Department Of Computer Science and Engineering

Course Code: CSE 430

Course Title: Compiler Design Lab

MiniCompiler Design Project

Submitted To:

Md.Shafayatul Haque
Lecturer
Department Of CSE,UAP

Submitted By:

Labanya Saha
Reg No:21201059
Roll:59
Sec:B2

Introduction

This project is a Mini C Compiler developed using Flex (Lex), Bison (Yacc), GCC, and Python. It takes Mini-C programs as input and processes them through all major compiler phases from lexical analysis to final assembly code generation.

The compiler performs tokenization, syntax and semantic analysis, intermediate code generation, and code optimization, finally producing optimized assembly output. A symbol table is maintained to store variable details such as name, type, size, scope, and memory address. Error detection and reporting are integrated at every stage to ensure correct and efficient program compilation.

It demonstrates how high-level source code is systematically converted into low-level assembly instructions. Each phase of the compiler is modular, allowing easy debugging and understanding of internal operations. The project also includes the use of Three Address Code (TAC) or Quadruple representation for intermediate code.

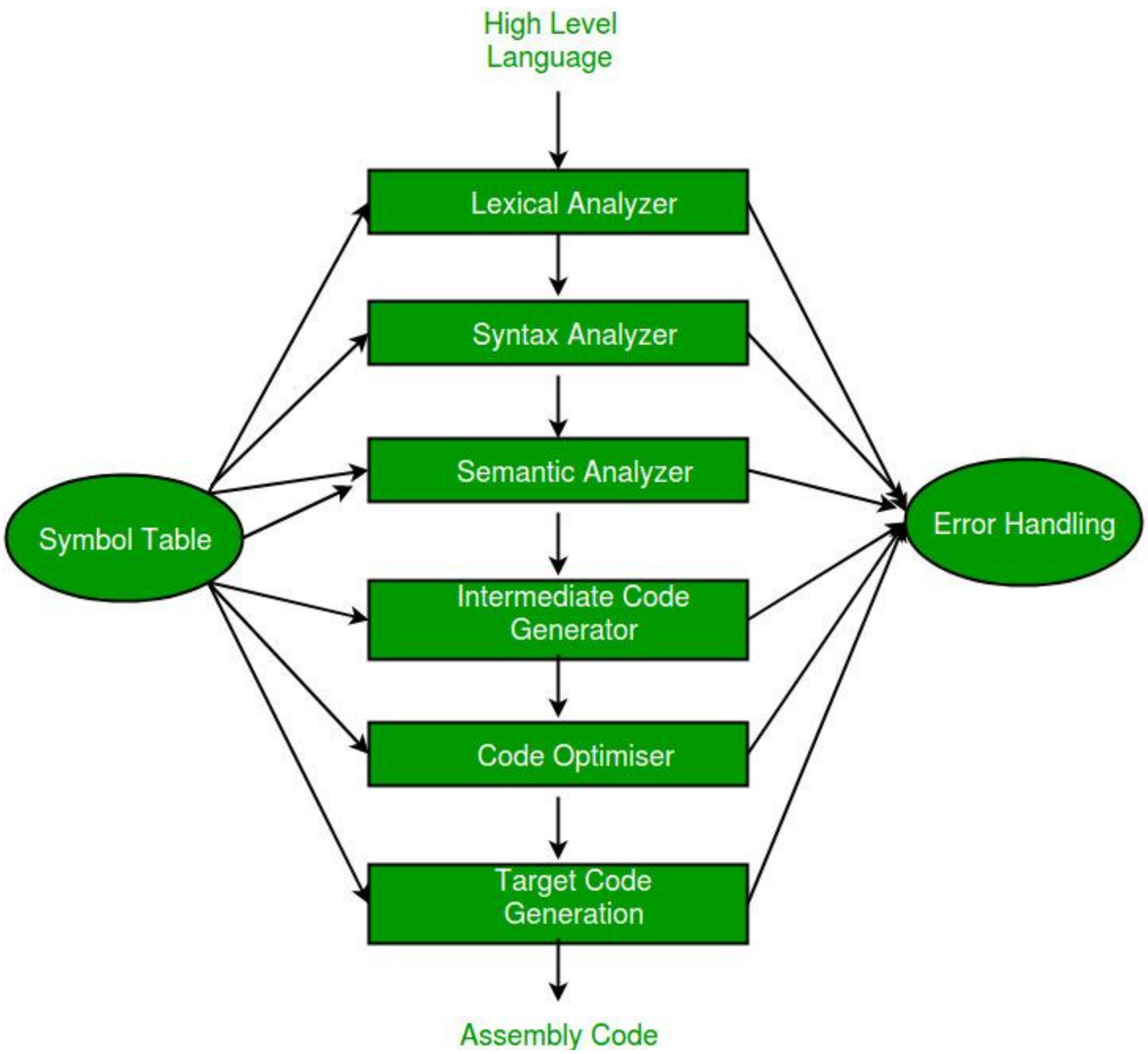
This Mini Compiler is a simplified model of a real-world compiler and helps in understanding how programming languages are translated for machine execution. It serves as an excellent educational tool for students learning compiler design and system programming concepts

Objective

To design and implement a Mini C Compiler that performs all essential phases of compilation, including Lexical Analysis, Symbol Table Construction, Syntax and Semantic Error Analysis, Parsing/Abstract Syntax Tree Generation, Intermediate Code Generation, Code Optimization, and Assembly Code Generation, using Flex, Bison, and Python.

The objective is to understand and demonstrate the internal working of a compiler how a high-level program is converted step by step into machine-level code. This project aims to provide practical experience in compiler design, grammar implementation, error handling, and code translation techniques. It also focuses on building a modular and educational compiler framework that highlights each stage's role in program execution.

Phases Of Compiler:



Block Diagram of the Mini C Compiler:

High-Level Language:
The process starts with a program written in a high-level language such as C or Mini-C.

Lexical Analyzer (Lexer):
This phase scans the input source code and converts it into a sequence of tokens (identifiers, keywords, operators, literals, etc.). It helps detect lexical errors like invalid symbols.

Syntax Analyzer (Parser):
The tokens from the lexer are analyzed according to grammar rules. This phase builds a parse tree or abstract syntax tree (AST) and checks the syntactic correctness of the program.

Semantic Analyzer:
This phase ensures that the program makes logical sense. It checks type compatibility, variable declarations, and scope rules using information from the Symbol Table.

Symbol Table:
The symbol table stores information about identifiers their names, data types, sizes, scopes, and memory addresses. It is accessed by almost every phase of the compiler.

Intermediate Code Generator:
After successful analysis, this phase generates Intermediate Code (IC), usually in the form of Three Address Code (TAC) or Quadruples, which acts as a bridge between source and target code.

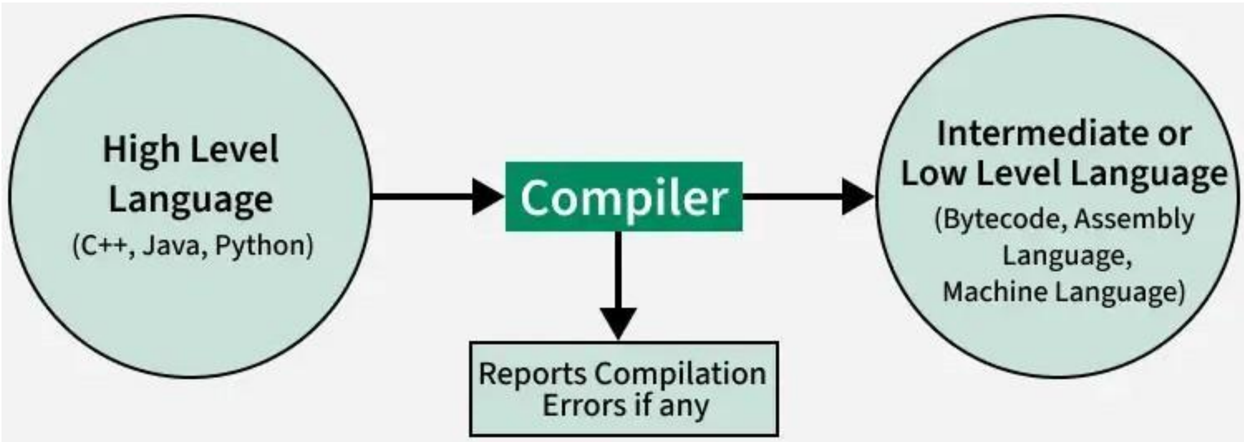
Code Optimizer:
The generated intermediate code is optimized to improve efficiency, reduce redundancy, and enhance performance without changing the program's meaning.

Target Code Generation:
This phase converts the optimized intermediate code into assembly code or machine-level instructions that can be executed by a computer.

Error Handling:
Error detection and handling occur at all major phases lexical, syntax, semantic, and code generation. This ensures reliable and accurate compilation.

Assembly Code Output:
Finally, the compiler produces assembly code, which is the low-level representation of the original high-level program

The Frontend-Backend Split in Compilers:



The compiler frontend analyzes the source code (language-dependent), and the backend synthesizes the final machine code (target-dependent). An intermediate representation (IR) is the bridge between them, allowing the same frontend to be used with different backends for portability.

Here is a breakdown:

- **Frontend**: Reads the source code and performs:

    1)Lexical analysis (scanning)

    2)Syntax analysis (parsing)

    3)Semantic analysis

    4)Intermediate Code Generation

- **Backend**: Optimizes and translates the intermediate code to machine code.

    1)Code Optimization

    2)Code Generation

Github Link:https://github.com/Labanya23/CSE-430-Compiler-Lab/tree/3c6a2f84a4662b73ba75c332ca661b0bcd705268/Mini%20Compiler%20Project_21201059

Key Responsibilities:

Folder Nmae: Mini Compiler Project 21201059

Folder 1 – Lexical Analysis
→ Used **Flex** and **C** to detect keywords, identifiers, digits, and comments.
→ Tested with multiple input files (.txt / .cpp/ .c).

→Command: cd "Lexical"

flex token.l

gcc lex.yy.c – o a.out

./a.out < input0.txt

Folder 2 – Symbol Table
→ Used **Flex** and **C**

→ Implemented **symbol table** using struct and array in C.
→ Stored name, type, size, and address for each variable.

→ Tested with multiple input files (.txt / .cpp/ .c).

→Command: cd "../symboltable"

flex symbol.l

gcc lex.yy.c – o a.out

./a.out input0.cpp

Folder 3 – Syntax Analysis

→ Used Flex, Bison, GCC to check grammar and report syntax errors.

→ Tested with multiple input files (.txt / .cpp/ .c).

→Command: cd "../syntax analyziz"

bison –d lexical.y

flex lexical.l

gcc lex.yy.c lexical.tab.c – o a.out

./a.out < input1.txt

Folder 4 – Abstract Syntax Tree

→ Generated AST using binary tree structure.
→ Linked operators and operands as tree nodes.

→ Used Flex, Bison, GCC to check grammar and report syntax errors.

→ Tested with multiple input files (.txt / .cpp/ .c).

→Command: cd "../parser tree"

bison –d tree.y

flex tree.l

gcc lex.yy.c lexical.tab.c – o treee

./treee < input.cpp

Folder 5 – Intermediate Code Generation

→ Created three-address code (TAC) for each statement.
→ Saved output in text files for next phase.

→ Used Flex, Bison, GCC to check grammar and report syntax errors.

→ Tested with multiple input files (.txt / .cpp/ .c).

→Command: cd "../intermediate code generator"

bison –d icg.y

flex icg.l

gcc lex.yy.c icg.tab.c – o a.out

./a.out < input.cpp

Folder 6 – Code Optimization (Python+Constant Propagation,Constant folding and Dead Code Elimination):

→ Used Python script to remove redundant instructions.

→ Optimized ICG for multiple input files using list-based processing.

→ Tested with multiple input files ((.txt / .cpp/ .c).

→Command: cd "../IC_CODE OPTIMIZATION"

python3 optimizer.py input.txt --print

Folder 7 – Assembly Code Generation (Python)

→ Converted optimized code into assembly format (.s file).

→ Used hash map to track variables and improve memory usage.

→ Tested with multiple input files ((.txt / .cpp/ .c).

→Command: cd "../ Assembly"

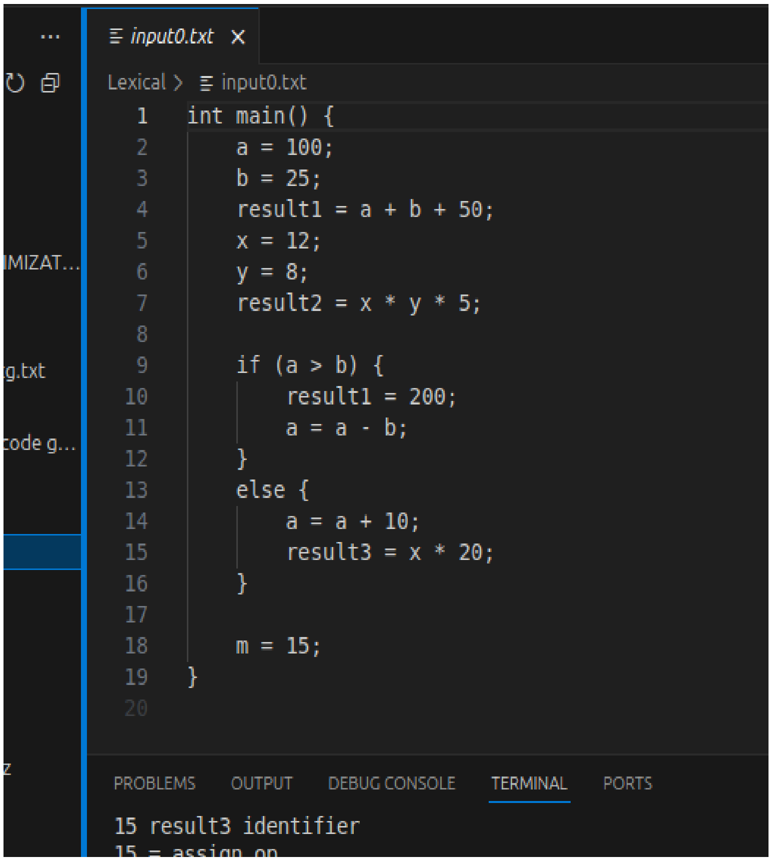
python3 assembly.py icg.txt

python3 assembly.py input.txt

Results

Task1- Lexical analysis (tokenization):

Input:



Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/Lexical
Enter C code (Ctrl+D to finish):
1 int keyword
1 main identifier
1 ( parenthesis
1 ) parenthesis
1 { parenthesis
2 a identifier
2 = assign_op
2 100 constant
2 ; punctuation
3 b identifier
3 = assign_op
3 25 constant
3 ; punctuation
4 result1 identifier
4 = assign_op
4 a identifier
4 + arithmetic_op
4 b identifier
4 + arithmetic_op
4 50 constant
4 ; punctuation
5 x identifier
5 = assign_op
5 12 constant
5 ; punctuation
6 y identifier
6 = assign_op
6 8 constant
6 ; punctuation
7 result2 identifier
7 = assign_op
7 x identifier
7 * arithmetic_op
7 y identifier
7 * arithmetic_op
7 5 constant
7 ; punctuation
```

```
9 if keyword
9 ( parenthesis
9 a identifier
9 > logical_op
9 b identifier
9 ) parenthesis
9 { parenthesis
10 result1 identifier
10 = assign_op
10 200 constant
10 ; punctuation
11 a identifier
11 = assign_op
11 a identifier
11 - arithmetic_op
11 b identifier
11 ; punctuation
12 } parenthesis
13 else keyword
13 { parenthesis
14 a identifier
14 = assign_op
14 a identifier
14 + arithmetic_op
14 10 constant
14 ; punctuation
15 result3 identifier
15 = assign_op
15 x identifier
15 * arithmetic_op
15 20 constant
15 ; punctuation
16 } parenthesis
18 m identifier
18 = assign_op
18 15 constant
18 ; punctuation
19 } parenthesis
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/Lexical$
```

Task 2- Symbol Table:

Input:

```cpp
int main() {
    a = 100;
    b = 25;
    result1 = a + b + 50;
    x = 12;
    y = 8;
    result2 = x * y * 5;

    if (a > b) {
        result1 = 200;
        a = a - b;
    }
    else {
        a = a + 10;
        result3 = x * 20;
    }

    m = 15;
}
```

Output:

```
<ARITHOP,*,7>
<ARITHOP,*,7>
<DIGIT,5,7>
<KEYWORD,if,9>
<OPBRACK,(,9>
<RELOP,>,9>
<CLOSEBRACK,),9>
<OPBRACK,{,9>
<ASOP,=,10>
<DIGIT,200,10>
<ASOP,=,11>
<ARITHOP,-,11>
<CLOSEBRACK,},12>
<KEYWORD,else,13>
<OPBRACK,{,13>
<ASOP,=,14>
<ARITHOP,+,14>
<DIGIT,10,14>
<IDENTIFIER,result3,15>
<ASOP,=,15>
<ARITHOP,*,15>
<DIGIT,20,15>
<CLOSEBRACK,},16>
<IDENTIFIER,m,18>
<ASOP,=,18>
<DIGIT,15,18>
<CLOSEBRACK,},19>

TOKEN#  DATA TYPE       TOKEN_TYPE      TOKEN_VALUE     LINE of CODE    SCOPE   VALUE
1       int     IDENTIFIER      a       2 4 9 11 11 14 14       1       9999999
2       int     IDENTIFIER      b       3 4 9 11        1       9999999
3       int     IDENTIFIER      result1 4 10    1       9999999
4       int     IDENTIFIER      x       5 7 15  1       9999999
5       int     IDENTIFIER      y       6 7     1       9999999
6       int     IDENTIFIER      result2 7       1       9999999
7       int     IDENTIFIER      result3 15      2       9999999
8       int     IDENTIFIER      m       18      1       9999999
```

Task 3- Syntax Error generated by parser along with token and symbol table

Input file: with error

```
input1.txt
syntax analyziz > input1.txt
1  int main() {
2      int a_val, b_val;
3      a_val = 77;
4      b_val = 33;
5
6      int sum1 = a_val + b_val + 11;
7      int counter1 = 5;  // example initial value
8      int counter = counter1;
9
10     while(counter1) {
11         counter--;
12     }
13
14     if(counter == 0) {
15         counter = counter + 2;
16     } else {
17         counter = 0;
18     }
19 }
20
```

Output:

```
decl:int
main
decl:int
id:a_val
id:b_val
id:a_val
assignop:=
num:77
id:b_val
assignop:=
num:33
decl:int
id:sum1
assignop:=
id:a_val
id:b_val
num:11
decl:int
id:counter1
assignop:=
num:5
decl:int
id:counter
assignop:=
id:counter1
while
id:counter1
Line no: 10
 The error is: syntax error, unexpected ')', expecting comparisionop
id:counter
unary:--
if
id:counter
compop:==
num:0
id:counter
```

Task 4- Parser/ Abstract Syntax Tree:

Input:

```
input.cpp
parser tree > input.cpp > main()
int main(){

    int i, a, b;
    int nume=3.45;
    for(i = 0;  i < 10; i++){
        a=i;
    }
    i=1;
}
```

Output:

```
parser tree > input.cpp > main()
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS            bash - parser tree

 identifier            i     int      1       3       1
 identifier            a     int      1       3       0
 identifier            b     int      1       3       -
 identifier         nume     int      1       4       3



Abstract Syntax Tree
                                                        main

                                                  assign

                                            assign            =

                                   =           for         i        1

                              i        0         ++          =

                                   <        i        a        i

                            i       10

main    ( assign    ( assign    ( =  i  0  )    ( for    ( ++    ( <  i  10  )  i  )    ( =  a
i  )  )  )    ( =  i  1  )  )
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/parser tree$
```

Task 5-Intermediate Code Generation:

Input file C++ Code:

```cpp
#include<iostream>
using namespace std;
int main()
{
    int i=0;
    int a=0;
    for(i=0;i<10;i++)
    {
        a=a+i;
    }
    a=2*a-1;
}
```

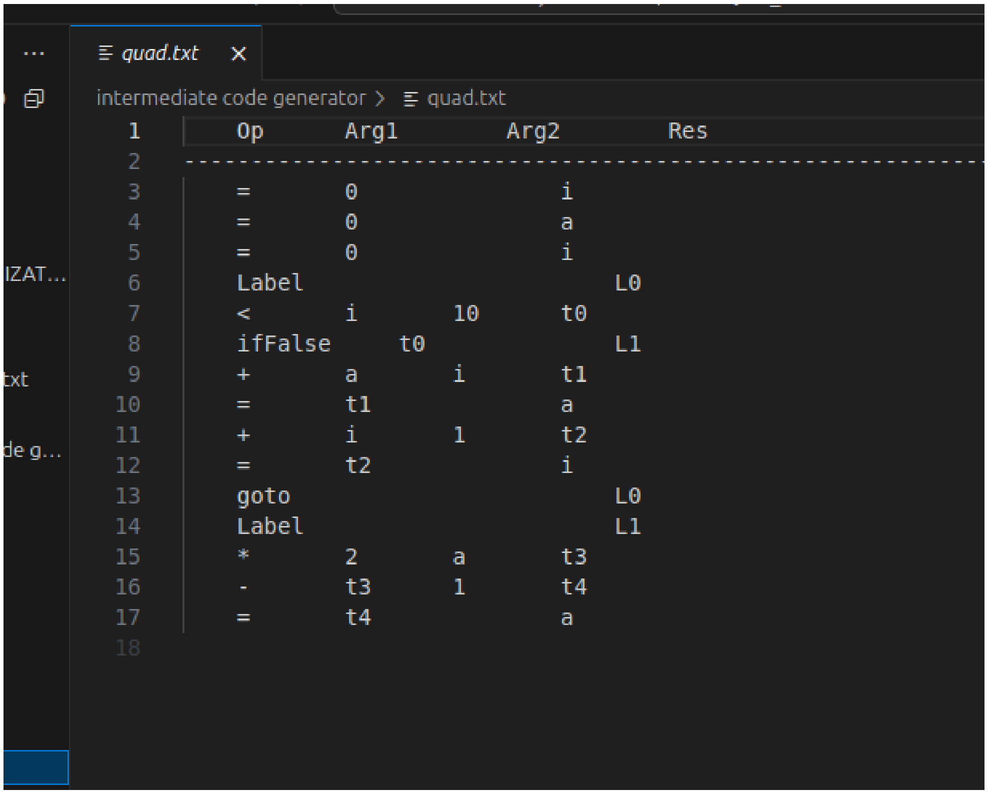
```
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/intermediate code generator$ gcc lex.yy.c icg.tab.c -o a.out
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/intermediate code generator$ /.a.out < input0.cpp
bash: /.a.out: No such file or directory
labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/intermediate code generator$ ./a.out < input0.cpp

labanya@ubuntuu:~/Pictures/Mini Compiler Project_21201059/intermediate code generator$
```

Output Of Tac:

```
i = 0
a = 0
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
t1 = a + i
a = t1
t2 = i + 1
i = t2
goto L0
L1:
t3 = 2 * a
t4 = t3 - 1
a = t4
```

Quadruple format



```
Op      Arg1        Arg2        Res
-----------------------------------------------------------
=       0               i
=       0               i
=       0               i
Label                       L0
<       i       10      t0
ifFalse     t0              L1
+       a       i       t1
=       t1              a
+       i       1       t2
=       t2              i
goto                        L0
Label                       L1
*       2       a       t3
-       t3      1       t4
=       t4              a
```

Task 6- Code Optimization(Constant Propagation,Constant folding and Dead Code Elimination):

Input before optimization

```
int main() {
    a = 100;
    b = 25;
    result1 = a + b + 50;
    x = 12;
    y = 8;
    result2 = x * y * 5;

    if (a > b) {
        result1 = 200;
        a = a - b;
    }
    else {
        a = a + 10;
        result3 = x * 20;
    }

    m = 15;
}
```

Output After Optimization:

```
--- Original ICG ---
i = 2
t0 = i > 1
ifFalse t0 goto L0
t1 = i + 1
i = t1
goto L1
L0:
t2 = i - 1
i = t2
L1:
t3 = i + 3
i = t3
L2:
t4 = i < 10
ifFalse t4 goto L3
t5 = i + 2
a = t5
t6 = i + 1
i = t6
goto L2
L3:
t7 = a * 3
t8 = t7 + 4
a = t8
i = t8
L4:
t9 = i < 11
ifFalse t9 goto L5
t10 = i - 2
a = t10
goto L4
L5:
t11 = 2 * a
t12 = i + t11
```

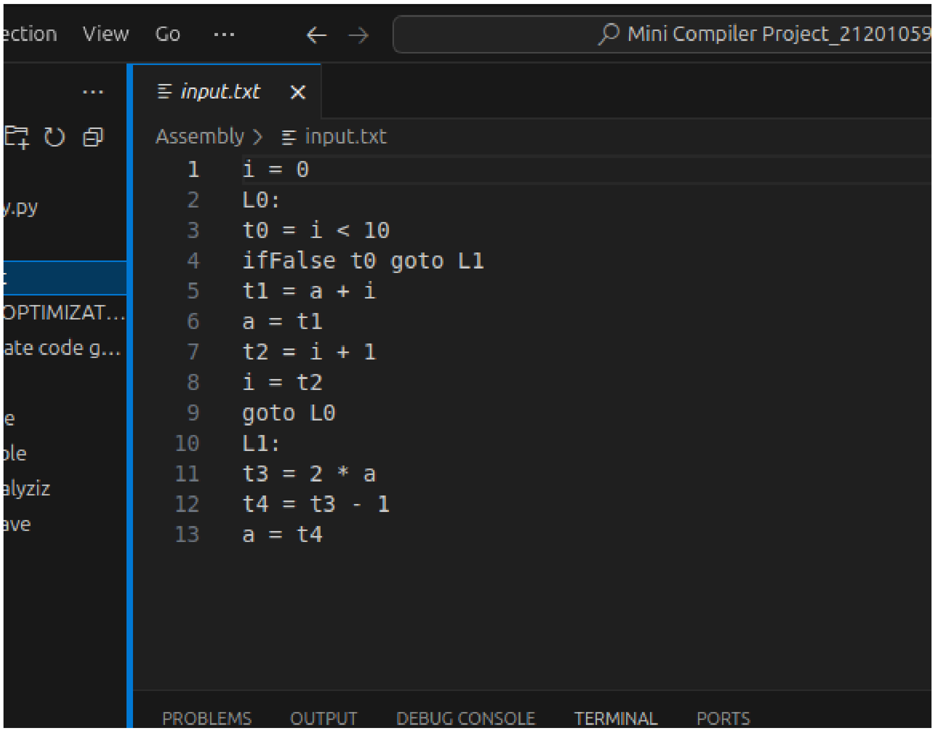
```
--- After Constant Propagation ---
i = 2
t0 = 2 > 1
ifFalse t0 goto L0
t1 = 2 + 1
i = t1
goto L1
L0:
t2 = 2 - 1
i = t2
L1:
t3 = 2 + 3
i = t3
L2:
t4 = 2 < 10
ifFalse t4 goto L3
t5 = 2 + 2
a = t5
t6 = 2 + 1
i = t6
goto L2
L3:
t7 = a * 3
t8 = t7 + 4
a = t8
i = t8
L4:
t9 = 2 < 11
ifFalse t9 goto L5
t10 = 2 - 2
a = t10
goto L4
L5:
t11 = 2 * a
```

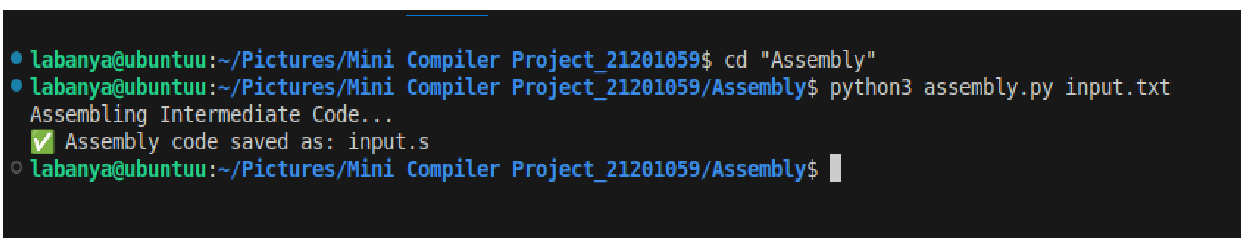
```
--- After Constant Folding --
i = 2
t0 = 1
ifFalse t0 goto L0
t1 = 3
i = t1
goto L1
L0:
t2 = 1
i = t2
L1:
t3 = 5
i = t3
L2:
t4 = 1
ifFalse t4 goto L3
t5 = 4
a = t5
t6 = 3
i = t6
goto L2
L3:
t7 = a * 3
t8 = t7 + 4
a = t8
i = t8
L4:
t9 = 1
ifFalse t9 goto L5
t10 = 0
a = t10
goto L4
L5:
```

```
--- After Dead Code Elimination ---
i = 2
t0 = 1
ifFalse t0 goto L0
t1 = 3
i = t1
goto L1
t2 = 1
i = t2
t3 = 5
i = t3
t4 = 1
ifFalse t4 goto L3
t5 = 4
a = t5
t6 = 3
i = t6
goto L2
t7 = a * 3
t8 = t7 + 4
a = t8
i = t8
t9 = 1
ifFalse t9 goto L5
t10 = 0
a = t10
goto L4
t11 = 2 * a
t12 = 2 + t11
a = t12
```

Task 7- Assembly Code Generation:

Input:





Output:

input.s

Assembly > input.s

```
.text
L0:
MOV R0, =i
MOV R1, [R0]
MOV R2, =t0
MOV R3, [R2]
NOP R3, #1, R10
STR R3, [R2]
MOV R4, =i
MOV R5, [R4]
CMP R5, #10
BGE L1
MOV R6, =a
MOV R7, [R6]
MOV R8, =i
MOV R9, [R8]
MOV R10, =t1
MOV R11, [R10]
ADD R11, R7, R9
STR R11, [R10]
MOV R12, =i
MOV R0, [R12]
MOV R1, =t2
MOV R2, [R1]
ADD R2, #0, R1
STR R2, [R1]
MOV R3, =i
MOV R4, [R3]
MOV R5, #t2
STR R5, [R3]
B L0
L1:
MOV R6, =a
MOV R7, [R6]
```

input.s

Assembly > input.s

```
ADD R2, #0, R1
STR R2, [R1]
MOV R3, =i
MOV R4, [R3]
MOV R5, #t2
STR R5, [R3]
B L0
L1:
MOV R6, =a
MOV R7, [R6]
MOV R8, =t3
MOV R9, [R8]
MUL R9, #2, R7
STR R9, [R8]
MOV R10, =t3
MOV R11, [R10]
MOV R12, =t4
MOV R0, [R12]
SUBS R0, #11, R1
STR R0, [R12]
MOV R1, =a
MOV R2, [R1]
MOV R3, #t4
STR R3, [R1]
SWI 0x011

.DATA
i: .WORD 0
a: .WORD t1
```

Conclusion

Through this project, I successfully implemented a simplified compiler that performs all major phases from lexical analysis to assembly code generation. Each stage was tested using multiple input files to ensure correctness. The project enhanced my understanding of compiler design concepts, syntax parsing, and code optimization. It also gave me practical experience using Flex,Bison, C,C++ and Python, and helped me understand how source code is translated step by step into machine-level instructions.