



Report on

“Mini C Compiler for ‘If-For-While’ Constructs”

*Submitted in partial fulfilment of the requirements for **Semester VI***

Compiler Design Laboratory

**Bachelor of Technology
in
Computer Science & Engineering**

Submitted by:

Chetan B Chavhannavar	PES1201701375
V.S.Vishruth	PES1201701878
Vithal P Nakod	PES1201701746

Under the guidance of

Suhas G K
Assistant Professor
PES University, Bengaluru

January – May 2020

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

Chapter No.	Title	Page No.
1.	INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)	03
2.	ARCHITECTURE OF LANGUAGE: <ul style="list-style-type: none"> What all have you handled in terms of syntax and semantics for the chosen language. 	03
3.	LITERATURE SURVEY (if any paper referred or link used)	03-04
4.	CONTEXT FREE GRAMMAR (which you used to implement your project)	04
5.	DESIGN STRATEGY (used to implement the following) <ul style="list-style-type: none"> SYMBOL TABLE CREATION ABSTRACT SYNTAX TREE INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator). TARGET CODE GENERATION 	06
6.	IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following): <ul style="list-style-type: none"> SYMBOL TABLE CREATION ABSTRACT SYNTAX TREE (internal representation) INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ASSEMBLY CODE GENERATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator). Provide instructions on how to build and run your program. 	07-09
7.	RESULTS AND possible shortcomings of your Mini-Compiler	09
8.	SNAPSHOTS (of different outputs)	10-17
9.	CONCLUSIONS	18
10.	FURTHER ENHANCEMENTS	18
REFERENCES/BIBLIOGRAPHY		18

Chapter 1- Introduction:

- We have implemented a Mini C Compiler which is capable of handling If, For and While Constructs along with basic C Operations.
- Given source program in C can be translated to a symbol table, abstract syntax tree, intermediate code, optimized intermediate code and target assembly code.
- The languages chosen for this implementation are Lex, Yacc and C. Lex which identifies pre-defined patterns and generates tokens for the patterns matched and Yacc which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code for the source code.

Chapter 2- Architecture:

- Used Lex to create the scanner for our language, Yacc to implement grammar rules to the token generated in the scanner phase.
- Arithmetic expressions with +, -, *, /, ++, -- are handled.
- Boolean expressions with >, <, >=, <=, == are handled.
- Ignore comments and white-spaces.
- Types: int, float, char, void.
- Constructs:
 - If Loop
 - While Loop
 - For Loop
- Error handling reports undeclared variables.
- Error handling also reports syntax errors with line numbers and Error handling related to scope and declaration.
- Code Optimisation Technique: Constant Folding and Common Sub Expression Elimination.

Chapter 3- Literature Survey and References:

- Course Material for Compiler Design CD 2020:
The course material shared by the department for compiler design helped us in understanding the various concepts that are required to build the Mini C Compiler and also to understand each phase of the compiler in great details. It also helped us to understand the basic concepts required in building each phase of the compiler from Lexical and Syntax Phase to generating the target assembly language code.

https://drive.google.com/open?id=1_HEqMdujON3L1ICtA059LjPY1HxhY0F5

- Lex and Yacc Tutorial:

Lex and yacc tutorial helped us to understand how to use the lex and yacc tools to build each phase of the compiler from Lexical Phase to the final phase.

<https://www.isi.edu/~pedro/Teaching/CSCI565Fall15/Materials/LexAndYaccTutorial.pdf>

- Reference for C Grammar:

The most important part that has to be done before starting to build the different phases of the compiler is the context free grammar for the language. It is this grammar that has rules and productions for the language and on the basis of this context free grammar we start building the compiler phase by phase.

<https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>

Chapter 4- Context Free Grammar:

Start	->	Type main() { Statements }
Type	->	int float char
Statements	->	Statements Statements Declarative Conditional_Statement Loop print_Statement AssignExpr UnaryExpr return num; ε
print_Statement	->	printf (“ String ”); printf (‘Fs’, Args); print_Statement print_Statement
String	->	a b c z A B C Z 0 1 9 Special_characters \\n \\t ε String String

Special_characters	->	@ # \$ % ^ & * ? , . ; ' ''
Fs	->	String %Format Fs String %Format
Format	->	d s c
Args	->	id , Args id
Declarative	->	Type varList ; Type Declarative_Init
Declarative_Init	->	AssignExpr
varList	->	varList , id id Declarative_Init varList , Declarative_Init
Conditional_Statement	->	if (condition) { Statements; }
Loop	->	while (condition) { Statements; break; continue; } for (AssignExpr ; Condition ; AssignExpr) { Statements; break; continue; }
UnaryExpr	->	++UE UE++ --UE UE—
UE	->	id (E)
condition	->	Boolean Expr Expr logOP Expr
Expr	->	relExpr logExpr E
relExpr	->	E relOP E

logExpr	->	E logOP E
logOP	->	! &&
relOP	->	< > <= >= == !=
AssignExpr	->	id=E ;
E	->	E + T E - T T
T	->	T * F T / F F
F	->	id num (E)

Chapter 5- Design Strategy:

- **Symbol Table:** Symbol table is a data structure that tracks the current bindings of identifiers for performing semantic checks and generating code efficiently. We have implemented the symbol table as a linked list of structures. The members of the structures include variable name, line of declaration, data type, value, scope. Every new variable encountered in the program is entered into the symbol table.
- **Abstract Syntax Tree:** We have implemented a binary tree to represent the abstract syntax tree internally and we output the tree in pre-order manner.
- **Intermediate Code Generation:** The intermediate code is generated on the fly, as we parse the code and check its grammar, the intermediate code is generated.
- **Code Optimization:** To increase efficiency the code optimization is done on the generated ICG. We have implemented constant folding and common subexpression elimination.

- **Error Handling:** In case of syntax error, the compilation is halted, and an error message along with the line number where error occurred is displayed. Semantic errors multiple declaration of the same variable, invalid assignment, scope errors are also explicitly pointed out.
- **Target Code Generation:** Python Program is written which converts the optimized ICG to Target Assembly Code.

Chapter 6- Implementation Details:

Lexical and Syntax Phase:

- The tools we have used for implementing the code are lex and yacc. The lex file has all the tokens specified with the help of regular expressions and the yacc file has grammar rules with corresponding actions.
- The scanner that is lexical analyser transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler. A global variable 'yyval' is used to record the value of each lexeme scanned. 'yytext' is the lex variable that stores the matched string. Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- The rules are regular expressions which have corresponding actions that execute on a match with the source input.
- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- As the code is being parsed, the tokens are generated and comments and extra spaces are ignored. For every new variable encountered, it is entered into the symbol table along with its attributes.
- Yacc tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

Semantic Phase:

- A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. As each line is parsed, the actions associated with the grammar rules is executed. A union is used which consists of int, float and char and void type to use it for assigning data type of each variable.
- Semantics Analysis uses available information in the Symbol table to check for Semantics.
- The scope check is done by having a variable which increments on every level of nesting. In this manner, the scope is checked for each variable and error messages are displayed if anything is used out of scope.
- \$1 is used to refer to the first token in the given production and \$\$ is used to refer to the resultant of the given production. Expressions are evaluated and the values of the used variables are updated accordingly. At the end of the parsing, the updated symbol table is displayed.

Abstract Syntax Tree:

- Once parsing is successful, we generate an abstract tree and it is shown in pre-order manner. To build the tree, a structure is maintained which has pointers to its children and a container for its data value.
- A tree structure representing the syntactical flow of the code is generated in this phase. For expressions associativity is indicated using the %left and %right fields.

Intermediate Code Generation:

- Intermediate code generator receives input from its predecessor phase, semantic analyser. The intermediate code generation also happens on the fly. Intermediate code tends to be machine independent code.
- Three Address Code: A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form $x = y \text{ op } z$, here x, y, z will have an address (memory location).
- The intermediate code generated is also displayed in quadruple format. It is shown with 4 columns- operator, operand1, operand2, and result.

Optimisation of ICG:

- After generating intermediate code, optimization is done by doing constant folding and common subexpression elimination.
- Constant Folding: Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time.
- An expression is a Common Subexpression (CSE) if the expression was previously computed and the values of the operands have not changed since the previous computation. Re-computing the expression can be eliminated by using the value of the previous computation.

Target Code Generation:

- After generating the optimized ICG, we generate the Target Assembly Code.

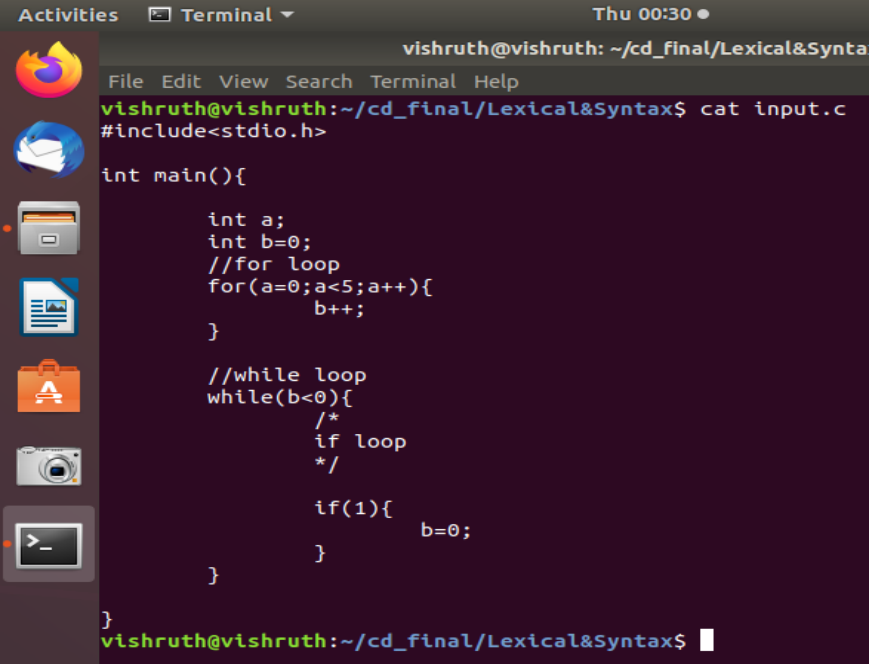
Chapter 7- Results and Possible Shortcomings:

- We have seen the design strategies and implementation of the different stages involved in building a mini C compiler. Traversing the symbol table is time consuming as we have implemented a linked list, no random access possible.

Chapter 8- Snapshots:

1. Lexical and Syntax Phase:

Input file:



A terminal window titled 'vishruth@vishruth: ~/cd_final/Lexical&Synta' with a menu bar (File, Edit, View, Search, Terminal, Help) and a sidebar with application icons. The terminal displays the following C code:

```
vishruth@vishruth:~/cd_final/Lexical&Syntax$ cat input.c
#include<stdio.h>

int main(){

    int a;
    int b=0;
    //for loop
    for(a=0;a<5;a++){
        b++;
    }

    //while loop
    while(b<0){
        /*
        if loop
        */

        if(1){
            b=0;
        }
    }
}
```

The prompt `vishruth@vishruth:~/cd_final/Lexical&Syntax$` is visible at the bottom.

Output:

```
Activities Terminal vishruth@vishruth: ~/cd_final/Lexical&Syntax
vishruth@vishruth:~/cd_final/Lexical&Syntax$ sh run.sh

UE17CS354

CD LABORATORY

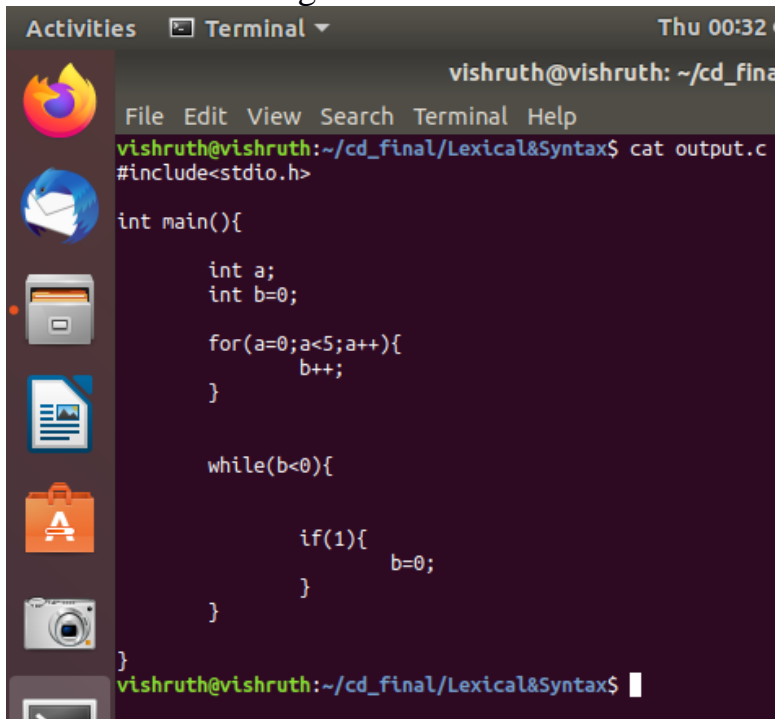
-----

TOKEN GENERATION

#          SPECIAL SYMBOL
#include   KEYWORD
<          SPECIAL SYMBOL
stdio.h    HEADER FILE
>          SPECIAL SYMBOL
int        KEYWORD
main       IDENTIFIER
(          SPECIAL SYMBOL
)          SPECIAL SYMBOL
{          SPECIAL SYMBOL
int        KEYWORD
a          IDENTIFIER
:          SPECIAL SYMBOL
```

```
Activities Terminal vishruth@vishruth: ~/cd_final/L
vishruth@vishruth:~/cd_final/L
++          OPERATOR
)           SPECIAL SYMBOL
{           SPECIAL SYMBOL
b           IDENTIFIER
++          OPERATOR
;           SPECIAL SYMBOL
}           SPECIAL SYMBOL
while       KEYWORD
(           SPECIAL SYMBOL
b           IDENTIFIER
<           SPECIAL SYMBOL
0           INTEGER NUMBER
)           SPECIAL SYMBOL
{           SPECIAL SYMBOL
if          KEYWORD
(           SPECIAL SYMBOL
1           INTEGER NUMBER
)           SPECIAL SYMBOL
{           SPECIAL SYMBOL
b           IDENTIFIER
=           SPECIAL SYMBOL
0           INTEGER NUMBER
;           SPECIAL SYMBOL
}           SPECIAL SYMBOL
}           SPECIAL SYMBOL
}           SPECIAL SYMBOL
Successful
vishruth@vishruth:~/cd_final/Lexical&Syntax$
```

Code after removing comments:

A terminal window titled 'Terminal' with a dark background. The prompt is 'vishruth@vishruth: ~/cd_final'. The command 'cat output.c' has been executed. The output shows C code for the Lexical & Syntax phase. The code includes a main function with two loops: a for loop that increments 'a' from 0 to 4, and a while loop that increments 'b' until it is no longer less than 0. Inside the while loop, there is an if statement that checks if '1' is true, which it is, and then sets 'b' to 0. The code is as follows:

```
vishruth@vishruth:~/cd_final/Lexical&Syntax$ cat output.c
#include<stdio.h>

int main(){

    int a;
    int b=0;

    for(a=0;a<5;a++){
        b++;
    }

    while(b<0){

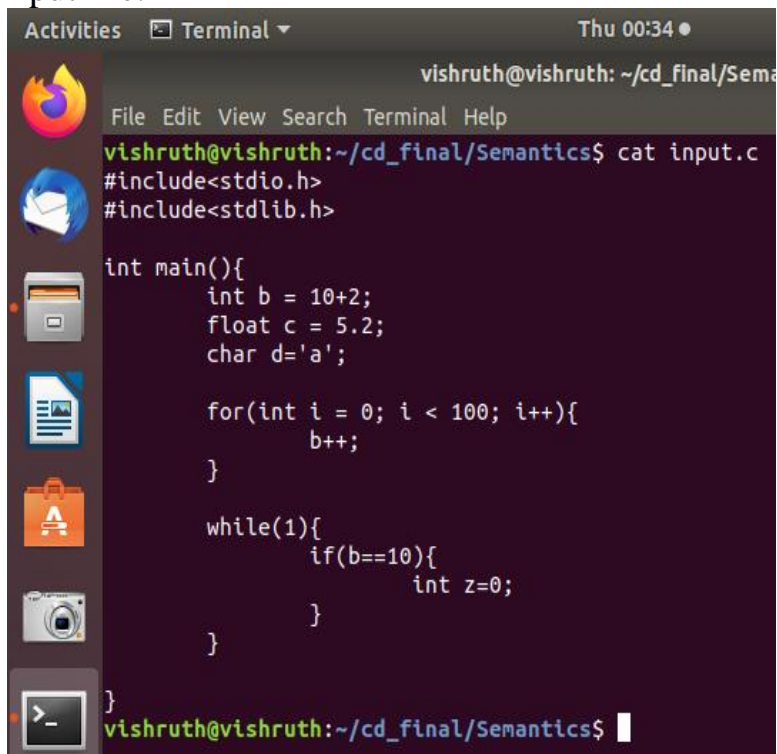
        if(1){
            b=0;
        }

    }

}
```

2. Semantic Phase:

Input file:

A terminal window titled 'Terminal' with a dark background. The prompt is 'vishruth@vishruth: ~/cd_final/Semantics'. The command 'cat input.c' has been executed. The output shows C code for the Semantic phase. The code includes a main function with variable declarations for 'b' (int), 'c' (float), and 'd' (char). It also has a for loop that increments 'i' from 0 to 99, and a while loop that checks if 'b' is equal to 10. Inside the while loop, there is an if statement that checks if 'b' is equal to 10, which it is, and then declares a variable 'z' of type 'int'. The code is as follows:

```
vishruth@vishruth:~/cd_final/Semantics$ cat input.c
#include<stdio.h>
#include<stdlib.h>

int main(){
    int b = 10+2;
    float c = 5.2;
    char d='a';

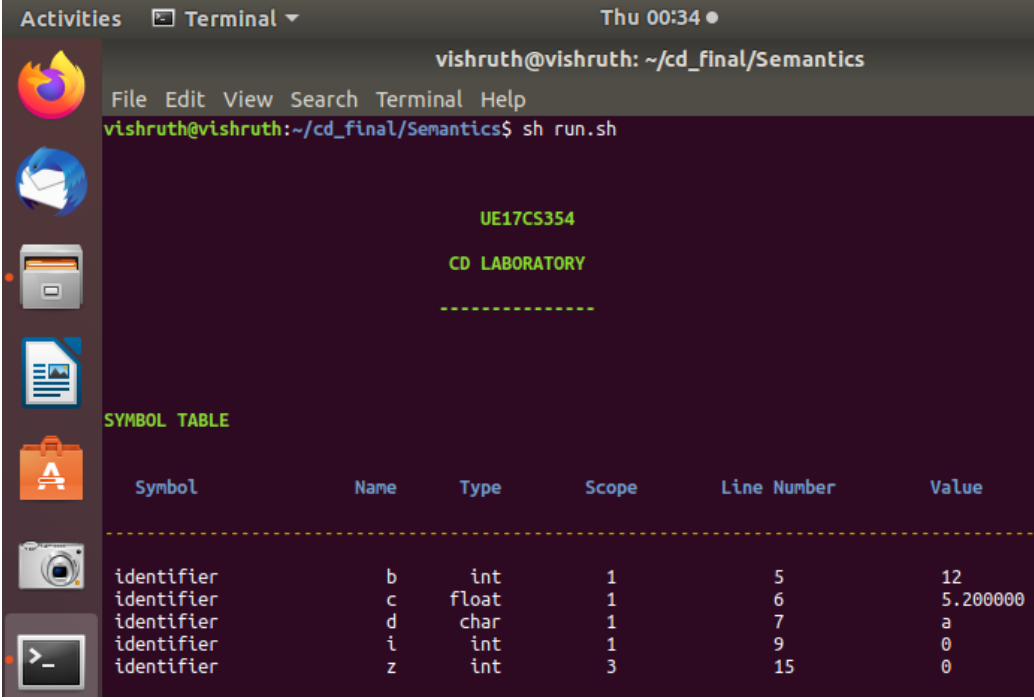
    for(int i = 0; i < 100; i++){
        b++;
    }

    while(1){
        if(b==10){
            int z=0;
        }

    }

}
```

Output:



```
vishruth@vishruth: ~/cd_final/Semantics
File Edit View Search Terminal Help
vishruth@vishruth:~/cd_final/Semantics$ sh run.sh

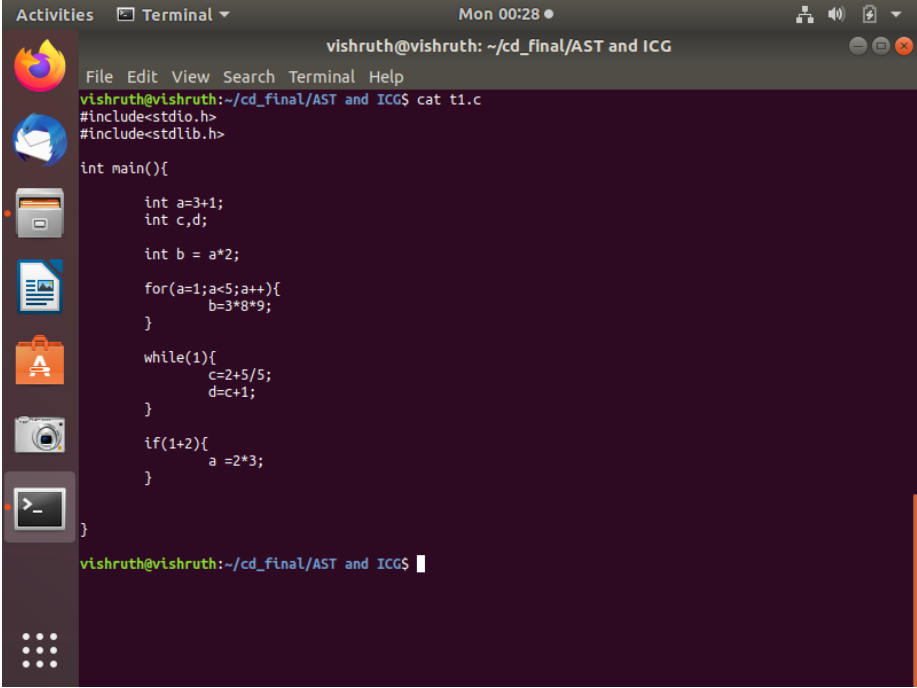
UE17CS354
CD LABORATORY
-----

SYMBOL TABLE
```

Symbol	Name	Type	Scope	Line Number	Value
identifier	b	int	1	5	12
identifier	c	float	1	6	5.200000
identifier	d	char	1	7	a
identifier	i	int	1	9	0
identifier	z	int	3	15	0

3. Abstract Syntax Tree:

Input file:



```
vishruth@vishruth: ~/cd_final/AST and ICG
File Edit View Search Terminal Help
vishruth@vishruth:~/cd_final/AST and ICG$ cat t1.c
#include<stdio.h>
#include<stdlib.h>

int main(){

    int a=3+1;
    int c,d;

    int b = a*2;

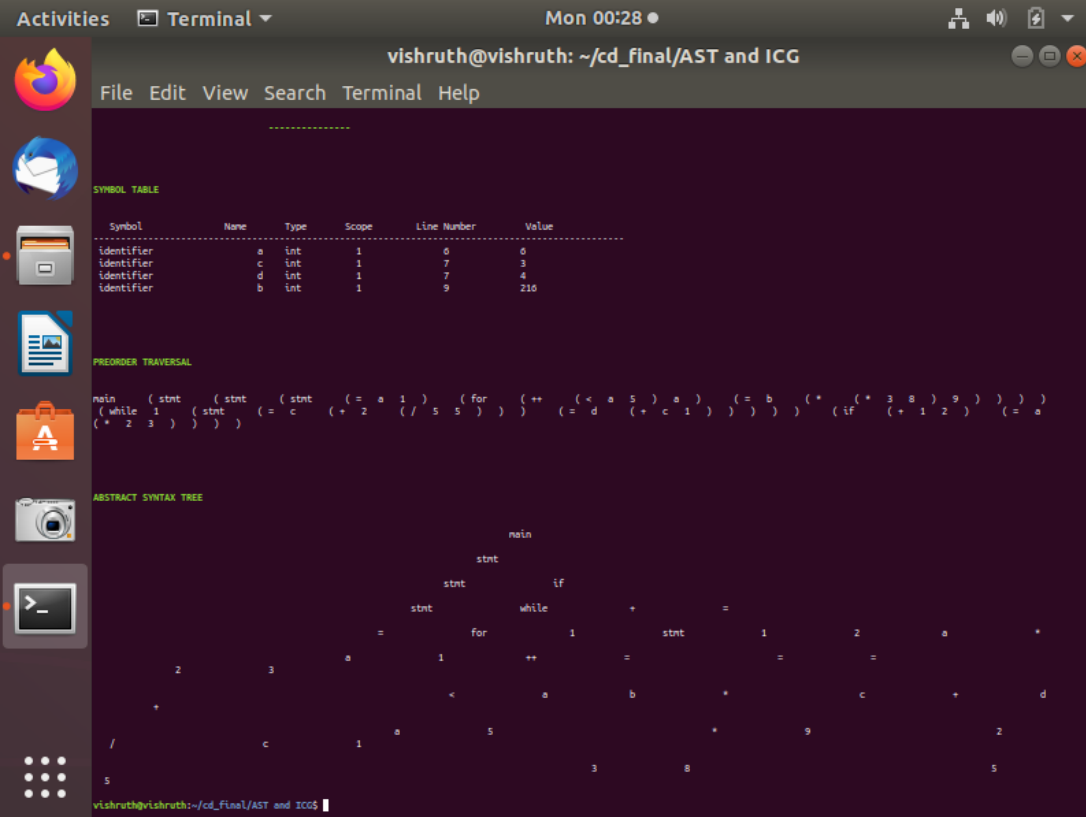
    for(a=1;a<5;a++){
        b=3*8*9;
    }

    while(1){
        c=2+5/5;
        d=c+1;
    }

    if(1+2){
        a =2*3;
    }

}
```

Output:

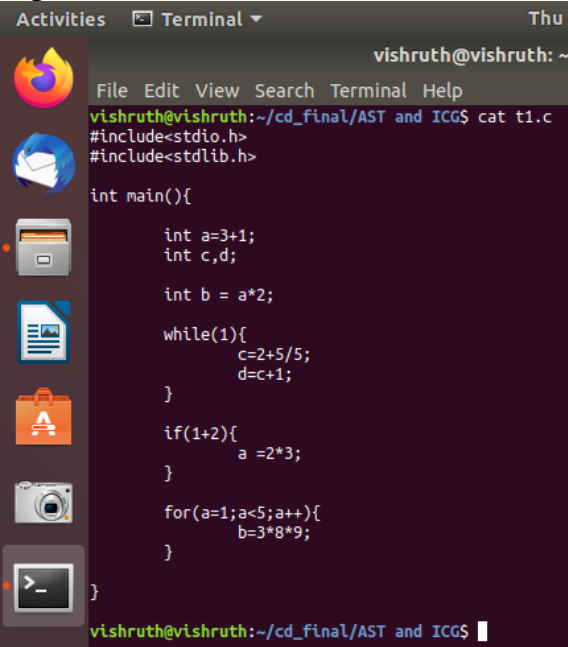


Terminal window showing the output of a compiler. The window title is "vishruth@vishruth: ~/cd_final/AST and ICG". The output is as follows:

```
-----  
SYMBOL TABLE  
-----  
Symbol      Name  Type  Scope  Line Number  Value  
-----  
identifier  a     int   1       6             0  
identifier  c     int   1       7             3  
identifier  d     int   1       7             4  
identifier  b     int   1       9            210  
-----  
PREORDER TRAVERSAL  
main ( stnt ( stnt ( stnt ( = a 1 ) ( for ( ++ ( < a 5 ) a ) ( = b ) ( * ( * 3 8 ) 9 ) ) ) )  
( * 2 3 ) ) ) )  
-----  
ABSTRACT SYNTAX TREE  
-----  
main  
  stnt  
    stnt  
      stnt  
        =  
          2  
        +  
          3  
        *  
          a  
      for  
        1  
        ++  
        <  
          a  
        b  
        *  
          c  
        +  
          d  
    while  
      1  
      stnt  
        =  
          1  
        =  
          2  
        a  
        *  
          3  
          8  
          9  
          2  
          5  
  if  
    1  
    2  
    a  
    *  
    3  
    8  
    9  
    2  
    5
```

4. Intermediate Code Generation:

Input file:



Terminal window showing the input C code for intermediate code generation. The window title is "vishruth@vishruth: ~". The input code is as follows:

```
#include<stdio.h>
#include<stdlib.h>

int main(){
    int a=3+1;
    int c,d;

    int b = a*2;

    while(1){
        c=2+5/5;
        d=c+1;
    }

    if(1+2){
        a =2*3;
    }

    for(a=1;a<5;a++){
        b=3*8*9;
    }
}
```

Output:

```
Activities Terminal Thu 00:48 ●
vishruth@vishruth: ~/cd_final/AST and ICG
File Edit View Search Terminal Help

INTERMEDIATE CODE

t0 = 3 + 1
a = t0
t1 = a * 2
b = t1

L0:
ifFalse t1 goto L1
t2 = 5 / 5
t3 = 2 + t2
c = t3
t4 = c + 1
d = t4
goto L0

L1:

L2:
t5 = 1 + 2
ifFalse t5 goto L3
t6 = 2 * 3
a = t6

L3:
a = t6

L4:
t7 = a < 5
ifFalse t7 goto L5
t8 = 3 * 8
t9 = t8 * 9
b = t9
t10 = a + 1
a = t10
goto L4

L5:
```

```
Activities Terminal Thu 00:49 ●
vishruth@vishruth: ~/cd_final/AST and ICG
File Edit View Search Terminal Help

L5:

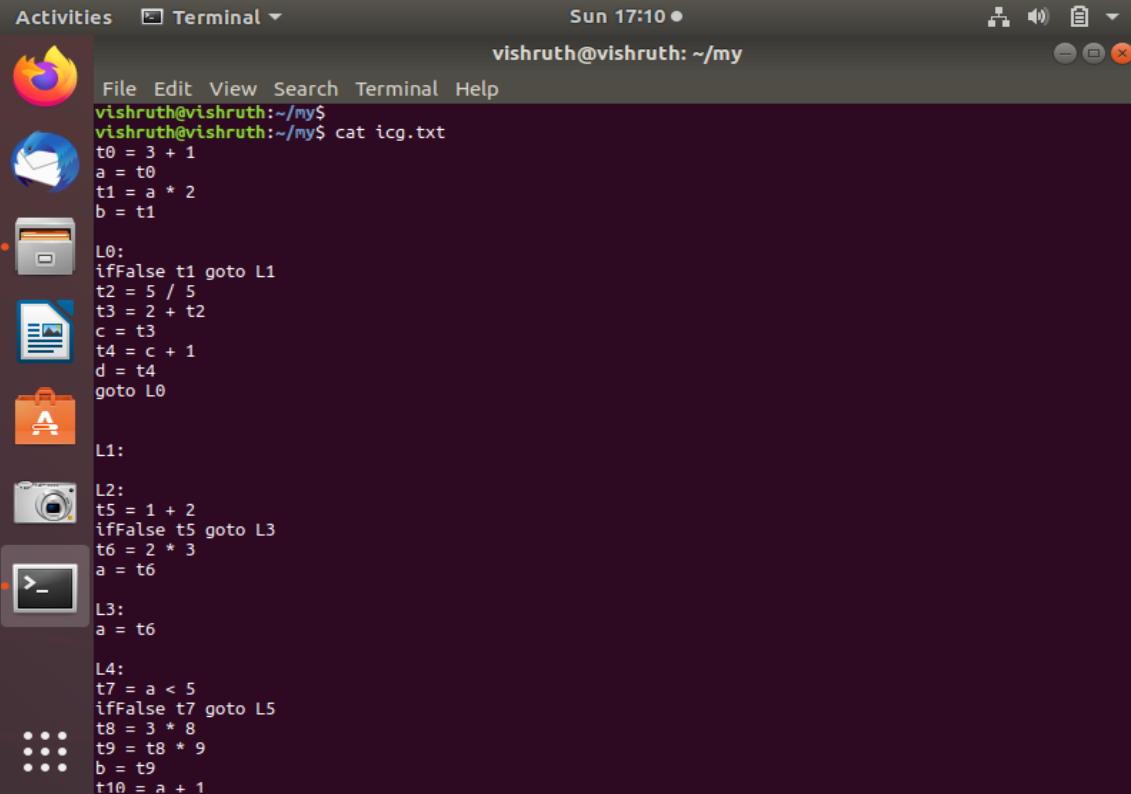
QUADRUPLE FORMAT

Op      Arg1      Arg2      Res
-----
+        3         1         t0
=        t0         a
*        a         2         t1
=        t1         b
Label    L0
ifFalse  t1         L1
/        5         t2
+        2         t2      t3
=        t3         c
+        c         1         t4
=        t4         d
goto     L0
Label    L1
Label    L2
+        1         2         t5
ifFalse  t5         L3
*        2         3         t6
=        t6         a
Label    L3
=        t6         a
Label    L4
<        a         5         t7
ifFalse  t7         L5
*        3         8         t8
*        t8         9         t9
=        t9         b
+        a         1         t10
=        t10        a
goto     L4
Label    L5

vishruth@vishruth:~/cd_final/AST and ICG$
```

5. Optimized Intermediate Code Generation:

Input:



```
vishruth@vishruth:~/my$ cat icg.txt
t0 = 3 + 1
a = t0
t1 = a * 2
b = t1

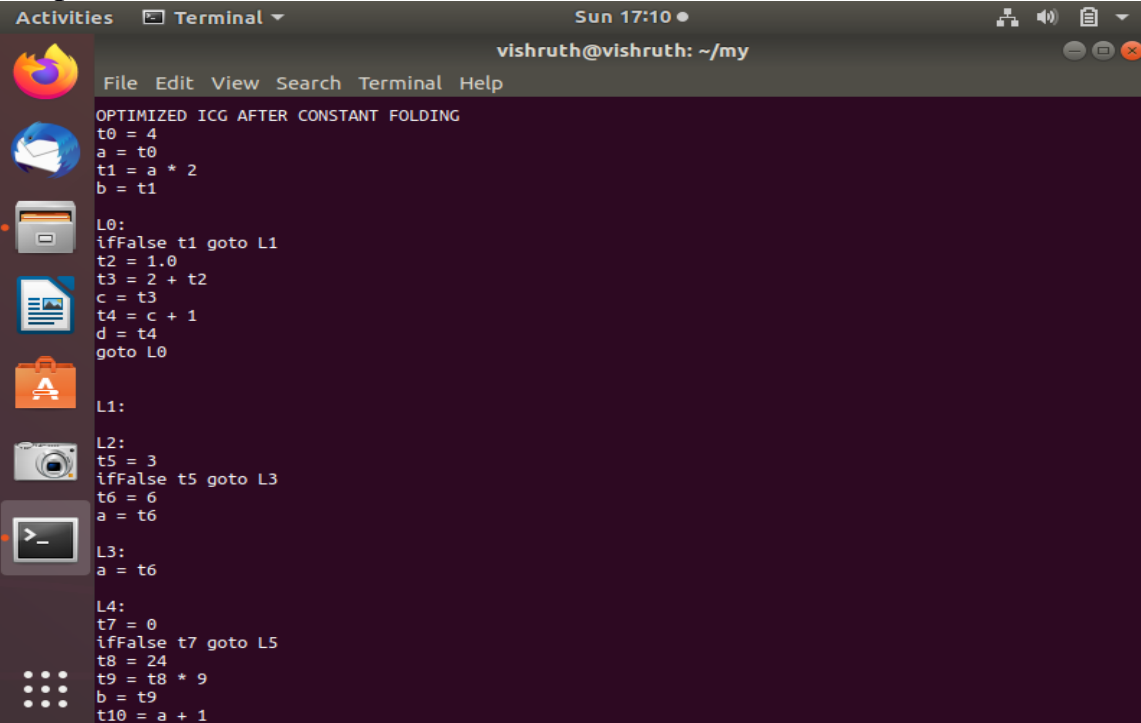
L0:
ifFalse t1 goto L1
t2 = 5 / 5
t3 = 2 + t2
c = t3
t4 = c + 1
d = t4
goto L0

L1:
L2:
t5 = 1 + 2
ifFalse t5 goto L3
t6 = 2 * 3
a = t6

L3:
a = t6

L4:
t7 = a < 5
ifFalse t7 goto L5
t8 = 3 * 8
t9 = t8 * 9
b = t9
t10 = a + 1
```

Output:



```
vishruth@vishruth:~/my$ cat icg.txt
OPTIMIZED ICG AFTER CONSTANT FOLDING
t0 = 4
a = t0
t1 = a * 2
b = t1

L0:
ifFalse t1 goto L1
t2 = 1.0
t3 = 2 + t2
c = t3
t4 = c + 1
d = t4
goto L0

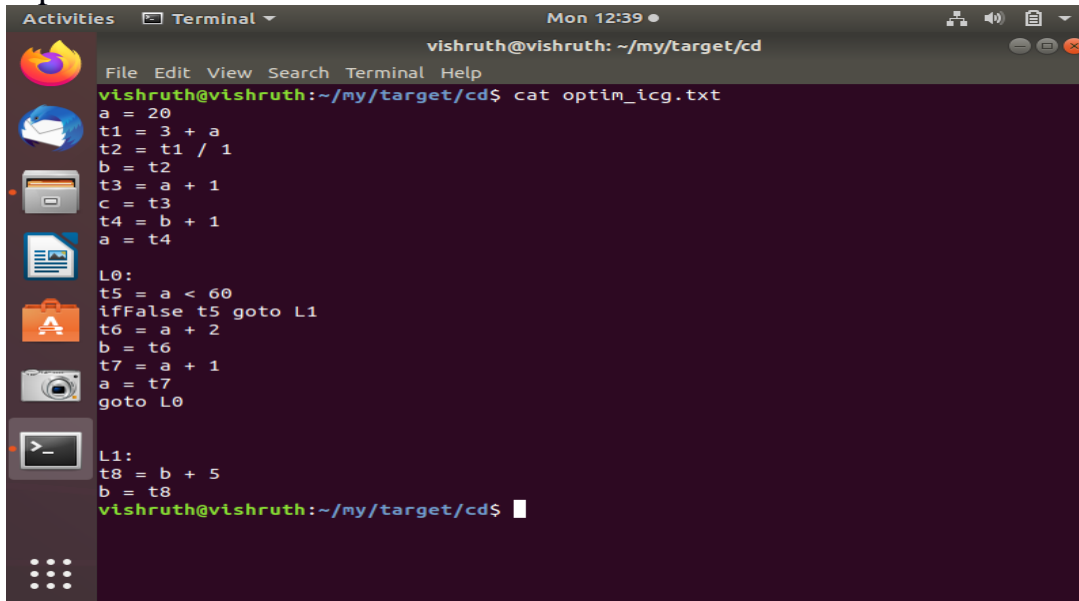
L1:
L2:
t5 = 3
ifFalse t5 goto L3
t6 = 6
a = t6

L3:
a = t6

L4:
t7 = 0
ifFalse t7 goto L5
t8 = 24
t9 = t8 * 9
b = t9
t10 = a + 1
```


6. Target Code Generation:

Input:



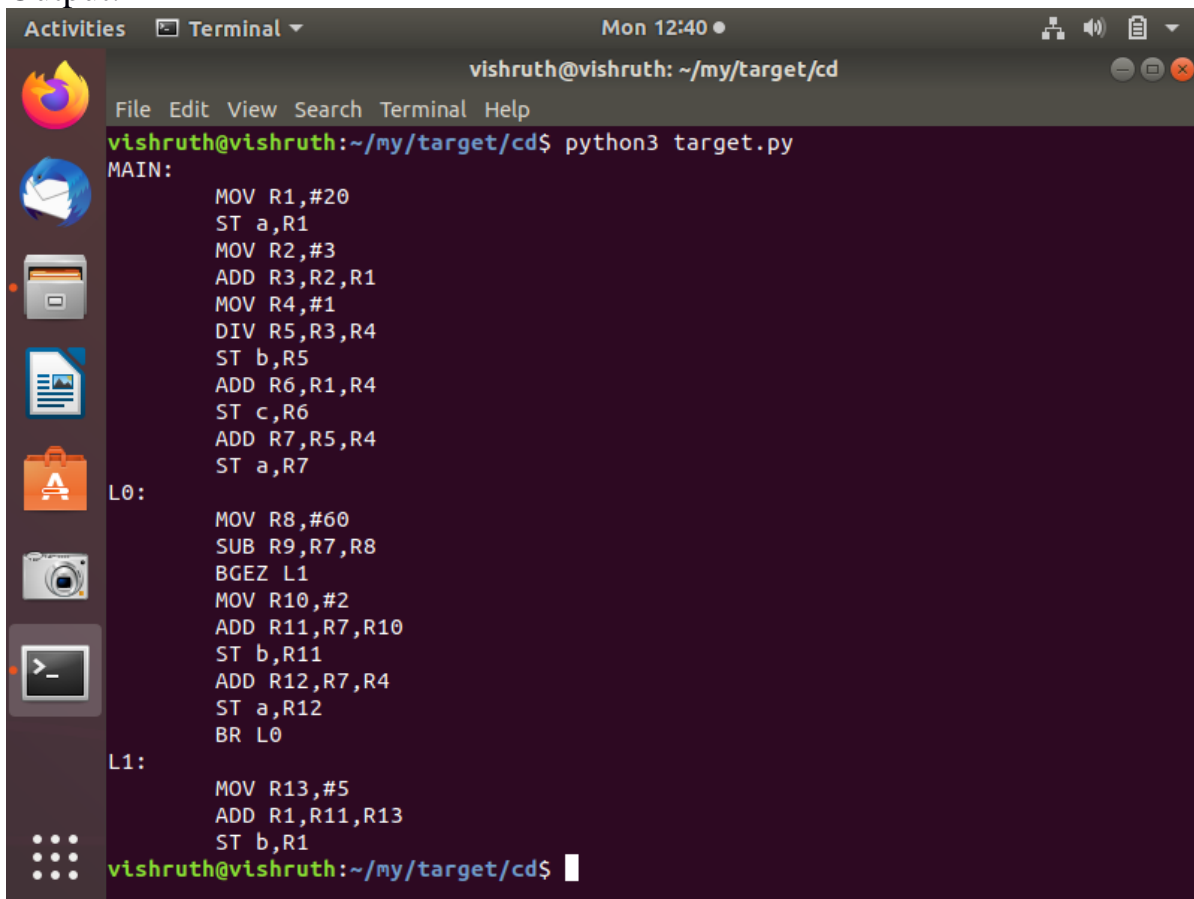
A terminal window titled 'vishruth@vishruth: ~/my/target/cd' showing the contents of a file named 'optim_icg.txt'. The code is a mix of high-level assignments and assembly-like control flow. The left sidebar shows various application icons.

```
vishruth@vishruth:~/my/target/cd$ cat optim_icg.txt
a = 20
t1 = 3 + a
t2 = t1 / 1
b = t2
t3 = a + 1
c = t3
t4 = b + 1
a = t4

L0:
t5 = a < 60
ifFalse t5 goto L1
t6 = a + 2
b = t6
t7 = a + 1
a = t7
goto L0

L1:
t8 = b + 5
b = t8
vishruth@vishruth:~/my/target/cd$
```

Output:



A terminal window titled 'vishruth@vishruth: ~/my/target/cd' showing the output of a Python script 'target.py'. The output is assembly code with labels 'MAIN:', 'L0:', and 'L1:'. The left sidebar shows various application icons.

```
vishruth@vishruth:~/my/target/cd$ python3 target.py
MAIN:
    MOV R1,#20
    ST a,R1
    MOV R2,#3
    ADD R3,R2,R1
    MOV R4,#1
    DIV R5,R3,R4
    ST b,R5
    ADD R6,R1,R4
    ST c,R6
    ADD R7,R5,R4
    ST a,R7

L0:
    MOV R8,#60
    SUB R9,R7,R8
    BGEZ L1
    MOV R10,#2
    ADD R11,R7,R10
    ST b,R11
    ADD R12,R7,R4
    ST a,R12
    BR L0

L1:
    MOV R13,#5
    ADD R1,R11,R13
    ST b,R1
vishruth@vishruth:~/my/target/cd$
```

Chapter 9- Conclusions:

- Thus, we have seen the design strategies and implementation of the different stages involved in building a mini C compiler and successfully built a working compiler that generates symbol table, abstract syntax tree, intermediate code, target code for a given C code as input.

Chapter 10- Further Enhancements:

- Future Enhancements that can be done are:
- Implement switch construct
- Implement if-else construct
- Implement do while construct
- Handle more data types and key words.

References / Bibliography:

- Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- Course material shared by department.
- <https://www.javatpoint.com/compiler-tutorial>
- <https://www.javatpoint.com/code-generation>

