



***Report on***

**IF-ELSE C- COMPILER**

*Submitted in partial fulfillment of the requirements for Sem VI*

***Compiler Design Laboratory***

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

***Submitted by:***

**Farheen Zehra  
S Akshatha  
Shivani Sweta S**

**PES2201800651  
PES2201800395  
PES2201800369**

***Under the guidance of***

**Swati Gambhire**

Professor at PES University

PES University, Bengaluru

**January – May 2021**

**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**

<b>Chapter No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1.</b>	<b>INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)</b>	<b>04</b>
<b>2.</b>	<b>ARCHITECTURE OF LANGUAGE:</b> <ul style="list-style-type: none"><li>• What all have you handled in terms of syntax and semantics for the chosen language.</li></ul>	<b>05</b>
<b>3.</b>	<b>LITERATURE SURVEY (if any paper referred or link used)</b>	<b>06</b>
<b>4.</b>	<b>CONTEXT FREE GRAMMAR (which you used to implement your project)</b>	<b>10</b>
<b>5.</b>	<b>DESIGN STRATEGY (used to implement the following)</b> <ul style="list-style-type: none"><li>• SYMBOL TABLE CREATION</li><li>• INTERMEDIATE CODE GENERATION</li><li>• CODE OPTIMIZATION</li><li>• ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).</li></ul>	<b>12</b>

6.	<b>IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):</b> <ul style="list-style-type: none"><li>● SYMBOL TABLE CREATION</li><li>● INTERMEDIATE CODE GENERATION</li><li>● CODE OPTIMIZATION</li><li>● ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).</li><li>● Provide instructions on how to build and run your program.</li></ul>	15
7.	<b>RESULTS AND possible shortcomings of your Mini-Compiler</b>	21
8.	<b>SNAPSHOTS (of different outputs)</b>	22
9.	<b>CONCLUSIONS</b>	37
10.	<b>FURTHER ENHANCEMENTS</b>	37
<b>REFERENCES/BIBLIOGRAPHY</b>		38

**GITHUB LINK FOR THE PROJECT:** <https://github.com/farheenzehra29/IF-ELSE-C-COMPILER>

# INTRODUCTION

Simulation of front end and back end phase of C Compiler involving if-else construct using lex and yacc tools. Nested IFs are also taken into account.

The **lexical analyser** generates the tokens in lexer.l using regular expressions. The **syntax analyser** (yacc) creates a grammar for the entire C code that has IF-ELSE construct. Error handling i.e. Panic mode error recovery and phrase level phase error recovery is implemented.

**Semantic Analyzer** verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. Type compatibility and errors in expressions are handled.

The **intermediate code** representation follows the three address code format. Code in quadruple format is implemented.

**Code Optimisation** techniques like constant folding, constant propagation, dead code elimination and strength reduction is taken into account and the optimised code in quadruple format is shown.

The **input** could be any valid C Program with conditions constructs. The **output** would be the expected output from “gcc” when run on a standard Unix shell along with the details of each phase of the compiler.

# ARCHITECTURE OF THE LANGUAGE

In terms of the **syntax**, we handled the following :

We have handled the syntactic phase errors such as Unbalanced parentheses , When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semicolon . We used **Panic mode recovery** to handle such errors.

When a parser encounters an error, it tries to take corrective measures so that the rest of the inputs of the statement allow the parser to parse ahead. In our designed compiler, when we replace a semicolon(;) with a comma(,) , the parser still continues to parse and it will generate the three address code and the optimised code. This is possible because of the **phrase level error recovery** strategy used in our design

In terms of the **semantics**, we handled the following :

**Type incompatibility:** If data types of two operands are incompatible then, automatic type conversion is done by the compiler.

E.x. float a1; a1="aa"; The type incompatibility is handled and the code proceeds with its normal execution.

**Errors in expressions:** Errors in evaluating expressions of the wrong type are also handled.

E.x.int a; float a1;

a1="aa";

a=a1-5; The compiler does not stop when it encounters errors. It continues with the normal execution of the rest of the program.

## LITERATURE SURVEY

### Topic 1:

**A Study on Language Processing Policies Compiler design.**

### Reference:

[https://www.researchgate.net/publication/338175915\\_A\\_Study\\_on\\_Language\\_Processing\\_Policies\\_in\\_Compiler\\_Design](https://www.researchgate.net/publication/338175915_A_Study_on_Language_Processing_Policies_in_Compiler_Design)

### Review:

Computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer.

This translation can be automated, which implies that it can be formulated as a program itself. The translation program is called a compiler, and the text to be translated is called source code.

A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provides a basis for communication interface between a user and a processor in a significant amount of time.

The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. We shall be concerned with the engineering of compilers covering organization, algorithms, data structures and user interfaces. Programming languages are tools used to construct formal descriptions of finite computation. Each computation consists of operations that transform a given initial state into some final state.

A compiler compiles a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed.

One advantage of using a high-level language can be the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

## 2. Code optimization techniques.

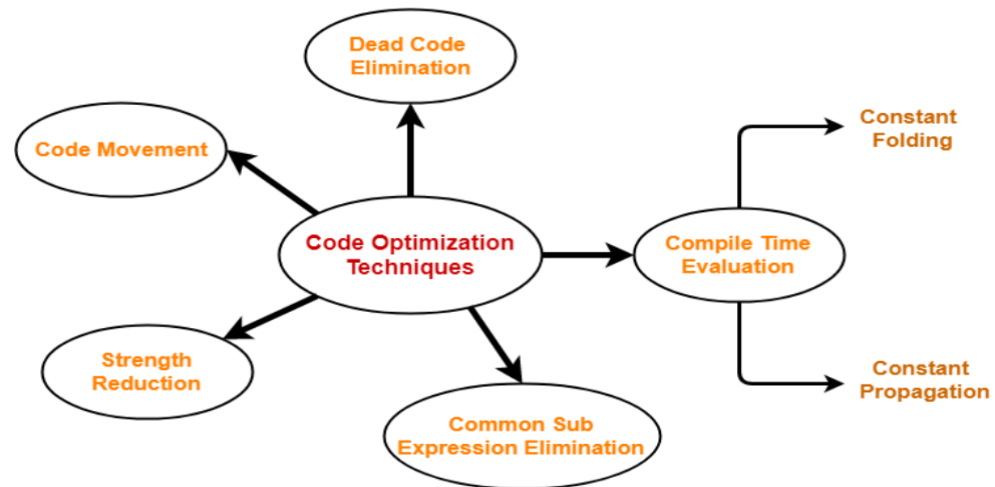
**Reference:** <https://www.gatevidyalay.com/code-optimization-techniques/#:~:text=In%20Compiler%20design%2C%20Code%20Optimization,Dead%20code%20elimination%2C%20Strength%20reduction.>

### **Review:**

The process of code optimization involves-

- Eliminating the unwanted code lines
- Rearranging the statements of the code.





Advantages: The optimized code has the following advantages-

- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.

# CONTEXT FREE GRAMMAR

```
CFG_FINAL - Notepad
File Edit Format View Help
stmt → selectStmt
selectStmt → if simpleExp then stmt | if simpleExp then stmt else stmt
simpleExp → andExp simpleExp'
simpleExp' → logop andExp simpleExp' | epsilon
andExp → unaryRelExp andExp'
andExp' → logop unaryRelExp andExp' | epsilon
unaryRelExp → logop unaryRelExp | relExp
relExp → minmaxExp relop minmaxExp | minmaxExp
relop → <= | < | > | >= | == | !=
logop → && | ! | ||
minmaxExp → sumExp minmaxExp'
minmaxExp' → minmaxop sumExp minmaxExp' | epsilon
minmaxop → opnd > opnd | opnd < opnd
sumExp → mulExp sumExp'
sumExp' → sumop mulExp sumExp' | epsilon
sumop → + | -
mulExp → mulExp | unaryExp mulExp'
mulExp' → mulop unaryExp mulExp' | epsilon
mulop → * | / | %
unaryExp → unaryop unaryExp | factor
unaryop → - | * | ?
factor → immutable | mutable
mutable → ID | ID [ exp ]
immutable → ( exp ) | call | constant
call → ID ( args )
args → argList | epsilon
argList → argList , exp | exp argList'
argList' → , exp argList' | epsilon
constant → NUMCONST | CHARCONST | STRINGCONST | true | false
```

## SOME TOKEN DEFINITIONS

letter = a | ... | z | A | ... | Z

digit = 0 | ... | 9

ID = letter letdig\*

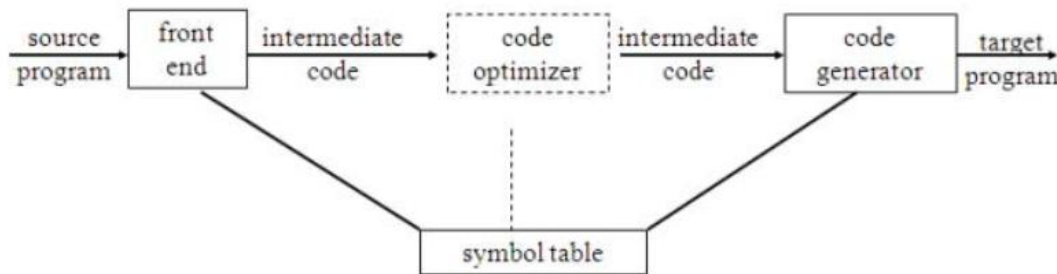
NUMCONST = digit+

opnd = letter+ | digit+

CHARCONST = is a representation for a single character by placing that character in single quotes. A backslash is an escape character. Any character preceded by a backslash is interpreted as that character. For example \x is the letter x, \' is a single quote, \\ is a single backslash. There are only two exceptions to this rule: \n is a newline character and \0 is the null character.

STRINGCONST = any series of zero or more characters enclosed by double quotes. A backslash is an escape character. Any character preceded by a backslash is interpreted as that character without meaning to the string syntax. For example \x is the letter x, \" is a double quote, \' is a single quote, \\ is a single backslash. There are only two exceptions to this rule: \n is a newline character and \0 is the null character. The string constant can be an empty string: a string of length 0. All string constants are terminated by the first unescaped double quote. String constants must be entirely contained on a single line, that is, they contain no unescaped newlines

# DESIGN STRATEGY



## SYMBOL TABLE CREATION

Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. It is used by both the analysis and synthesis phases.

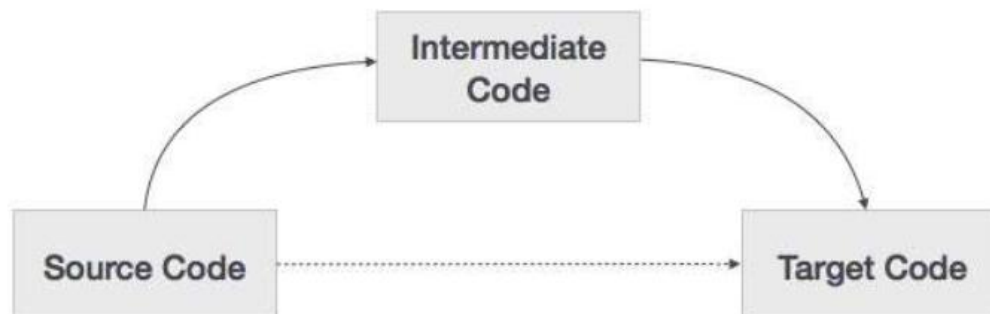
Design followed for the creation of symbol table is as follows :

A structure named “symbol\_table” has been created which accepts the name of the token (in case of keywords, the name is the "keyword" itself), the symbol, type of token (in case of keywords, the type is the name of the keyword), line number where token is declared (default

0),line number where token is first used (default 0),size of the token, its scope, checking for arrays etc.

The functions like push, pop are used to enter and pop out the values from the symbol table. st is the variable used to access the symbol table.

### INTERMEDIATE CODE GENERATION



For each new machine, a full native compiler is required if the translation of the source language to its target machine language is done without the option for generating intermediate code.

The Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion the same for all the compilers.

## **CODE OPTIMIZATION**

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

Optimisation techniques used in our compiler are as follows :

1. Constant folding
2. Dead code elimination
3. Constant propagation
4. Strength reduction

## **ERROR HANDLING**

In error handling, the errors like the missing parentheses, type incompatibility , incorrect token are being handled using the techniques like panic mode error recovery which means that When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semicolon and phrase level recovery which states that the routines may change, insert, or delete symbols on the input and handle the error.

# IMPLEMENTATION DETAILS

## SYMBOL TABLE CREATION

We are using a linked list of structures to implement our Symbol Table. It is created on the Heap and is called to a display function just before the compilation ends. The display function outputs a formatted symbol table to STDOUT.

A node of the symbol table has the following structure.

1. Line (lno)
2. Name
3. Scope (-1 for Global, 0 for Main and >0 for extraneous scopes)
4. Value
5. ID
6. Data Type

which is displayed in a tabular form during the end of the program to represent the symbol table that has been generated during the first phase. We store the entire table in a list data structure using struct.

## **Abstract Syntax Tree**

The syntax tree is generated entirely using Yacc. We create an Abstract Syntax Tree data structure to store the Syntax Tree, which is essentially something that mimics the way Yacc parses the grammar.

The treeNode has:

1. left child
2. right child
3. value
4. dataType
5. token

Every production is evaluated to the tree. So, as soon as we encounter a non-terminal, we process the expression through the Yacc grammar to generate nodes in the AST. It contains all the metadata as enumerated above. We basically use Dollar variables (\$) in Yacc that is used to store the various tokens and expressions that we parse along the way.

To print (export) the syntax tree, we essentially print all the above information using the showTrunks() and use indentation to mimic a “tree” visualization. The showTrunks() is a helper function to print branches of the binary tree.



The print function uses a structure that contains the follows:

1. Previous node
2. nodeType
3. string
4. value
5. dataType

Upon compilation, the tree is then printed out, as a graphically accurate representation of a tree.

### **INTERMEDIATE CODE GENERATION**

Intermediate Code Generation is implemented using various functions and data structures that are used to generate and store the Intermediate Codes. We have a list of structures of type Quadruple to store the Quadruples generated by the compiler.

The quad structure contains the follows:

- 1.Operator
- 2.Argument 1
- 3.Argument 2
4. Result

## 5.Scope

## 6.Mark

Several functions are used to accomplish the required processing. The AddQuadruple() function is used to fill in the quadruple structure for intermediate code. We then use a printICG function that neatly formats the Intermediate Code and prints it onto STDOUT.

### CODE OPTIMIZATION

**Constant folding** is done by using the checkVal function. It checks if the value is present in the argument attribute. If it is present, the expression is evaluated based on the operands and operators.

**Constant propagation** uses the QUAD structure to get the arguments and the checkVal function to check if values are assigned to the argument variables. If yes, the values are propagated (substituted). These values are added to the optimised QUAD structure.

**Dead code elimination** checks if the result field in the argument has a return value. Any code present after that will be treated as dead and will not be added to the optimised structure.

**Strength reduction** is implemented by taking into consideration the '\*' and '+' operator. The expensive multiplication operation is replaced with addition using the addVal Function.

## **ERROR HANDLING**

In our compiler, we have implemented the error recovery for unbalanced parentheses by making use of two variables namely

1. opening\_brackets
2. closing\_brackets

Each time when the opening bracket is encountered, the count variable is increased. For each closing bracket encountered, the count variable is decreased. If the value of count is 0, it means that the number of opening and closing parentheses is equal and there is no error. If the value of count is less than or greater than zero, it means that there are an unequal number of parentheses. However, we have implemented a panic mode error recovery strategy to handle the same.

We have implemented the panic mode error recovery even for handling errors which require successive characters from the input to be removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon (;)

The information about the data types is maintained and is used to check for type compatibility. The type of the token is obtained via the symbol table entry. The type checking also depends on the type expression of the language. The check\_type() function is used to check for the data type and value associated with it.

### **Instructions to build and run the program:**

To Run Program:

1. Place all files in the same directory.
2. `flex -l proj.l`
3. `yacc -vd icg.y`
4. `gcc lex.yy.c y.tab.c -lm -ll`
5. `./a.out`

Here `proj.l` is the lexical analyzer, `yacc -vd icg.y` runs `icg.y` written in YACC and `-d` creates `y.tab.h` and `-v` creates `y.output` (debugger for parser).

The `gcc` is used for the compilation of the C code. The executable file(`./a.out`) is run to get the desired output.

# **RESULTS AND possible shortcomings of your Mini-Compiler**

## **RESULTS:**

Our Compiler has been able to compile and generate code for the sample Input files pretty accurately. It can detect a plethora of errors and satisfactorily compile and produce optimal code.

## **POSSIBLE SHORTCOMINGS:**

The compiler we built is a if-else C compiler which is a part of C compiler. We can implement various other functionalities of C compiler. . The generated code may be a bit buffed up compared to a highly optimized version of the same, generated by an Official C Compiler.

# SNAPSHOTS

## Build step:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal

```
[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ flex -l proj.l
[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ yacc -vd icg.y
[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ gcc lex.yy.c y.tab.c -lm -ll
proj.l:101:1: warning: return type defaults to 'int' [-Wimplicit-int]
{
^
y.tab.c: In function 'yyparse':
y.tab.c:1571:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
    yychar = yylex ();
                ^
y.tab.c:3177:7: warning: implicit declaration of function 'yyerror' [-Wimplicit-function-declaration]
    yyerror (YY_("syntax error"));
            ^
icg.y: At top level:
icg.y:2287:1: warning: return type defaults to 'int' [-Wimplicit-int]
    yyerror(s)
    ^
[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ ./a.out
```

## Syntax Tree and Symbol Table: (Review1)

```
#####
                        Syntax Tree in Inorder traversal
#####
start , definition , statement , definition , statement , definition , statement , definition , statement , c , > , 3 ,
IF , a , = , 2 , + , 3 , statement , definition , statement , definition , statement , z , = , 1 , statement , 2 ,
> , 10 , IF , c , = , c , + , 2 ,

Parsing is Successful
#####
                        Symbol table
#####

symbol   type    identify    line number
-----
#include<stdio.h>      Header    0
#include<stdlib.h>     Header    1
int      N/A      KEYWORD      3
main     int      IDENTIFIER   3
(        N/A      Punctuation  3
argv     int      IDENTIFIER   3
,        N/A      Punctuation  3
char     N/A      KEYWORD      3
argc     char*    IDENTIFIER   3
]        N/A      Punctuation  3
)        N/A      Punctuation  3
{        N/A      Punctuation  4
;        N/A      Punctuation  5
d        int      variable     5
x        int      variable     6
a        int      variable     7
float    N/A      KEYWORD      8
c        float    variable     8
if       N/A      KEYWORD      10
```

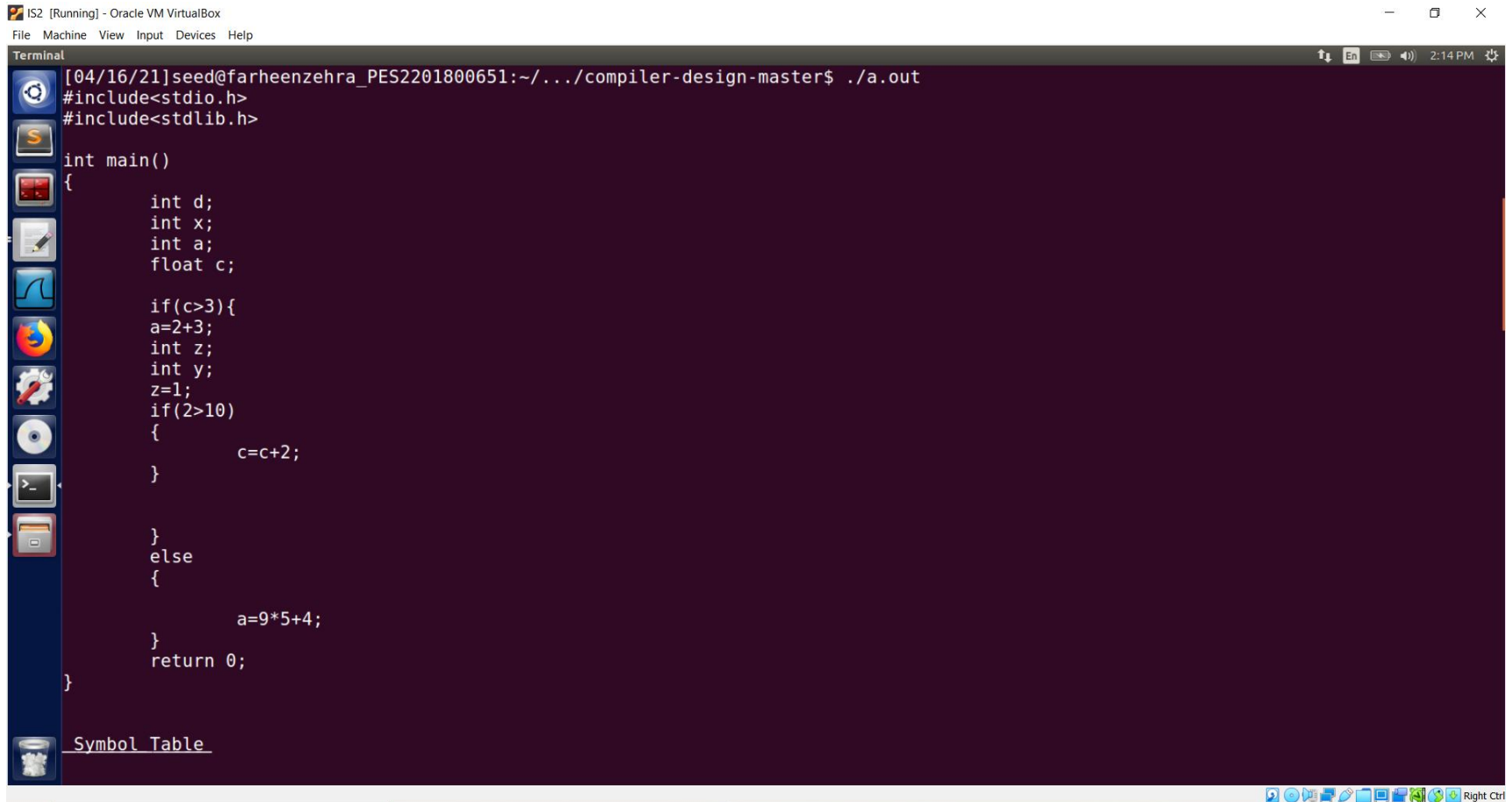
## Symbol Table(Contd.): (Review1)

```
Parsing is Successful
#####
                        Symbol table
#####

symbol   type   identify   line number
-----
#include<stdio.h>      Header  0
#include<stdlib.h>     Header  1
int      N/A     KEYWORD     3
main     int     IDENTIFIER  3
(        N/A     Punctuation 3
argv     int     IDENTIFIER  3
,        N/A     Punctuation 3
char     N/A     KEYWORD     3
argc     char*   IDENTIFIER  3
]        N/A     Punctuation 3
)        N/A     Punctuation 3
{        N/A     Punctuation 4
;        N/A     Punctuation 5
d        int     variable   5
x        int     variable   6
a        int     variable   7
float    N/A     KEYWORD     8
c        float   variable   8
if       N/A     KEYWORD     10
>        float   IDENTIFIER  10
3        int     NUMBER      10
else     N/A     KEYWORD     22
0        int     NUMBER      27
}        N/A     Punctuation 28
[02/28/21]seed@farheenzehra_PES2201800651:~/cd$
```



## Final Output:



The screenshot shows a terminal window titled "IS2 [Running] - Oracle VM VirtualBox". The terminal displays the execution of a C program. The code includes `<stdio.h>` and `<stdlib.h>`, and defines a `main` function. Inside `main`, it declares variables `d`, `x`, `a`, and `c`. It then enters a conditional block: if `c > 3`, it sets `a = 2 + 3`, `z = 1`, and enters another if-block where `2 > 10` is false, so `c = c + 2` is not executed. Otherwise, it sets `a = 9 * 5 + 4` and returns 0. A "Symbol Table" window is visible at the bottom left of the terminal.

```
[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ ./a.out
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int d;
    int x;
    int a;
    float c;

    if(c>3){
        a=2+3;
        int z;
        int y;
        z=1;
        if(2>10)
        {
            c=c+2;
        }
    }
    else
    {
        a=9*5+4;
    }
    return 0;
}
```

Symbol Table

## Final Symbol Table:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

2:15 PM

Symbol Table

Symbol	Name	Array	Type	Scope	Size	Line Number	Lines Used	Value
-								
keyword	char	0	NIL	0	0	1	[ 1 ]	0
keyword	int	0	NIL	0	0	1	[ 1 ]	0
keyword	float	0	NIL	0	0	1	[ 1 ]	0
keyword	short	0	NIL	0	0	1	[ 1 ]	0
keyword	long	0	NIL	0	0	1	[ 1 ]	0
keyword	double	0	NIL	0	0	1	[ 1 ]	0
keyword	void	0	NIL	0	0	1	[ 1 ]	0
keyword	if	0	NIL	0	0	1	[ 1 ]	0
keyword	else	0	NIL	0	0	1	[ 1 ]	0
keyword	while	0	NIL	0	0	1	[ 1 ]	0
keyword	do	0	NIL	0	0	1	[ 1 ]	0
keyword	continue	0	NIL	0	0	1	[ 1 ]	0
keyword	break	0	NIL	0	0	1	[ 1 ]	0
keyword	return	0	NIL	0	0	1	[ 1 ]	0
function	printf	0	NIL	0	0	1	[ 1 ]	0

Right Ctrl

## Symbol Table (Contd.):

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

2:16 PM

function	printf	0	NIL	0	0	1	[ 1 ]	0
identifier	d	0	int	1	4	6	[ 6 ]	0
identifier	x	0	int	1	4	7	[ 7 ]	0
identifier	a	0	int	1	4	8	[ 8 ]	0
identifier	c	0	float	1	4	9	[ 9 ]	0
temporary	t0	0	int	1	4	11	[ 11 ]	0
temporary	t1	0	int	2	4	12	[ 12 ]	0
identifier	z	0	int	2	4	13	[ 13 ]	0
identifier	y	0	int	2	4	14	[ 14 ]	0
temporary	t2	0	int	2	4	16	[ 16 ]	0
temporary	t3	0	int	3	4	18	[ 18 ]	0
temporary	t4	0	int	4	4	25	[ 25 ]	0
temporary	t5	0	int	4	4	25	[ 25 ]	0
function	main	0	int	0	4	28	[ 28 ]	0

Intermediate Code

## Intermediate code:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

2:16 PM

### Intermediate Code

Three Address Code Quadruple

pos	op	arg1	arg2	result	scope
0	>	c	3	t0	1
1	if	t0		L0	1
2	goto			L1	1
3	Label			L0	1
4	+	2	3	t1	2
5	=	t1		a	2
6	=	1		z	2
7	>	2	10	t2	2
8	if	t2		L2	2
9	goto			L3	2
10	Label			L2	2
11	+	c	2	t3	3
12	=	t3		c	3
13	Label			L3	2
14	goto			L4	1
15	Label			L1	1
16	*	9	5	t4	4
17	+	t4	4	t5	4
18	=	t5		a	4
19	Label			L4	1

After optimization:

### Intermediate Code

## Optimised Code:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

14	goto			L4	1
15	Label			L1	1
16	*	9	5	t4	4
17	+	t4	4	t5	4
18	=	t5		a	4
19	Label			L4	1

After optimization:

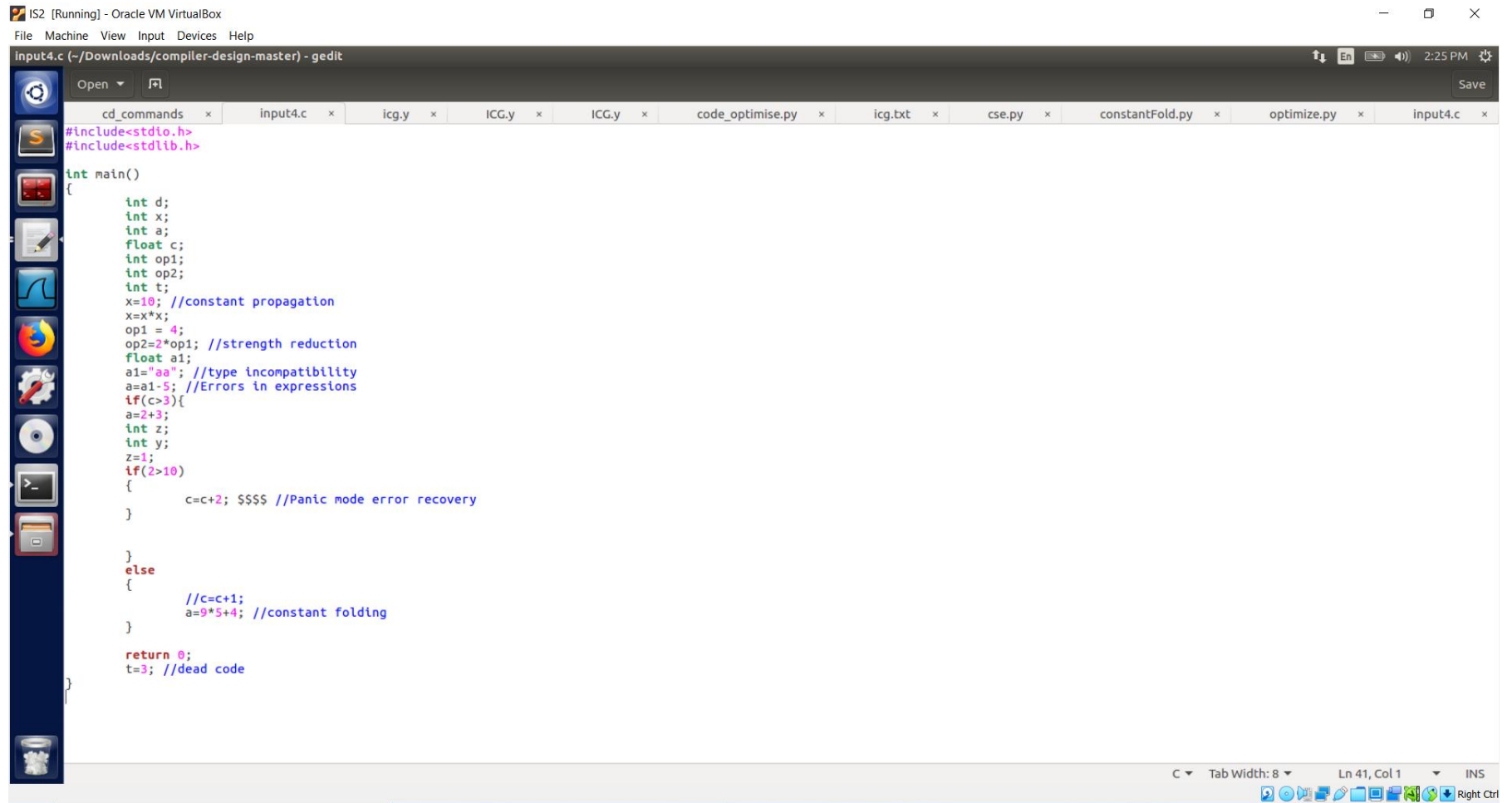
Intermediate Code

Three Address Code Quadruple

pos	op	arg1	arg2	result	scope
1	if	1		L0	1
2	goto			L1	1
3	Label			L0	1
5	=	5		a	2
6	=	1		z	2
8	if	0		L2	2
9	goto			L3	2
10	Label			L2	2
12	=	2		c	3
13	Label			L3	2
14	goto			L4	1
15	Label			L1	1
18	=	49		a	4
19	Label			L4	1

[04/16/21]seed@farheenzehra\_PES2201800651:~/.../compiler-design-master\$

## input file with errors:



The screenshot shows a VirtualBox window titled "IS2 [Running] - Oracle VM VirtualBox". Inside, a gedit editor is open with the file "input4.c" located at "~/Downloads/compiler-design-master". The editor has several tabs open: "cd\_commands", "input4.c", "icg.y", "ICG.y", "code\_optimise.py", "icg.txt", "cse.py", "constantFold.py", "optimize.py", and "input4.c". The C code in "input4.c" contains several errors as indicated by comments and syntax:

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

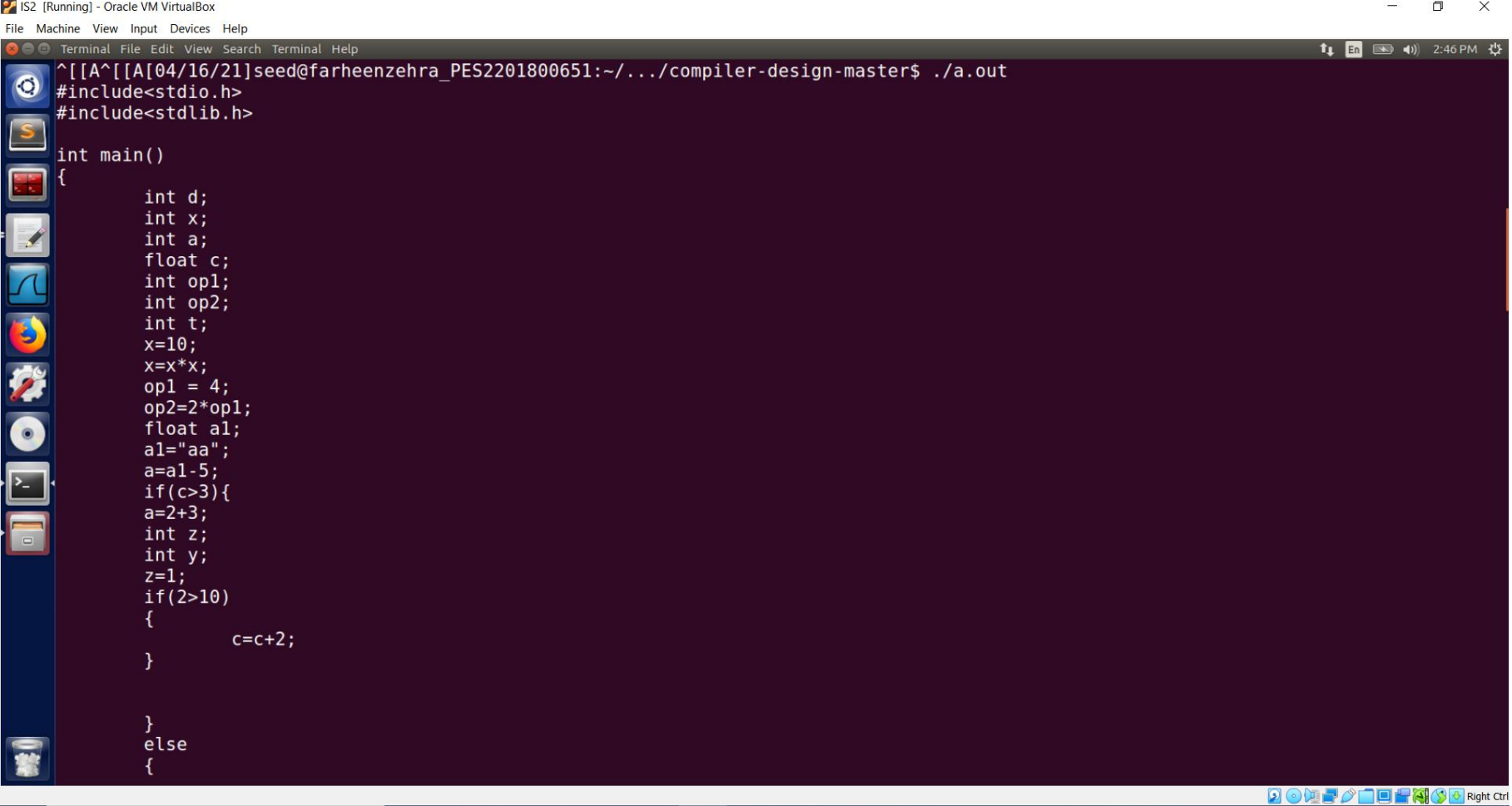
int main()
{
    int d;
    int x;
    int a;
    float c;
    int op1;
    int op2;
    int t;
    x=10; //constant propagation
    x=x*x;
    op1 = 4;
    op2=2*op1; //strength reduction
    float a1;
    a1="aa"; //type incompatibility
    a=a1-5; //Errors in expressions
    if(c>3){
        a=2+3;
        int z;
        int y;
        z=1;
        if(2>10)
        {
            c=c+2; $$$$ //Panic mode error recovery
        }
    }
    else
    {
        //c=c+1;
        a=9*5+4; //constant folding
    }

    return 0;
    t=3; //dead code
}
```

The status bar at the bottom indicates "C", "Tab Width: 8", "Ln 41, Col 1", and "INS".

## Output for the files with errors:

c=c+2 without \$ in the output demonstrates panic mode recovery



The screenshot shows a terminal window titled "IS2 [Running] - Oracle VM VirtualBox". The terminal displays the output of a C program. The code includes `#include<stdio.h>` and `#include<stdlib.h>`. The `main` function declares variables `d`, `x`, `a`, `c`, `op1`, `op2`, and `t`. It initializes `x` to 10, calculates `op1 = 4` and `op2 = 2 * op1`, and declares `float a1` with value "aa". It then enters a loop where it checks `if(c>3)`. Inside this loop, it calculates `a = a1 - 5`, `a = 2 + 3`, declares `int z` and `int y`, sets `z = 1`, and checks `if(2>10)`. Inside this nested loop, it prints `c=c+2;` without a dollar sign. The loop ends with `} else {`. The terminal output shows the execution of the program, with the first line of output being `^[[A^[[A[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ ./a.out`.

```
^[[A^[[A[04/16/21]seed@farheenzehra_PES2201800651:~/.../compiler-design-master$ ./a.out
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int d;
    int x;
    int a;
    float c;
    int op1;
    int op2;
    int t;
    x=10;
    x=x*x;
    op1 = 4;
    op2=2*op1;
    float a1;
    a1="aa";
    a=a1-5;
    if(c>3){
        a=2+3;
        int z;
        int y;
        z=1;
        if(2>10)
        {
            c=c+2;
        }
    }
    else
    {
```

## Symbol Table:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

2:46 PM

<



## Symbol Table(Contd.):

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

function	printf	0	NIL	0	0	1	[ 1 ]	0
identifier	d	0	int	1	4	6	[ 6 ]	0
identifier	x	0	int	1	4	7	[ 7 ]	0
identifier	a	0	int	1	4	8	[ 8 ]	0
identifier	c	0	float	1	4	9	[ 9 ]	0
identifier	op1	0	int	1	4	10	[ 10 ]	0
identifier	op2	0	int	1	4	11	[ 11 ]	0
identifier	t	0	int	1	4	12	[ 12 ]	0
temporary	t0	0	int	1	4	13	[ 13 ]	0
temporary	t1	0	int	1	4	15	[ 15 ]	0
identifier	a1	0	float	1	4	15	[ 15 ]	0
temporary	t2	0	int	1	4	16	[ 16 ]	0
temporary	t3	0	int	1	4	16	[ 16 ]	0
temporary	t4	0	int	2	4	17	[ 17 ]	0
identifier	z	0	int	2	4	18	[ 18 ]	0
identifier	y	0	int	2	4	19	[ 19 ]	0
temporary	t5	0	int	2	4	21	[ 21 ]	0

Right Ctrl

## Intermediate Code:

IS2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal File Edit View Search Terminal Help

Three Address Code Quadruple

pos	op	arg1	arg2	result	scope
0	=	10		x	1
1	*	x	x	t0	1
2	=	t0		x	1
3	=	4		op1	1
4	*	2	op1	t1	1
5	=	t1		op2	1
6	=	"aa"		a1	1
7	-	a1	5	t2	1
8	=	t2		a	1
9	>	c	3	t3	1
10	if	t3		L0	1
11	goto			L1	1
12	Label			L0	1
13	+	2	3	t4	2
14	=	t4		a	2
15	=	1		z	2
16	>	2	10	t5	2
17	if	t5		L2	2
18	goto			L3	2
19	Label			L2	2
20	+	c	2	t6	3
21	=	t6		c	3
22	Label			L3	2
23	goto			L4	1
24	Label			L1	1
25	*	9	5	t7	4
26	+	t7	4	t8	4
27	=	t8		a	4
28	Label			L4	1
29	=	3		t	1

Right Ctrl

### **Optimised Code:**

Statements 25 ,26 ,27 in three address code are optimised to statement 27 in optimised three address code because of constant folding optimisation technique.

Statements 1 and 2 are optimised to statement 2 after optimisation because of constant propagation optimisation technique.

Statement 29 (t=3) in the actual three address code is eliminated and is not found in the optimised code because of the dead code elimination technique.

Statements 3, 4 and 5 are converted to 3 and 5 after optimization because of the strength reduction technique.

## Optimised Code Output:

```
IS2 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal File Edit View Search Terminal Help
After optimization:
Intermediate Code
Three Address Code Quadruple
```

pos	op	arg1	arg2	result	scope
0	=	10		x	1
2	=	100		x	1
3	=	4		op1	1
5	=	8		op2	1
6	=	"aa"		a1	1
8	=	5		a	1
10	if	1		L0	1
11	goto			L1	1
12	Label			L0	1
14	=	5		a	2
15	=	1		z	2
17	if	0		L2	2
18	goto			L3	2
19	Label			L2	2
21	=	2		c	3
22	Label			L3	2
23	goto			L4	1
24	Label			L1	1
27	=	49		a	4
28	Label			L4	1

```
[04/16/21]seed@farheenzehra_PES2201800651:~/../compiler-design-master$
```

## **CONCLUSION**

We can conclude that a satisfactorily accurate if-else C compiler can be built using Lex and Yacc for a number of different languages spreading across multiple genres. We can conclude that the various phases of a standard compiler can be built and implemented using these tools and by following all regulations, a standard compiler can be built for almost any language.

## **FURTHER ENHANCEMENTS**

We have implemented an if-else C compiler that can be further extended to other functionalities.

New languages which are more close to general languages are being invented. With the growth of technology ease of working is given priority. We have emerged from C , C++ to python ,ruby , etc.

which require less lines of code . There are other platforms such as Android Studio, Qt which provide easy GUI creation and use the popular languages Java and C++ respectively.

Our project can be extended to form a new language which is easy to learn, faster , has more inbuilt features and has many more qualities of a good programming language.

# REFERENCES

1. <https://norasandler.com/2017/11/29/Write-a-Compiler.html>
2. [https://www.researchgate.net/publication/338175915\\_A\\_Study\\_on\\_Language\\_Processing\\_Policies\\_in\\_Compiler\\_Design](https://www.researchgate.net/publication/338175915_A_Study_on_Language_Processing_Policies_in_Compiler_Design)
3. <https://www.gatevidyalay.com/code-optimization-techniques/#:~:text=In%20Compiler%20design%2C%20Code%20Optimization,Dead%20code%20elimination%2C%20Strength%20reduction.>
4. [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_intermediate\\_code\\_generations.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm)
5. [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_error\\_recovery.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_error_recovery.htm)
6. [https://www.vssut.ac.in/lecture\\_notes/lecture1422914957.pdf](https://www.vssut.ac.in/lecture_notes/lecture1422914957.pdf)