Report on

"MINI-JAVA COMPILER"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Nikhil J K	**PES2201800303**
PurushothamaReddy	**PES2201800473**
Nikhil Karle	**PES2201800642**

Under the guidance of

Prof. Swathi Gambhire
Assistant Professor
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

1. INTRODUCTION

Simple constructs from the language JAVA were implemented. The frontend of the compiler including Symbol table generation, Abstract Syntax tree construction, Intermediate Code generation and Code Optimization was implemented using lex and yacc.

The main functionality of the project is to generate an optimized intermediate code for the given Java source code.

This is done using the following steps:

- Generate symbol table after performing expression evaluation
- Generate Abstract Syntax Tree
- Generate 3 address code followed by corresponding quadruples
- Perform Code Optimization

Lex

Lex is a program that generates lexical analyzers. It is used with a YACC parser generator.The lexical analyzer is a program that transforms an input stream into a sequence of tokens.It reads the input stream and produces the source code as output through implementing the lexical analyzer .

Yacc

YACC stands for Yet Another Compiler Compiler.YACC provides a tool to produce a parser for a given grammar.YACC is a program designed to compile a LALR (1) grammar.It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.The input of YACC is the rule or grammar and the output is a C program.

Sample input

new 2 | inp_final.java

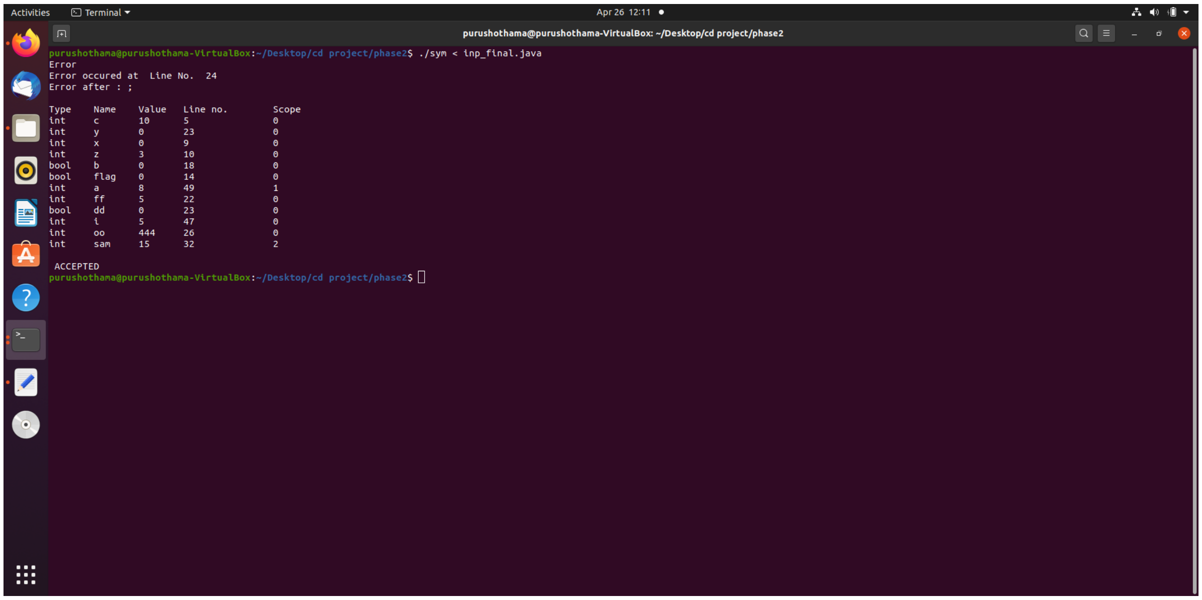
```java
package project;
import *;
public class myPrj{
    public static void main(String[] args){
int c=10;
    int y;
    int x;
    int z;
    y=x;
    z=3+y;


Boolean b=false;
Boolean flag;
int a =99;
int ff = 555;
Boolean dd  =a<7777;
a=a+b;
a= 6/3;
int y =99;
int c=5+10;
ff = 5+y;
dd  = y>=99;
zx=10;
int i =0;
int oo =444;
if(a==2)
{
    if(a>=4)
     {
    a=10;
    int sam=15;
     }
    else
    {
    a=4;
    }

```

```java
else
{
a=8;
}



for(i=5;a<10;)
{
a=8;
}




 }
}



```

Java source file   length : 434   lines : 59   Ln : 1   Col : 1   Sel : 0 | 0   Unix (LF)   UTF-8   INS

Type here to search   3:20 PM 26-Apr-21

Sample output



2. ARCHITECTURE OF LANGUAGE

We have implemented our Java compiler for constructs “for” and “ifelse”.

- Arithmetic expressions with +, -, *, /, ++, -- are handled.
- Boolean expressions with >,=,<=,== are handled.
- Error handling reports undeclared variables with line numbers.
- Error handling also reports syntax errors with line numbers.

3. LITERATURE SURVEY AND OTHER REFERENCES

Lex,yacc and its working:
https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

https://tldp.org/HOWTO/Lex-YACC-HOWTO-6.html
https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
Building a mini compiler:
https://www.tutorialspoint.com/compiler_design/index.html

4. CONTEXT FREE GRAMMAR

```
compilation_unit:pack
age_statement
import_statement
class_stmt
                ;
package_statement:  PACKAGE  IDENTIFIER  SEMC
                ;
import_statement:  IMPORT  IDENTIFIER
        DOT  MUL SEMC
                  |  IMPORT    class_name  SEMC
                    |  IMPORT  MUL  SEMC
                    |
                  ;class_name  SEMC
                    |  IMPORT  MUL  SEMC
                    |
                  ;


class_name:  IDENTIFIER;

class_stmt:  PUBLIC  CLASS  class_name
OF main_method  CF;
main_method:  PUBLIC  STATIC  VOID
MAIN  OC  STRING OS  FS  ARGS  CC  OF
sl  CF;
sl:  sl  s1|;

s1:  variable_declaration  SEMC                 | expression  SEMC

                |  if_stmt

                |for_stmt
                | SEMC;
```

```
variable_declaration:  dtypes
                        ;

dtypes:  INT  ids1

        |  FLOAT    ids2
        |  BOOLEAN    ids3;

ids1:  IDENTIFIER  EQ  arithm_e
        |  ids1 COMMA
IDENTIFIER
    |  IDENTIFIER
    |  IDENTIFIER  EQ  rel_e                    ;

 ids2:  IDENTIFIER
   EQ  arithm_e
  |  ids2  COMMA
     IDENTIFIER
    |  IDENTIFIER
    |  IDENTIFIER                    ;

expression:  arithm_e

|  rel_e

        ;

rel_e:  arithm_e  LT  arithm_e
        |  arithm_e  GT  arithm_e
        |  arithm_e  LE  arithm_e
        |  arithm_e  GE  arithm_e
        |  arithm_e  DEQ  arithm_e
        |  arithm_e  NE_OParithm_e
        |  IDENTIFIER  EQ  rel_e
        | TRUE1
        | FALSE1                                    ;

arithm_e:  arithm_e  MUL  arithm_e

        |  arithm_e  DIV  arithm_e
        |  arithm_e  ADD  arithm_e
```

```
            | arithm_e SUB arithm_e
            | IDENTIFIER
            | NUM
    | IDENTIFIER INC_OP
            | IDENTIFIER DEC_OP
            | INC_OP IDENTIFIER
            | DEC_OP IDENTIFIER
            | IDENTIFIER EQ arithm_e



    if_stmt: IF OC rel_e CC OF sl CF

    | IF OC rel_e CC OF
    sl CF else_if_blocks


    sl CF else_if_blocks ELSE OF sl CF

            ;


else_if_blocks :   ELSE else_if_block

          | else_if_blocks   ELSE
    else_if_block
                  ;
```

5. DESIGN DETAILS

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

*To store the names of all entities in a structured form at one place.

*To verify if a variable has been declared.

*To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

*To determine the scope of a name .

ABSTRACT SYNTAX TREE

Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

This tree is constructed as the input is parsed. Each node of this tree contains a pointer to left, a pointer to right and a member for a string.

Intermediate Code Generation (ICG)

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

CODE OPTIMIZATION

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.

- Optimization should itself be fast and should not delay the overall compiling process.

ERROR HANDLING

The tasks of the Error Handling process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle error. During this whole process processing time of the program should not be slow. An Error is the blank entries in the symbol table.

6.IMPLEMENTATION

SYMBOL TABLE

• A structure is maintained to keep track of the variables. The attributes are Line number, Variable name, Type, Value and Scope.

```
struct SymTable
{
char idName[50];
int value;
int type; //0-int , 1-float , 2-true , 3-false
int line_no;
int scope;
};
```

• As each line is parsed, the actions associated with the grammar rules are executed. Symbol table functions such as lookup, fill and update are called appropriately.

• $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

The following structure is created to store node details of abstract syntax trees.

```
typedef struct node

{
```

```
struct node *left;

struct node *right;

struct node *another;

char *token;

} node;
```

When every new token is encountered during parsing, the buildTree function takes in the value of the token, creates a node of the tree and attaches it to its head of the reduced production.

INTERMEDIATE CODE GENERATION

In order to generate 3 address code, an explicit stack was used. Whenever an operator,operand or a constant was encountered, it was pushed to the stack.

Whenever reduction occurred, the codegen( ) function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack. After That it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computationAfter generation of every intermediate code instruction,it needs to be stored to optimizecode.So a data structure called quadruples is used.

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generators assume an unlimited number of memory storage (register) to generate code.

For example: a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;

r2 = b + r1;

a = r2
```

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

The data structure used to represent three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result. The following structure stores the information of a quadruple.

```
struct OPT{

char op[10];

char arg1[10];

char arg2[10];

char result[10]; };
```

CODE OPTIMIZATION

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.

- Optimization should itself be fast and should not delay the overall compiling process.

Copy propagation:Copy propagation is a process of replacing the targets of direct Assignment with the value.

For example when a variable x is assigned with y,when we encounter variable y we substitute it with x.

Constant Folding: Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but they may also be variables whose values are known at compile time.

Constant Propagation:Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values.One strategy would be to use symbol table to get values of variables(which we have used)

Dead Code Elimination:

Code that is never executed or does useful computation is called dead code.

ERROR HANDLING Basic syntax and semantic error are handled.Syntax error handling is done byusing yyerrok() and yyclearin() functions provided by yacc tool,which uses panic error recovery method.

Instruction to Build and Run:

Symbol Table:

1)Run yacc file using “yacc -vd codeSymboleTable.y”

2)Next Run lex file using “lex code.l”

3)Run the generate y.tab.c using “gcc y.tab.c -o sym -ll”

4)Finally to run output file using “./sym < inp_final.java”

Abstract Syntax Tree:

1)Run yacc file using “yacc -vd codeAST.y”

2)Next Run lex file using “lex code.l”

3)Run the generate y.tab.c using “gcc y.tab.c -o ast -ll”

4)Finally to run output file using “./ast < inp_final.java”

Intermediate Code:

1)Run yacc file using “yacc -vd codeinter.y”

2)Next Run lex file using “lex code.l”

3)Run the generate y.tab.c using “gcc y.tab.c -o inter -ll”

4)Finally to run output file using “./inter < inp_final.java”

Code Optimization:

1)gcc codeoptimization.c

2)./a.out

7. RESULTS AND POSSIBLE SHORTCOMINGS

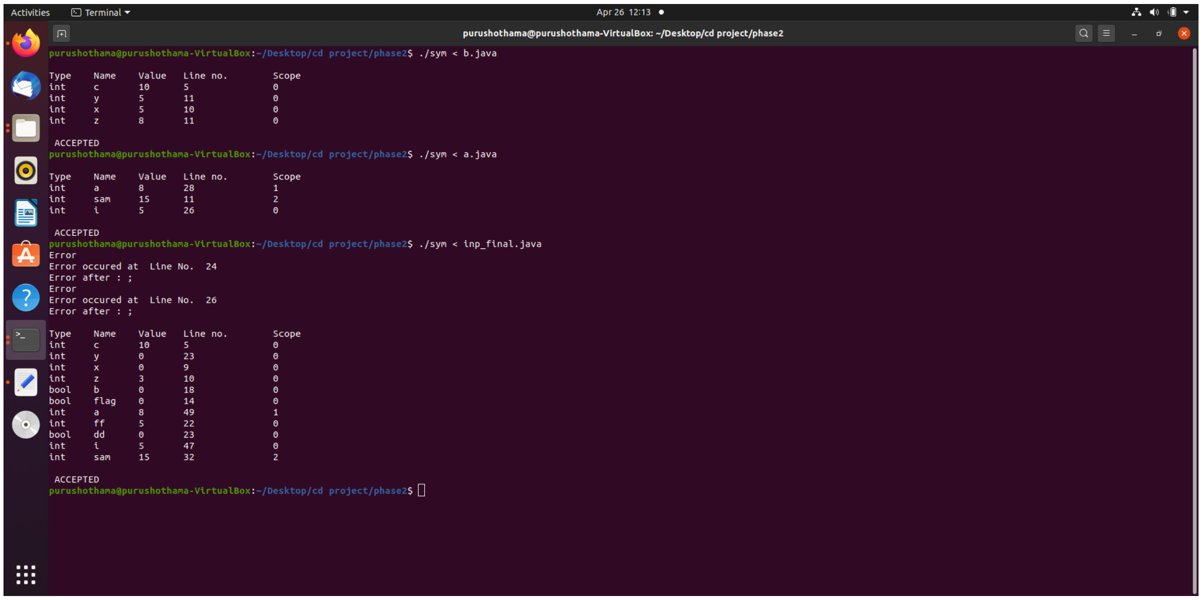
Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an intermediate code, given a JAVA code as input.

There are a few shortcomings with respect to our implementation. The symbol table is built only for variables and doesn’t include other tokens like keywords, operators, etc.

The code optimiser doesn’t work well when propagating constants across branches and that needs to be rectified.

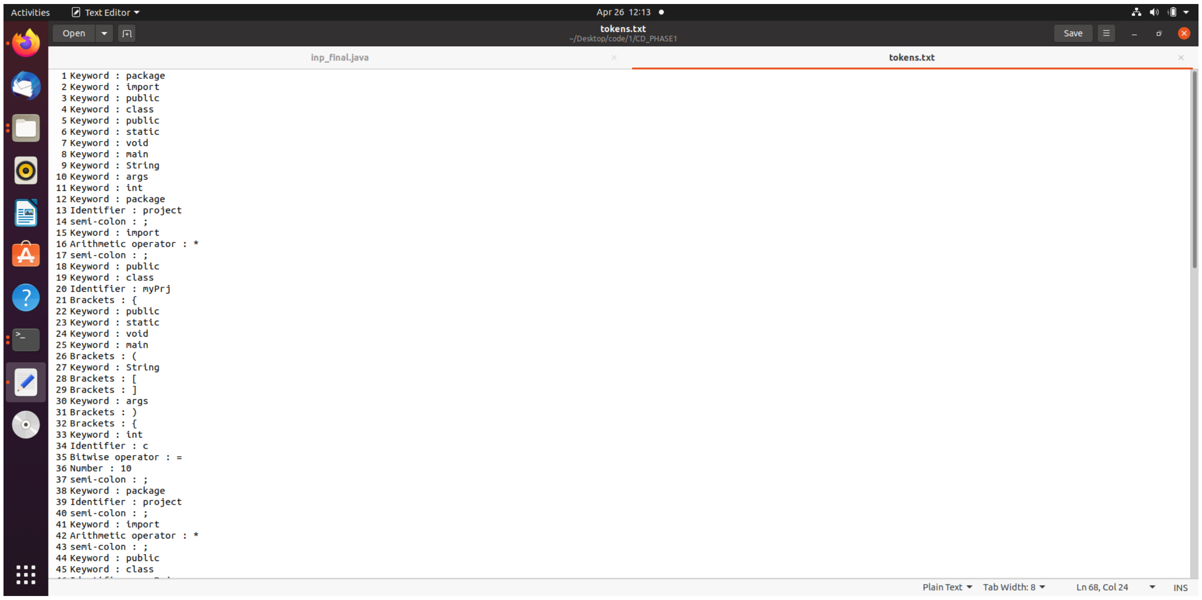
8.SNAPSHOTS

SYMBOL TABLE

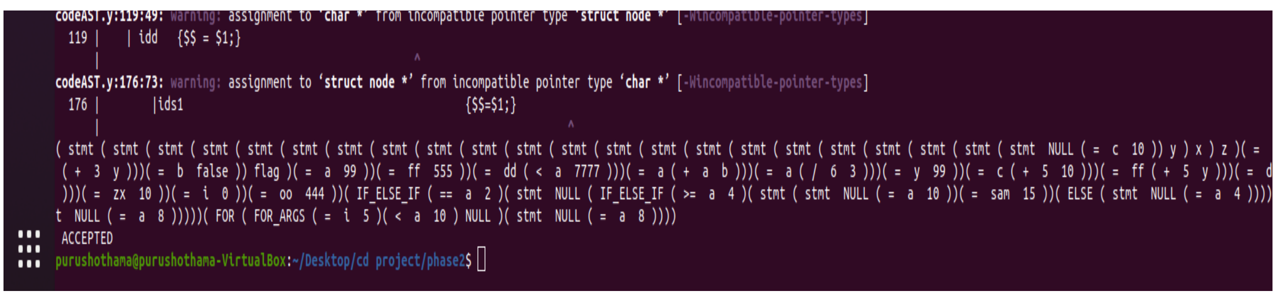


This ss shows the terminal output for the symbol table fo three test case files,it is implemented for keywords,operators and boolean values.Error handling is implemented for syntax and lexical errors.

These are the tokens generated for the symbol table.



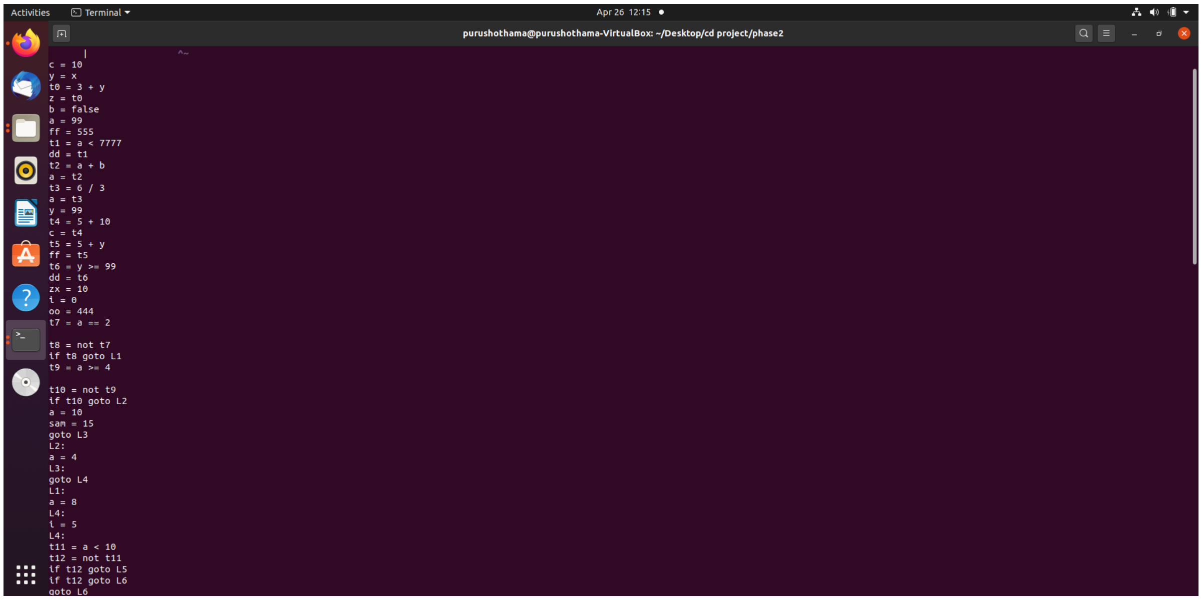
ABSTRACT SYNTAX TREE



INTERMEDIATE CODE GENERATION

The intermediate code generated in the 3AC is converted into quadruples representation.

3AC for B.java file

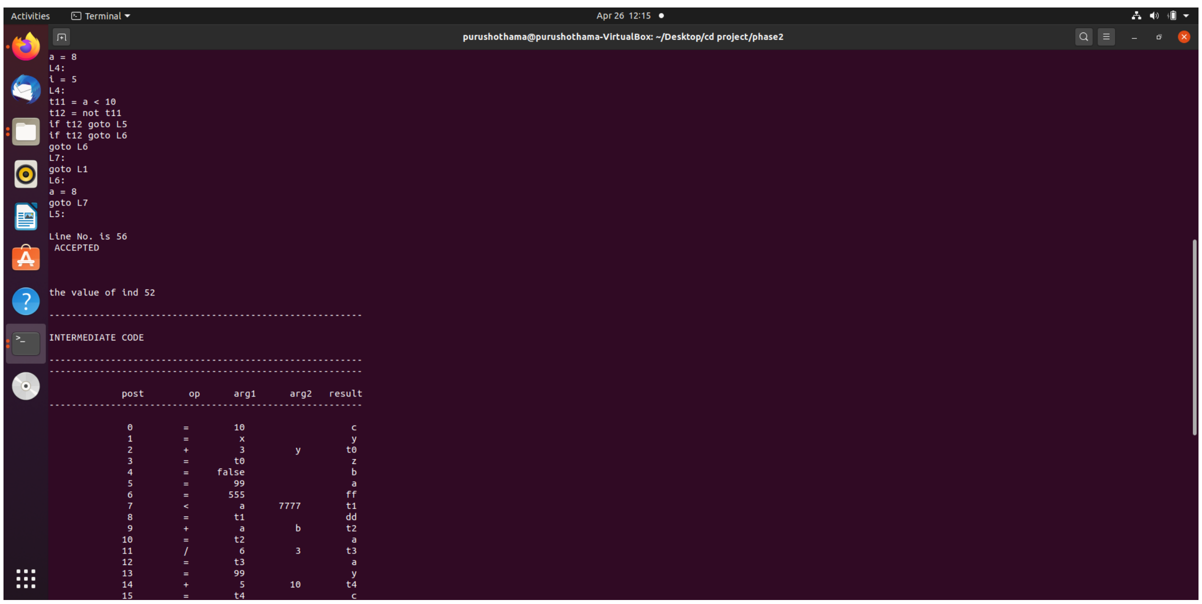


Activities Terminal

Apr 26 12:15

purushothama@purushothama-VirtualBox: ~/Desktop/cd project/phase2

```
c = 10
y = x
t0 = 3 + y
z = t0
b = false
a = 99
ff = 555
t1 = a < 7777
dd = t1
t2 = a + b
a = t2
t3 = 6 / 3
a = t3
y = 99
t4 = 5 + 10
c = t4
t5 = 5 + y
ff = t5
t6 = y >= 99
dd = t6
zx = 10
i = 0
oo = 444
t7 = a == 2

t8 = not t7
if t8 goto L1
t9 = a >= 4

t10 = not t9
if t10 goto L2
a = 10
sam = 15
goto L3
L2:
a = 4
L3:
goto L4
L1:
a = 8
L4:
i = 5
L4:
t11 = a < 10
t12 = not t11
if t12 goto L5
if t12 goto L6
goto L6
```



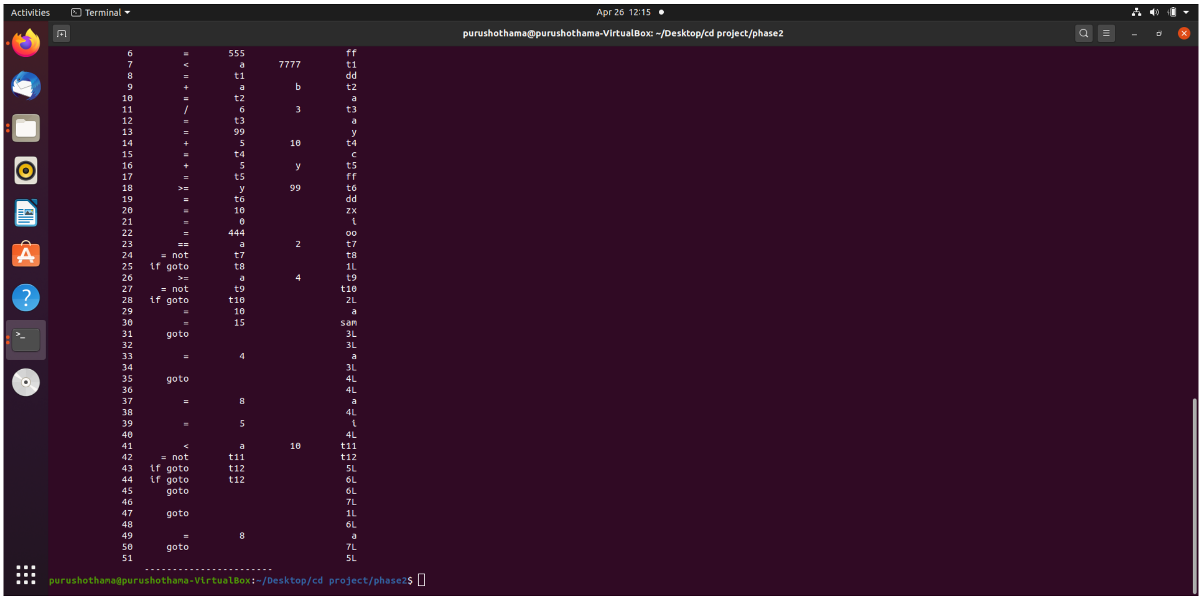
Activities Terminal

Apr 26 12:15

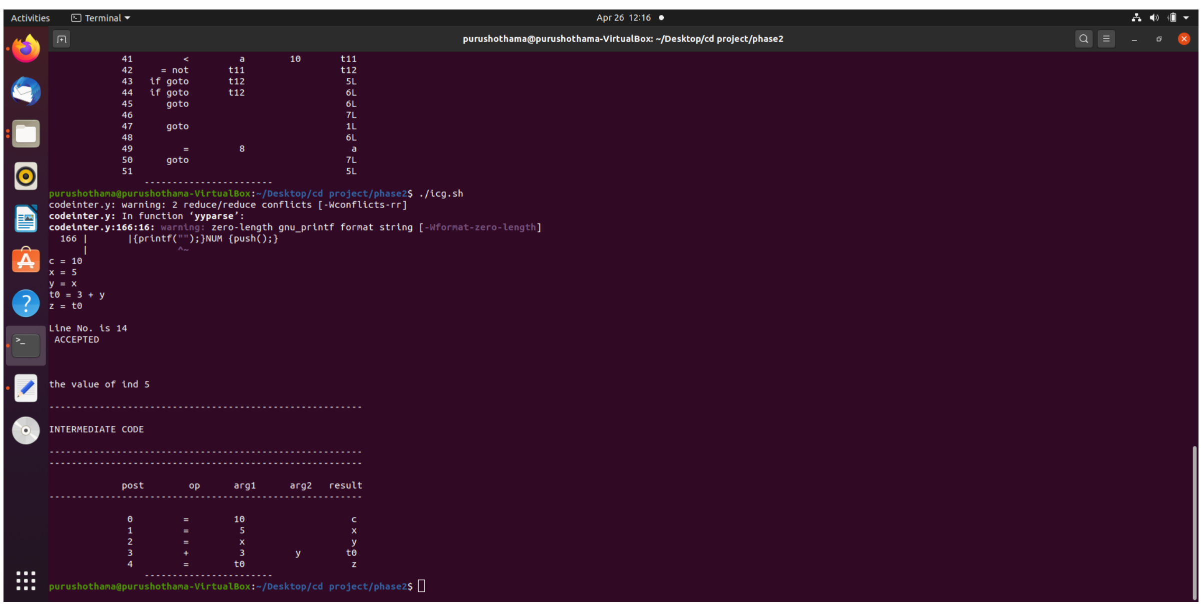
purushothama@purushothama-VirtualBox: ~/Desktop/cd project/phase2

```
a = 8
L4:
i = 5
L4:
t11 = a < 10
t12 = not t11
if t12 goto L5
if t12 goto L6
goto L6
L7:
goto L1
L6:
a = 8
goto L7
L5:

Line No. is 56
 ACCEPTED


the value of ind 52

-------------------------------------------------------

INTERMEDIATE CODE

-------------------------------------------------------
-------------------------------------------------------
```

post	op	arg1	arg2	result
0	=	10		c
1	=	x		y
2	+	3	y	t0
3	=	t0		z
4	=	false		b
5	=	99		a
6	=	555		ff
7	<	a	7777	t1
8	=	t1		dd
9	+	a	b	t2
10	=	t2		a
11	/	6	3	t3
12	=	t3		a
13	=	99		y
14	+	5	10	t4
15	=	t4		c



3AC for a.java File

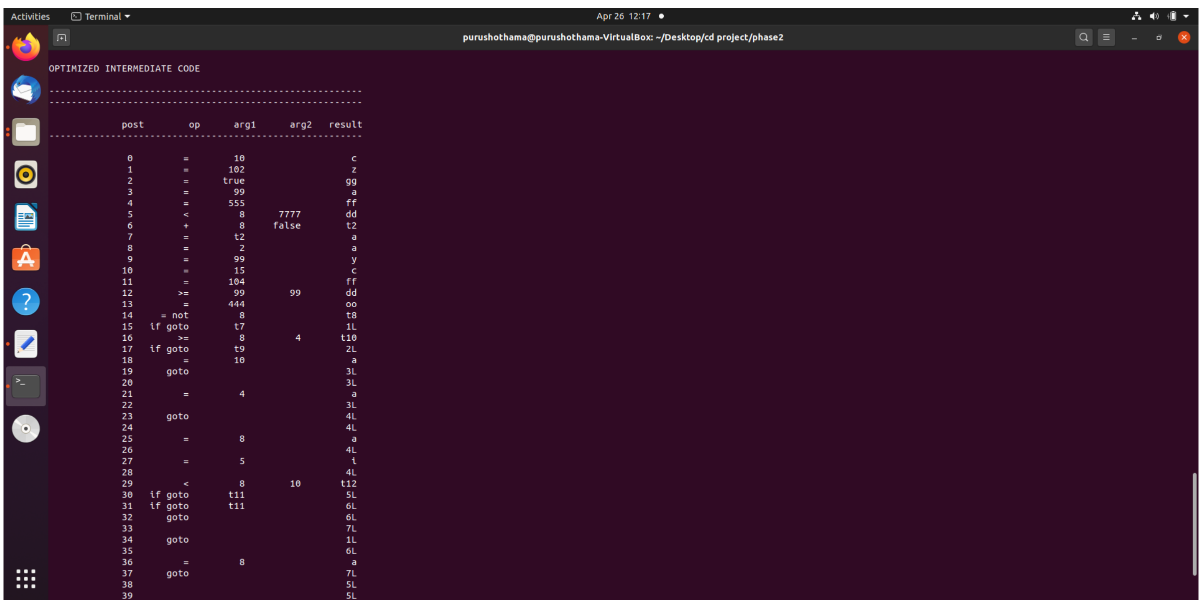


CODE OPTIMIZATION

We have implemented 4 code optimization methods

- **constant folding**
- **copy propagation**
- **constant propagation**
- **Dead code elimination**

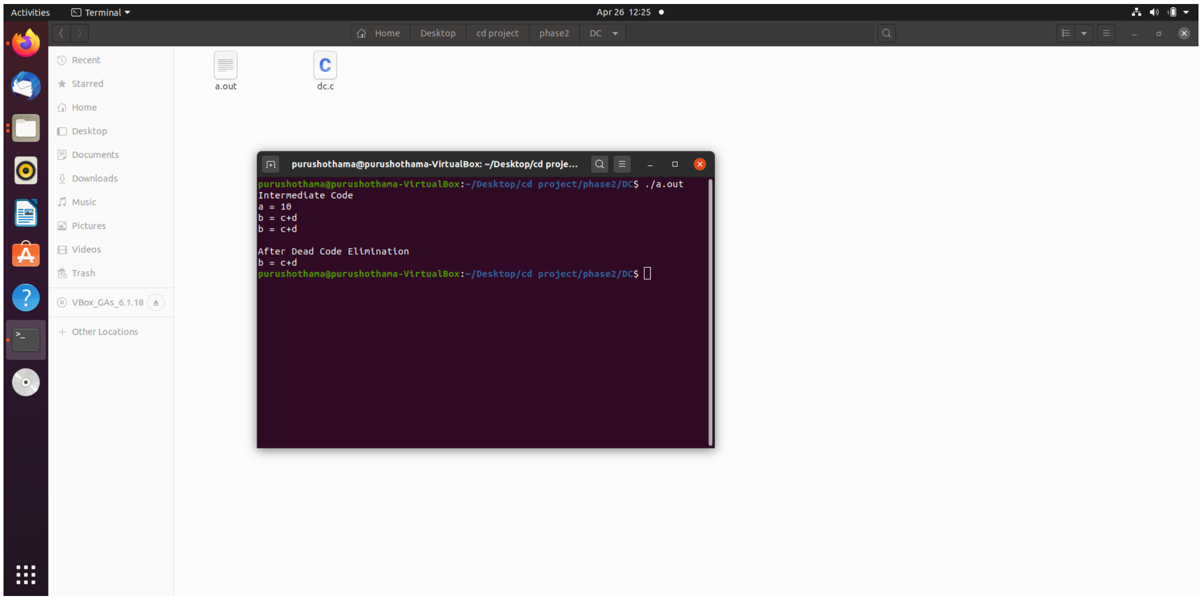
This is the output after applying the 4 types of code Optimization techniques.

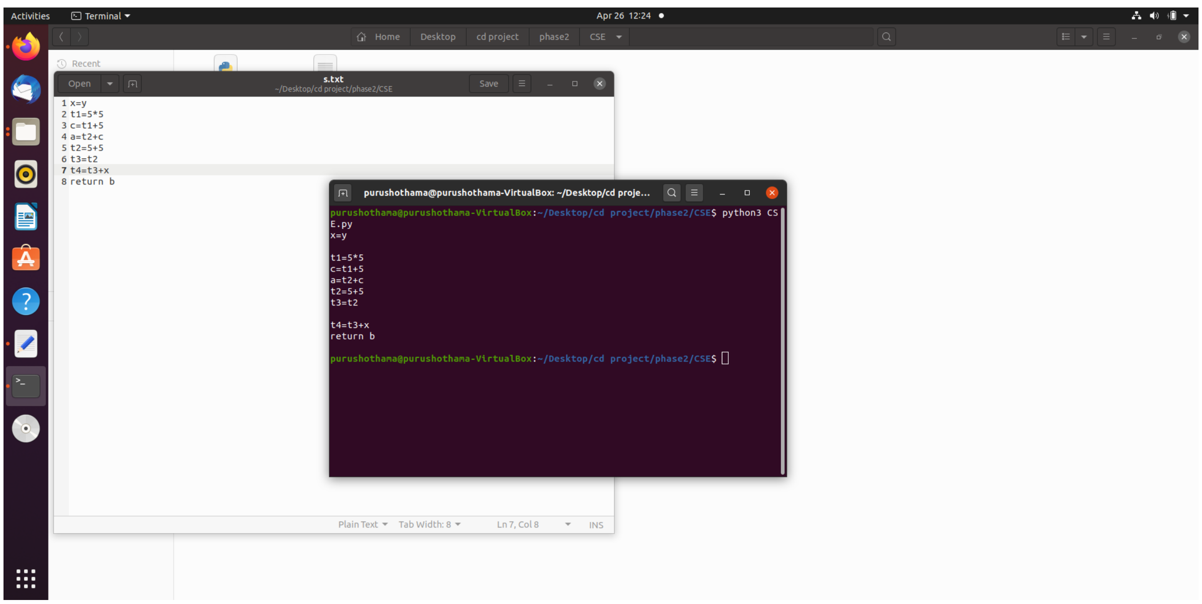


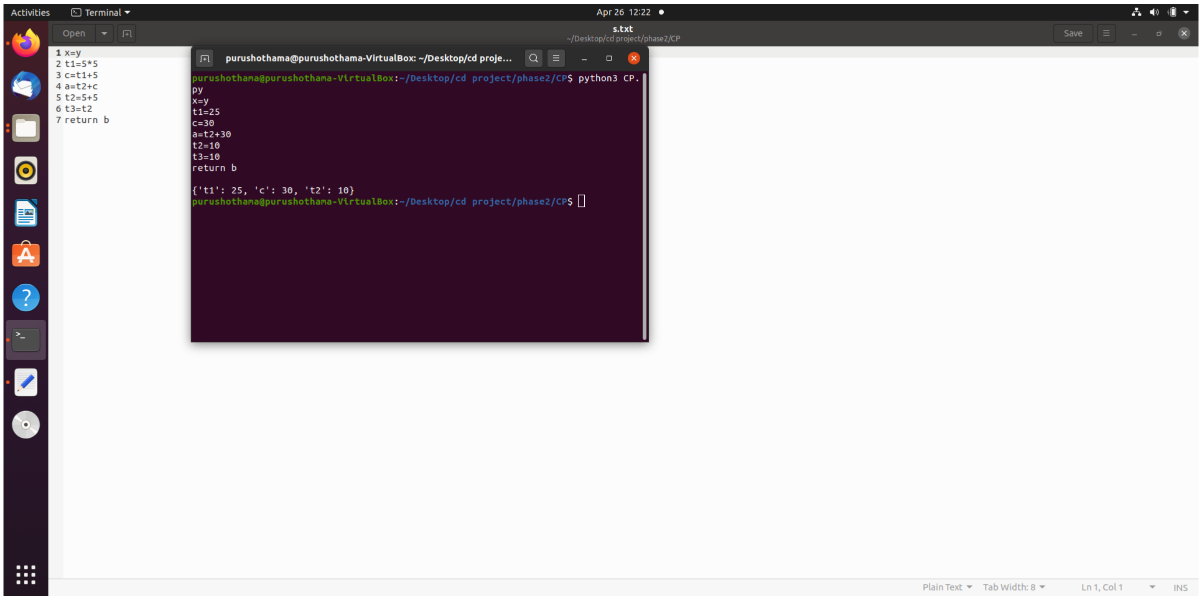
OPTIMIZED INTERMEDIATE CODE

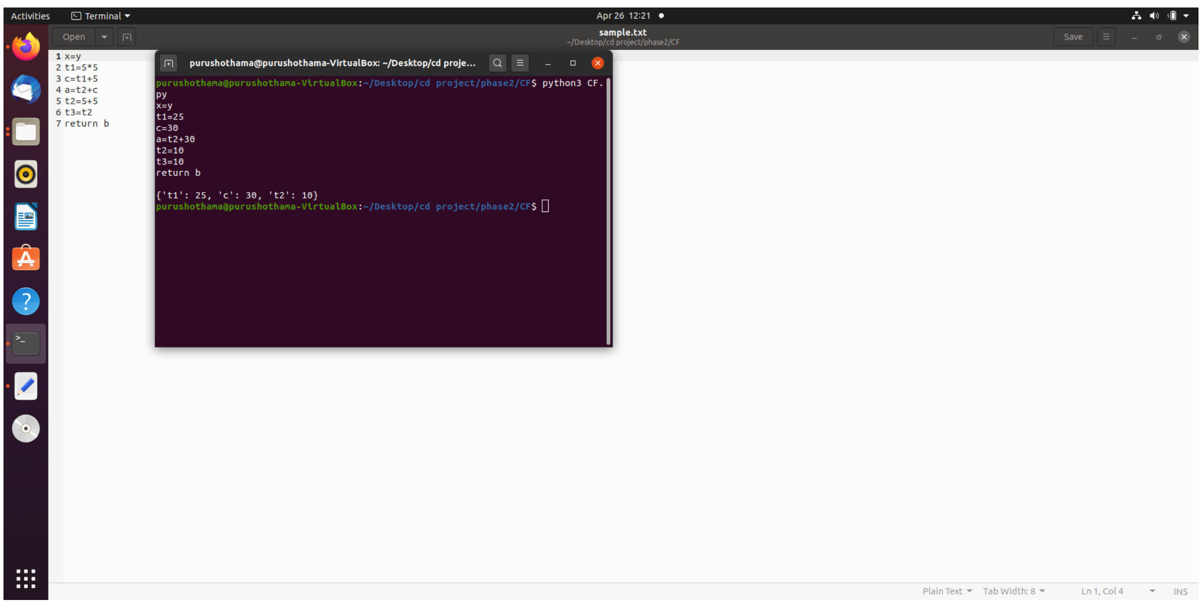
post	op	arg1	arg2	result
0	=	10		c
1	=	102		z
2	=	true		gg
3	=	99		a
4	=	555		ff
5	<	8	7777	dd
6	+	8	false	t2
7	=	t2		a
8	=	2		a
9	=	99		y
10	=	15		c
11	=	104		ff
12	>=	99	99	dd
13	=	444		oo
14	= not	8		t8
15	if goto	t7		1L
16	>=	8	4	t10
17	if goto	t9		2L
18	=	10		a
19	goto			3L
20				3L
21	=	4		a
22				3L
23	goto			4L
24				4L
25	=	8		a
26				4L
27	=	5		i
28				4L
29	<	8	10	t12
30	if goto	t11		5L
31	if goto	t11		6L
32	goto			6L
33				7L
34	goto			1L
35				6L
36	=	8		a
37	goto			7L
38				5L
39				5L

Test case code optimization for 4 techniques









9. CONCLUSION

A compiler for JAVA was thus created using lex and yacc. In addition to the constructs specified, basic building blocks of the language (declaration statements, assignment statements, etc) were handled.

This compiler was built keeping the various stages of Compiler Design, ie, Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Optimisation in mind.

As a part of each stage, an auxiliary part of the compiler was built (Symbol Table, Abstract Syntax Tree and Intermediate Code). Each of these components are required to compile code successfully.In addition to this, basic error handling has also been implemented.

10. FUTURE ENHANCEMENTS

As mentioned above, we can use separate structures for the different types of tokens and then declare a union of these structures. This way, memory will be properly utilized. For constant propagation at branches, we need to implement SSA form of the code. This will work well in all cases and yield the right output.

REFERENCES

https://www.javatpoint.com/lex
http://software.ucv.ro/~mbrezovan/Cd/Cd_StandardProj.html
https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
https://tldp.org/HOWTO/Lex-YACC-HOWTO-6.html
https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf