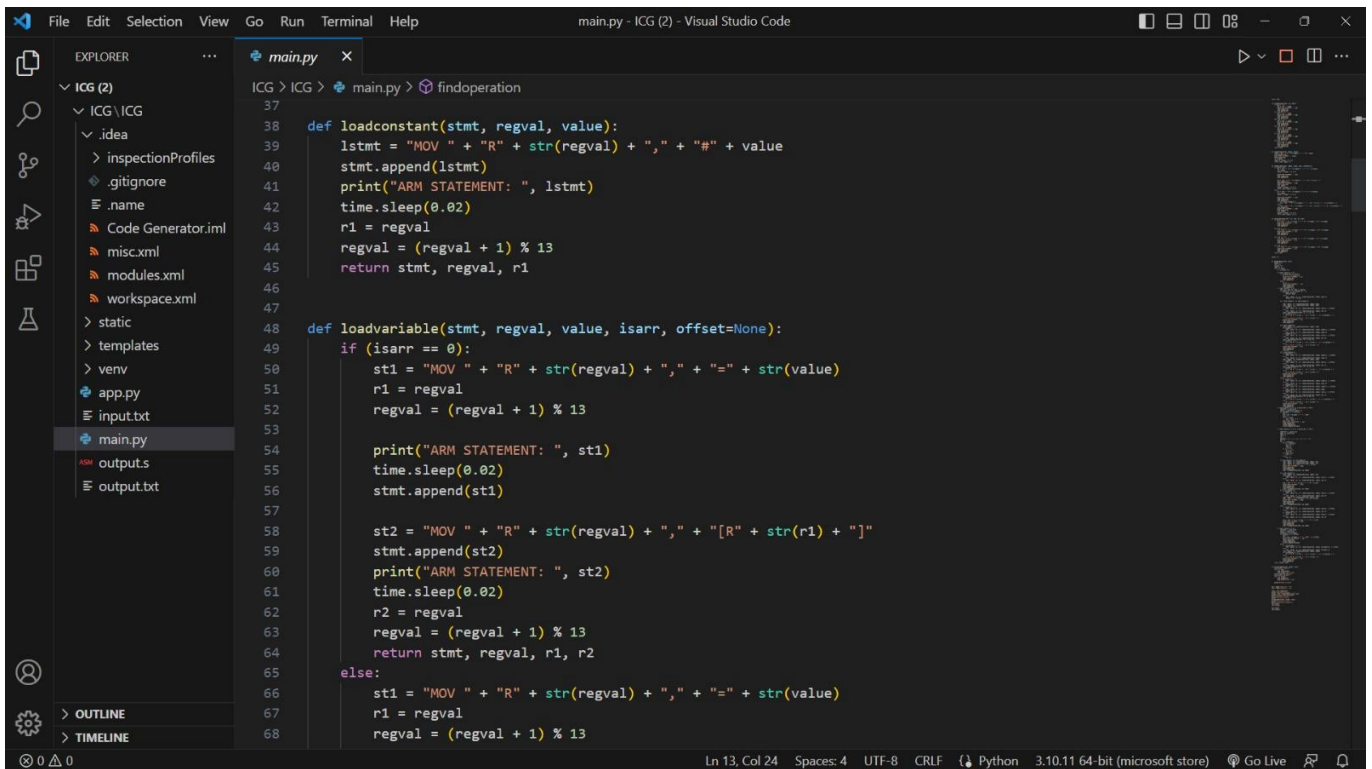
4.2.2 BACKEND



The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure for 'ICG (2)'. The main editor window shows the 'main.py' file with the following code:

```
1 import time
2
3
4 def findoperation(stmt, op, label):
5     if (op == ">"):
6         cmp = "BGT " + label
7         print("ARM STATEMENT: ", cmp)
8         time.sleep(0.02)
9         stmt.append(cmp)
10    elif (op == "<"):
11        cmp = "BLT " + label
12        print("ARM STATEMENT: ", cmp)
13        time.sleep(0.02)
14        stmt.append(cmp)
15    elif (op == ">="):
16        cmp = "BGE " + label
17        print("ARM STATEMENT: ", cmp)
18        time.sleep(0.02)
19        stmt.append(cmp)
20    elif (op == "<="):
21        cmp = "BLE " + label
22        print("ARM STATEMENT: ", cmp)
23        time.sleep(0.02)
24        stmt.append(cmp)
25    elif (op == "=="):
26        cmp = "BEQ " + label
27        print("ARM STATEMENT: ", cmp)
28        time.sleep(0.02)
29        stmt.append(cmp)
30    elif (op == "!="):
31        cmp = "BNE " + label
32        print("ARM STATEMENT: ", cmp)
```

The status bar at the bottom indicates the cursor is at line 13, column 24, with 4 spaces, UTF-8 encoding, CRLF line endings, Python 3.10.11 64-bit (microsoft store), and Go Live extension.



The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project structure for 'ICG (2)'. The main editor window shows the 'main.py' file with the following code:

```
37
38 def loadconstant(stmt, regval, value):
39     lstmt = "MOV " + "R" + str(regval) + "," + "#" + value
40     stmt.append(lstmt)
41     print("ARM STATEMENT: ", lstmt)
42     time.sleep(0.02)
43     r1 = regval
44     regval = (regval + 1) % 13
45     return stmt, regval, r1
46
47
48 def loadvariable(stmt, regval, value, isarr, offset=None):
49     if (isarr == 0):
50         st1 = "MOV " + "R" + str(regval) + "," + "=" + str(value)
51         r1 = regval
52         regval = (regval + 1) % 13
53
54         print("ARM STATEMENT: ", st1)
55         time.sleep(0.02)
56         stmt.append(st1)
57
58         st2 = "MOV " + "R" + str(regval) + "," + "[" + str(r1) + "]"
59         stmt.append(st2)
60         print("ARM STATEMENT: ", st2)
61         time.sleep(0.02)
62         r2 = regval
63         regval = (regval + 1) % 13
64         return stmt, regval, r1, r2
65     else:
66         st1 = "MOV " + "R" + str(regval) + "," + "=" + str(value)
67         r1 = regval
68         regval = (regval + 1) % 13
```

The status bar at the bottom indicates the cursor is at line 13, column 24, with 4 spaces, UTF-8 encoding, CRLF line endings, Python 3.10.11 64-bit (microsoft store), and Go Live extension.

```

65     else:
66         st1 = "MOV " + "R" + str(regval) + ", " + "=" + str(value)
67         r1 = regval
68         regval = (regval + 1) % 13
69
70         print("ARM STATEMENT: ", st1)
71         time.sleep(0.02)
72         stmt.append(st1)
73         if (not offset.isdigit()):
74             st2 = "MOV " + "R" + str(regval) + ", " + "[R" + str(r1) + ", " + str(offset) + "]"
75         else:
76             st2 = "MOV " + "R" + str(regval) + ", " + "[R" + str(r1) + ", " + str("#" + str(offset) + "]"
77         stmt.append(st2)
78         print("ARM STATEMENT: ", st2)
79         time.sleep(0.02)
80         r2 = regval
81         regval = (regval + 1) % 13
82         return stmt, regval, r1, r2
83
84
85 def binaryoperation(stmt, lhs, arg1, op, arg2):
86     if (op == "+"):
87         st = "ADD " + "R" + str(lhs) + ", " + "R" + str(arg1) + ", R" + str(arg2)
88         print("ARM STATEMENT: ", st)
89         time.sleep(0.02)
90         stmt.append(st)
91
92     elif (op == "-"):
93         st = "SUBS " + "R" + str(lhs) + ", " + "R" + str(arg1) + ", R" + str(arg2)
94         print("ARM STATEMENT: ", st)
95         time.sleep(0.02)
96         stmt.append(st)

```

```

98     elif (op == "*"):
99         st = "MUL " + "R" + str(lhs) + ", " + "R" + str(arg1) + ", R" + str(arg2)
100         print("ARM STATEMENT: ", st)
101         time.sleep(0.02)
102         stmt.append(st)
103
104     elif (op == "/"):
105         st = "SDIV " + "R" + str(lhs) + ", " + "R" + str(arg1) + ", R" + str(arg2)
106         print("ARM STATEMENT: ", st)
107         time.sleep(0.02)
108         stmt.append(st)
109     return stmt
110
111
112 offset = 0
113
114
115 def genAssembly(lines, file):
116     vardec = []
117     stmt = []
118     varlist = []
119     regval = 0
120     for i in lines:
121         i = i.strip("\n")
122
123         if (len(i.split()) == 2):
124             if (i.split()[0] == "GOTO"):
125                 st = "B " + i.split()[1]
126                 print("ARM STATEMENT: ", st)
127                 time.sleep(0.02)
128                 stmt.append(st)
129         else:

```

```

129         else:
130             st = i
131             print("ARM STATEMENT: ", st)
132             time.sleep(0.02)
133             stmt.append(st)
134         if (len(i.split()) == 5):
135             lhs, ass, arg1, op, arg2 = i.split()
136             if (lhs[0] == '*' and arg1[0] == '*'):
137                 if (arg2.isdigit()):
138                     offset = arg2
139                 else:
140                     stmt, regval, r1, r2 = loadvariable(stmt, regval, arg2, 0)
141                     offset = "R" + str(r2)
142             elif (arg1.isdigit() and arg2.isdigit()):
143
144
145                 stmt, regval, r1 = loadconstant(stmt, regval, arg1)
146                 stmt, regval, r2 = loadconstant(stmt, regval, arg2)
147                 if (lhs[0] == '*'):
148                     stmt, regval, r3, r4 = loadvariable(stmt, regval, lhs[1:], 1, offset)
149                 else:
150                     stmt, regval, r3, r4 = loadvariable(stmt, regval, lhs, 0)
151                 stmt = binaryoperation(stmt, r4, r1, op, r2)
152                 if (lhs[0] == '*'):
153                     st = "STR R" + str(r4) + ", [R" + str(r3) + ", #", str(offset) + "]"
154                 else:
155                     st = "STR R" + str(r4) + ", [R" + str(r3) + "]"
156                 print("ARM STATEMENT: ", st)
157                 time.sleep(0.02)
158                 stmt.append(st)
159             elif (arg1.isdigit()):

```

```

179         elif (arg2.isdigit()):
180             if (arg1[0] == '*'):
181                 stmt, regval, r1, r2 = loadvariable(stmt, regval, arg1[1:], 1, offset)
182             else:
183                 stmt, regval, r1, r2 = loadvariable(stmt, regval, arg1, 0)
184             stmt, regval, r3 = loadconstant(stmt, regval, arg2)
185             if (lhs[0] == '*'):
186                 stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs[1:], 1, offset)
187             else:
188                 stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs, 0)
189             stmt = binaryoperation(stmt, r5, r2, op, r3)
190             if (lhs[0] == '*'):
191                 st = "STR R" + str(r5) + ", [R" + str(r4) + ", #", str(offset) + "]"
192             else:
193                 st = "STR R" + str(r5) + ", [R" + str(r4) + "]"
194             print("ARM STATEMENT: ", st)
195             time.sleep(0.02)
196             stmt.append(st)
197         else:
198             if (arg1[0] == '*'):
199                 stmt, regval, r1, r2 = loadvariable(stmt, regval, arg1[1:], 1, offset)
200             else:
201                 stmt, regval, r1, r2 = loadvariable(stmt, regval, arg1, 0)
202             if (arg2[0] == '*'):
203                 stmt, regval, r3, r4 = loadvariable(stmt, regval, arg2[1:], 1, offset)
204             else:
205                 stmt, regval, r3, r4 = loadvariable(stmt, regval, arg2, 0)
206             if (lhs[0] == '*'):
207                 stmt, regval, r5, r6 = loadvariable(stmt, regval, lhs[1:], 1, offset)
208             else:
209                 stmt, regval, r5, r6 = loadvariable(stmt, regval, lhs, 0)
210             stmt = binaryoperation(stmt, r6, r2, op, r4)

```



```
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264
```

```
if (len(i.split()) == 4 and i.split()[0] != "ARR"):  
  
    condition = i.split()[1]  
    label = i.split()[3]  
    flag = 0  
    lhs = ""  
    rhs = ""  
    operator = [ ">", "<", ">=", "<=", "=", "!=" ]  
    op = ""  
    for j in condition:  
        if (j in operator):  
            op = op + j  
            flag = 1  
            continue  
        if (j == "="):  
            op = op + j  
            continue  
        if (flag == 0):  
            lhs += j  
        else:  
            rhs += j  
  
    if (rhs.isdigit() and lhs.isdigit()):  
        stmt, regval, r1 = loadconstant(stmt, regval, lhs)  
        stmt, regval, r2 = loadconstant(stmt, regval, rhs)  
        cmp = "CMP R" + str(r1) + ", " + "R" + str(r2)  
        print("ARM STATEMENT: ", cmp)  
        time.sleep(0.02)  
        stmt.append(cmp)  
        stmt = findoperation(stmt, op, label)
```

Ln 13, Col 24 Spaces: 4 UTF-8 CRLF Python 3.10.11 64-bit (microsoft store) Go Live

CHAPTER 5

RESULT

The implementation of the phases of a compiler, including the lexical analyser, intermediate code generation, has been successful in our project. The integration of modern web technologies, such as Node JS and FLASK, has made the compiler more accessible to a wider range of users. Our compiler can effectively translate source code into executable code, making it a useful tool for developers and programmers. Through the development process, we encountered various challenges, such as ensuring the accuracy of the intermediate code generation process, but we were able to overcome these challenges through careful planning and testing. Overall, our project demonstrates our proficiency in compiler development and our ability to apply the concepts and tools learned in class to real-world applications. We believe that our compiler has the potential to be a valuable resource for the programming community and we are excited to see how it will be used in the future.

CHAPTER 6

CONCLUSION

In conclusion, the development of a compiler that implements the various phases of the compilation process has been a challenging and rewarding experience. We now have a better knowledge of how compilers operate on a fundamental level and the significance of each step in the compilation process thanks to the implementation of the lexical analyser and intermediate code generation. The compiler is now more accessible to a larger range of users and more user-friendly thanks to the inclusion of contemporary web technologies. We are excited to see how our compiler is utilised in the future because we think it has the potential to be a useful tool for programmers and developers.

We have learned a lot about software engineering and project management during the development process. We discovered how critical thorough planning, testing, and documentation are to a project's success. We also learned how crucial it is for teams to communicate and work together, and how productive teams may produce results that are more effective and efficient. Overall, our project has been a valuable learning experience that has allowed us to apply the concepts and tools learned in class to a real-world application. We are proud of what we have accomplished and look forward to applying our newfound knowledge to future projects and endeavours.