Report on

"Constructing Java compiler for IF, ELSE and FOR constructs using Lex and Yacc"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

ABHIRAM G.K	**PES1201700102**
CH ABHISHEK	**PES1201700194**
ADVAITH K V	**PES1201700207**

Under the guidance of

Prof. Preet Kanwal
Asst.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

Introduction:

In this project, we will be building a mini-compiler for the Java programming language. Java programming is a general-purpose, object oriented language. We will be using Lex and Yacc for the

implementation of the Java compiler in this project. So in this project, we have five different phases. The five phases are:

- ❏ Lexical Analysis
- ❏ Syntax Analysis
- ❏ Semantic Analysis
- ❏ Intermediate code generation
- ❏ Target code generation

We have to go through all the above mentioned phases to get assembly code since machines cannot understand the Java code which we have written.
Beginning with the lexical phase, here we will be generating different types of tokens from the code we have written. Then comes syntax analysis, in this phase with the written grammar rules we will be checking the syntax of the java program we have written. Then semantic phase, in this phase we will do semantic checks of the code and we will also build Abstract Syntax Tree. After the semantic phase, next is Intermediate code generation, in this phase, we will be generating three address codes which are intermediate codes. After this phase, we will generate assembly code which is the target code in this case. So finally we build a code that is easily understood by the machine.

So, our project will mainly focus on building these three constructs

- ❏ For looping statement
- ❏ If – else conditional statement

Along with these two constructs, we will be building a symbol table, we will be performing handling error, we will be generating syntax tree, we will be converting the given Java program into intermediate code and with the help of intermediate code which is a three address code into target code. The target code generated will be in the MIPS assembly code language.

So if we give Java program as the input we will give Symbol table, syntax tree, intermediate code, and target code as the output.

Original Java code:

```java
public class a
{
    public static void main(String []args)
    {
        int a=3,b=2;
        for(c=20;c<25;c++)
        {
            int m=15;
        }
        int k=3*a;
        if(k>2)
        {
            int j=1;
        }
        else
        {
            int i=0;

        }

    }
}
```

Assembly Code:

```
-------------------- ASSEMBLY CODE (MIPS ARCHITECTURE)--------------------
main:
        la $s0, b
        addi $s7,$0,2
        sw $s7, 0($s0)
        la $s0, a
        addi $s7,$0,3
        sw $s7, 0($s0)
        la $s0, c
        addi $s7,$0,20
        sw $s7, 0($s0)

L1:
        lw $s1, c
        slti t0, $s1, 25
        bgt $t0, $0, L2
        b L3

L4:
        lw $s1, c
        addi t1, $s1, 1
        la $s0, c
        sw t1, 0($s0)
        b L1

L2:
        la $s0, m
        addi $s7,$0,15
        sw $s7, 0($s0)
        b L4

L3:
        lw $s1, a
        addi $s2, $0, 3
        mult $s1, $s2
        mflo t2
        la $s0, k
        sw t2, 0($s0)
```

ARCHITECTURE OF LANGUAGE:

- Beginning with generating with tokens - We will be using Lex to generate tokens
- We will be using Yacc for writing grammar rules for the tokens generated by Lex/Flex
- All the tokens which are further needed in the project are written
- The first task we do with the tokens generated is we ignore the single line and multi-line comments. We just ignore when the tokens are generated
- Grammar rules are written for:
  - For
  - If else
- The main places where we did error checking was
  - If a variable is already declared
  - When the variable is not found in the symbol table
  - Syntax errors

- We built the syntax tree we printed the tree in pre-order.
- After verifying the syntax tree, we converted the Java code into three address codes which is a part of the Intermediate code generation phase.
- The optimization part of the intermediate code is done with the help of three techniques
  - Constant folding
  - Dead code elimination
  - Eliminating common subexpression
- We have written a python file to convert optimized intermediate code to assembly code as a part of Target code generation.

Literature Survey:

1) Lex,yacc and its internal working. We have got this information from tldp.org website and also from the CD Lab resources where we learnt how to run the lex and yacc files and also what lex and yacc does such as lex is a tool for writing lexical analyzers and yacc is a tool for constructing parsers.

2) Building a mini compiler from tutorials point and our prescribed textbook from where we have got our basic ideas of what each phase in the compiler does and got some detail explanation of concepts with some good example code snippets .

3) From The below mentioned website we have learnt a little bit of MIPS architecture where we learnt some immediate instructions,saved registers which helped us in doing our target code generation. The website provides us with a good view of all syntaxes and the purpose of each command and what it does and its semantics

CONTEXT FREE GRAMMAR:

```
START: MODIFIER T_CLASS T_ID '{'
        MODIFIER TYPE T_MAIN'('T_STRING'['']' T_ID')' '{'
        S
       '}' '}' ;

MODIFIER:W1 W2;

W1:T_PUBLIC
   |T_PRIVATE ;

W2:T_STATIC
         | ;

S:          DECLR ';' S
            |ASSGN ';' S
            |IF ELSE S
            |FOR '{'S'}' S
            |UNREXPR';' S
            |;

ASSGN:      Assignment
            |Array_initialisation;

DECLR:      Variable_declaration
            |Array_declaration ;

IF:         T_IF '('LOGICALOREXPR')' '{'S'}' ;

ELSE:       T_ELSE '{'S'}'
            | ;

FOR: T_FOR'('';'';'')'
            |T_FOR'('INIT';'';'')'
            |T_FOR'('INIT';'LOGICALOREXPR';'')'
```

```
                |T_FOR'('INIT';'';'UNREXPR')'
                |T_FOR'('';'LOGICALOREXPR';'')'
                |T_FOR'('';'LOGICALOREXPR';'UNREXPR')'
                |T_FOR'('INIT';'LOGICALOREXPR';'UNREXPR')'
                |T_FOR'('';'';'UNREXPR')' ;

INIT: Variable_declaration
                |Assignment  ;

UNREXPR:                T_INC Expr
                |T_DEC Expr
                |Expr T_INC
                |Expr T_DEC
                |LOGICALOREXPR;


Variable_declaration:Type T_ID T_ASSGN LOGICALOREXPR X
                |Type T_ID X ;



X:      ','Assignment1 X
        |',' T_ID X
        | ;

Assignment1:T_ID Assignment_operator LOGICALOREXPR ;

Array_declaration:Type Brackets T_ID
                        |Type T_ID Brackets ;

Brackets:           WI
                        |WOI ;

WOI:                    '['']'WI
                        |'['']';

WI:             '[' INDEX ']'
                        | '[' INDEX ']' WOI ;
```

```
INDEX:              T_NUM
                    | T_ID ;

Array_initialisation:Array_declaration Assignment_operator K;

K:                  V
                    |V',' K
                    |T_NEW Type WI ;

V:                  T_NUM
                    |R ;

R:                  '{'K'}' ;

Type:       T_INT
                    |T_DOUBLE
                    |T_CHAR
                    |T_STRING
                    |T_VOID  ;

Assignment: T_ID Assignment_operator LOGICALOREXPR ;

Assignment_operator:T_ASSGN
                        |T_ADDASSGN
                        |T_SUBASSGN
                        |T_MULASSGN
                        |T_DIVASSGN
                        |T_ANDASSGN
                        |T_ORASSGN
                        |T_XORASSGN
                        |T_MODASSGN ;

LOGICALOREXPR:LOGICALOREXPR T_LOGOR LOGICALANDEXPR
            |LOGICALANDEXPR   ;

LOGICALANDEXPR: LOGICALANDEXPR T_LOGAND
EQUALITYEXPR
            |EQUALITYEXPR ;
```

```
EQUALITYEXPR: EQUALITYEXPR T_EQ RELEXPR
            | EQUALITYEXPR T_NEQ RELEXPR
            |RELEXPR ;

RELEXPR: RELEXPR T_LT ADDEXPR
            | RELEXPR T_GT ADDEXPR
            | RELEXPR T_LTEQ ADDEXPR
            | RELEXPR T_GTEQ ADDEXPR
            |ADDEXPR ;

ADDEXPR:  ADDEXPR T_ADD MULTEXPR
            | ADDEXPR T_SUB MULTEXPR
            |MULTEXPR ;

MULTEXPR: MULTEXPR T_MUL Expr
            | MULTEXPR T_DIV Expr
            | MULTEXPR T_MOD Expr
            | Expr  ;

Expr:            '('LOGICALOREXPR')'
                   |T_NUM
                   |T_ID ;
```

DESIGN STRATEGY:

Symbol table:

First when the lex file generates the tokens then with the grammar we check if it is variable declaration. If it is variable declaration we add it to the symbol table. If it is variable initialization we update it to the already existing symbol table. Symbol table is being stored as the structure. The structure consists of the fields which are the name of the variable, data type of variable, value which is assigned to the variable and scope of the variable.

Abstract Syntax tree:

Abstract syntax trees consist of nodes and childrens as its code. Those nodes consists of the operators or operands. mainly parents contain the operators and children have operands. These are interconnected to form a tree. We used quadraple tree.This tree starts from ‘main’ followed by the code as the children. The nodes below ‘main’ consists of code from main function in Java code. Then we write a display() function to display the code.

Intermediate code generation:

Whenever we encounter the statements, we match it with the grammar and we print the grammar in the form of three address code accordingly. Whenever we have to print If or else or for, we take care of printing goto, labels and other required statements .

Code optimization:

We do three types of optimization:

- **Constant folding:** In this we try to see if there are only numbers on the RHS. If it has only numbers, then we solve those equations and replace operands and operators with the final output so it shouldn’t create much burden during run time.
- **Dead code elimination:** In this we try to remove the Java statements which are not used further. We keep track of the temporary variables and Java variables which are not used and we eliminate those.
- **Eliminating common subexpression :** Here we replace the subexpression with their stored values.

Error handling:

Here we check for errors such as redeclaration and undeclared variables and give appropriate messages with respective error lines. And we used panic mode recovery where it it ignores error and goes on to execute the code.

Target code generation:

Our target code is MIPS assembly code. We use temporary registers for temporary variables and saved registers for other variables. Register allocation method used is round robin.

Implementation:

Symbol table:

When there is variable declaration or variable assignment or expression evaluation, first we check if the variable is in the symbol table. The symbol table is a linked list. We traverse the list to search the table. If it is not present then we call fill() function for adding those variables. If the variables are present then we call update function which will update the symbol table. lookupsymbol() function is used to check if the variable is present or not.

Abstract Syntax tree:

The structure of the AST quadruple tree. Each parent node has four child nodes as well as the value and name of the node as its data. Here we implemented two functions which are newnode() and newleaf(). newnode() is called when it is a operator or an operand is a variable or if we want to store ‘FOR’ or ‘IF’ or ‘ELSE’ or any other keyword. newleaf() id called when we encounter a type as operand.
Then we have a printBT() function which will print everything in Pre-Order format as this format is very easy to understand.

Intermediate code generation:

Reading the statement, matching with the grammar and printing based on the grammar. We used a linked list to keep record of all the variables and values.

Code optimization:

We have written three functions, eliminate_common_subexpressions() , remove_dead_code(), fold_constants() which will do the work of optimization.

Error handling:

In case of any variable declaration it checks the symbol table for the name of the variable and the scope and it declares the error if it already exists. In case of variable initialization we do the same and it gives the error if the variable is not already exisiting in the table.

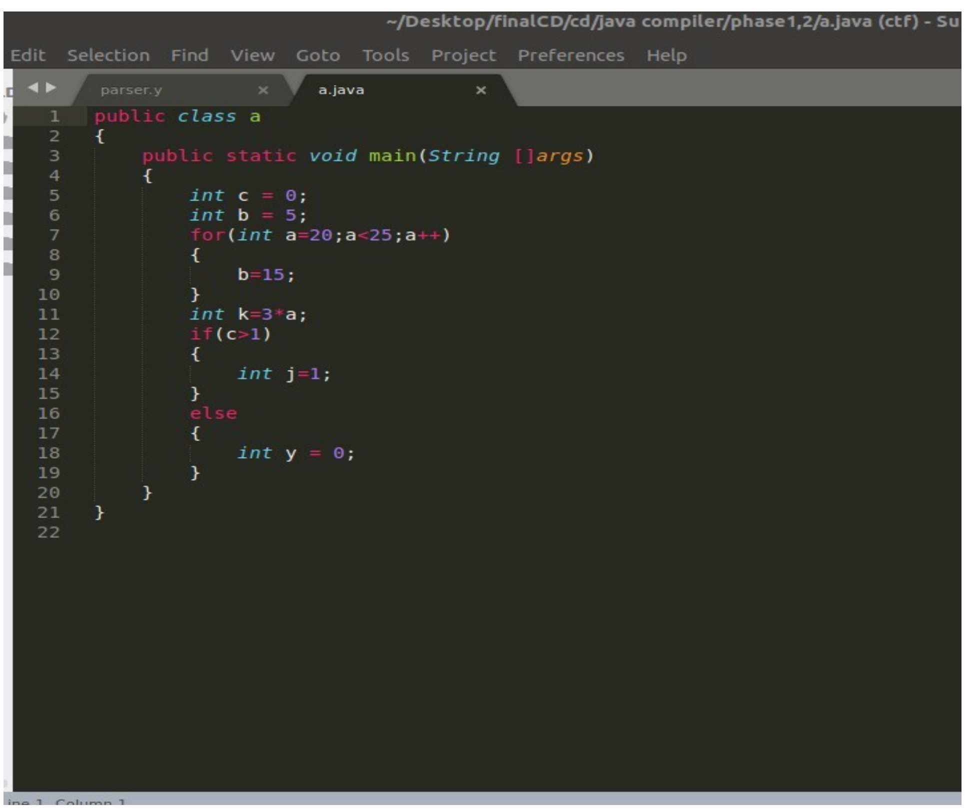
Target code generation:

The Target code is in MIPS Architecture and it is written with the help of python language. Mainly dict() is used to handle the registers. In case of variables we are loading it into saved registers operating on it and storing it back immediately.

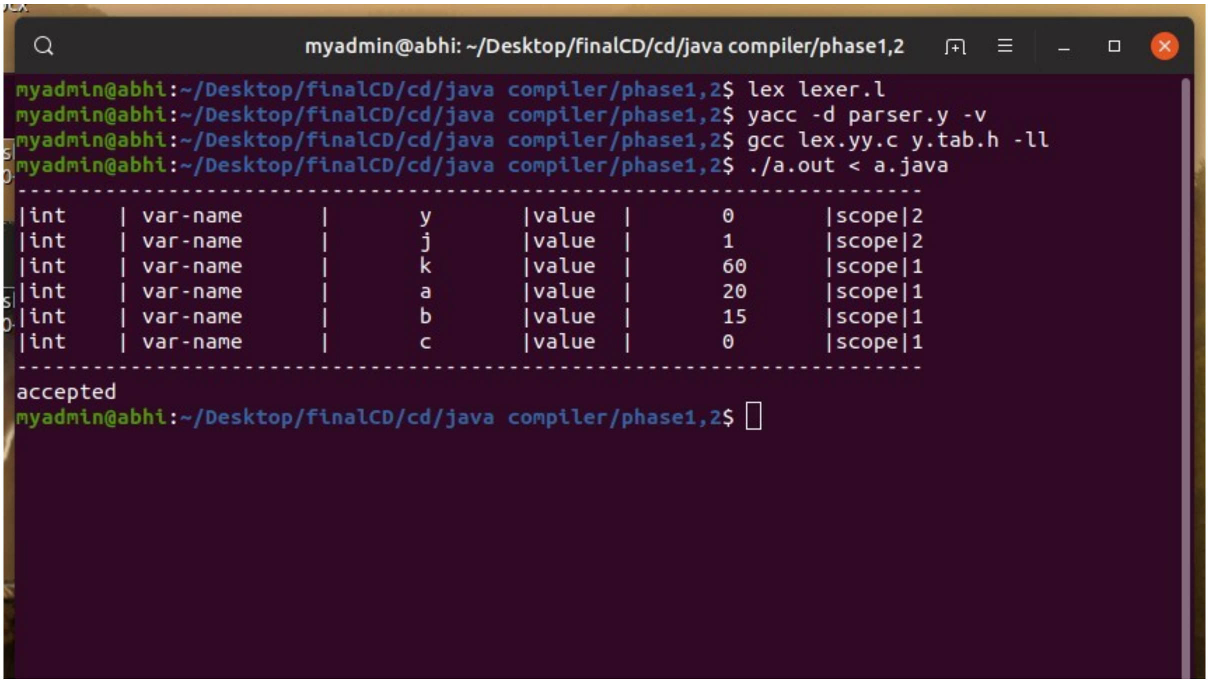
<u>SNAPSHOTS:</u>

Symbol Table :

Input :

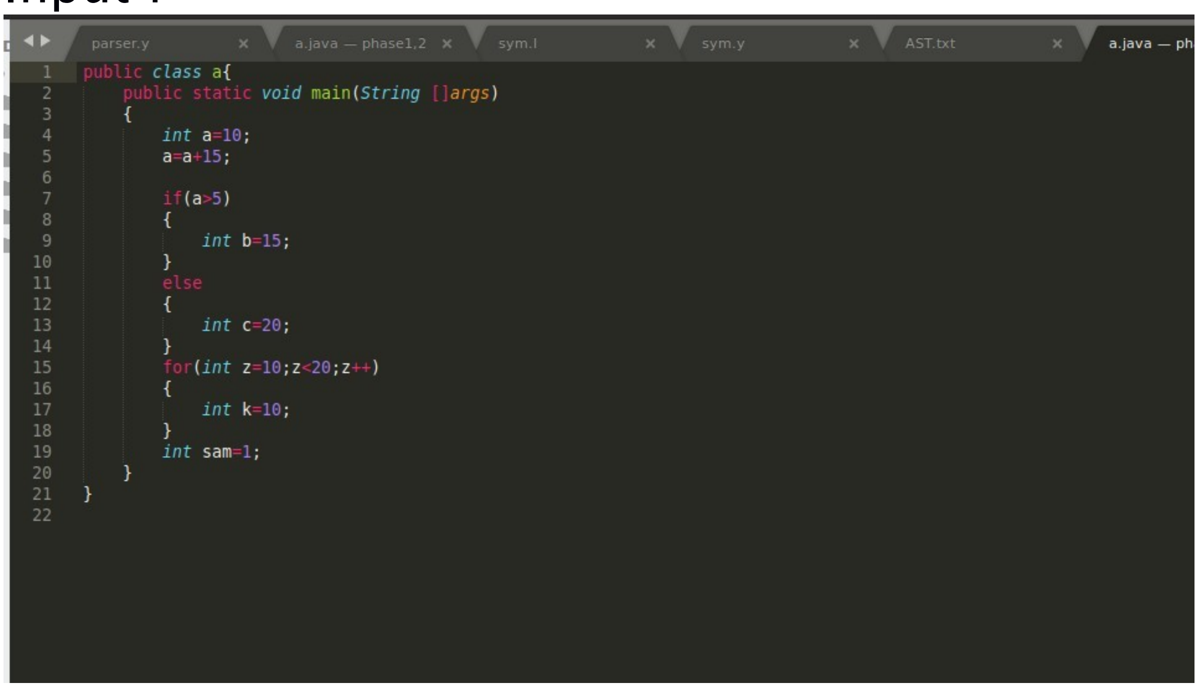


Output:

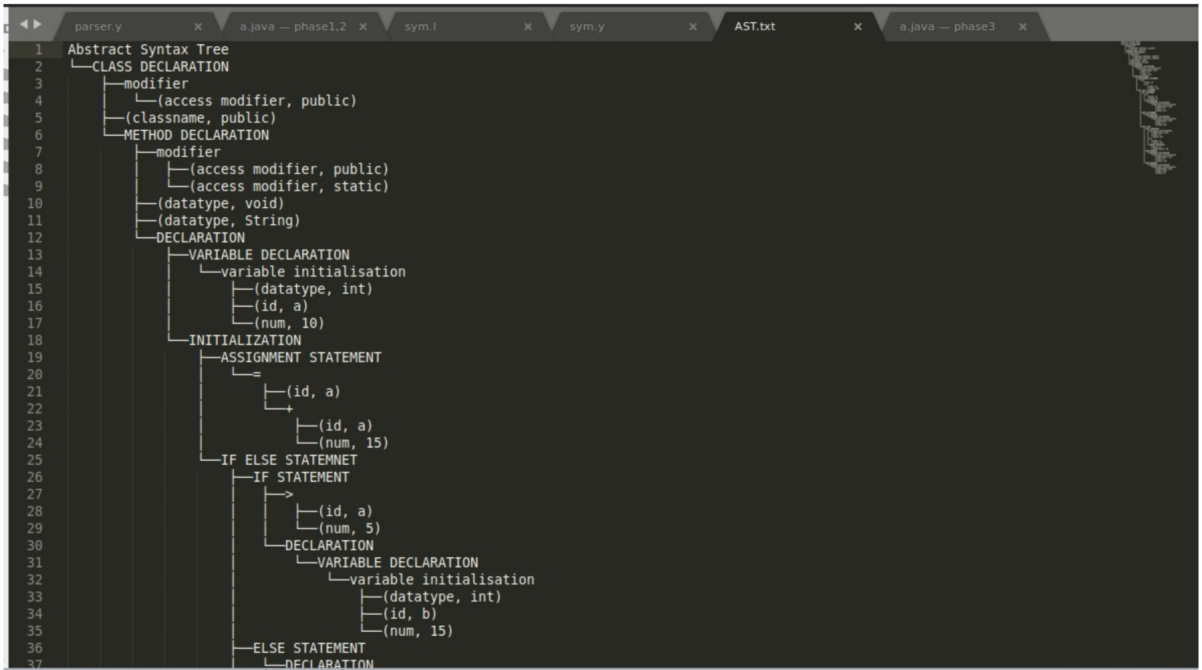


Abstract Syntax Tree :

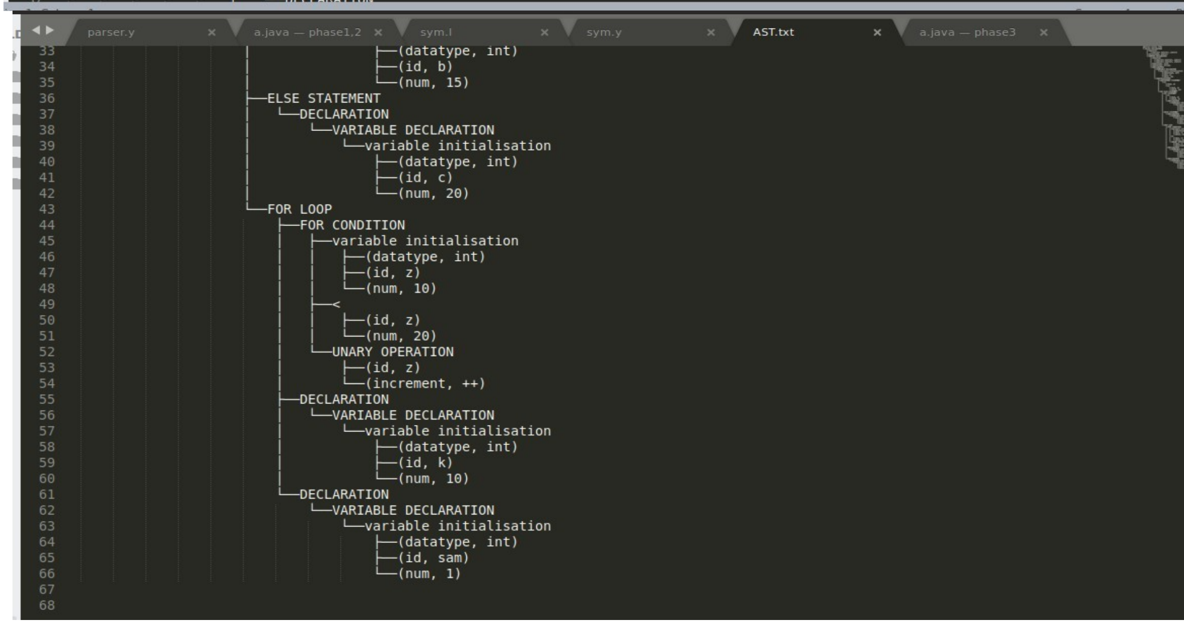
Input :



Output :



```
Abstract Syntax Tree
└──CLASS DECLARATION
    ├──modifier
    │   └──(access modifier, public)
    ├──(classname, public)
    └──METHOD DECLARATION
        ├──modifier
        │   ├──(access modifier, public)
        │   └──(access modifier, static)
        ├──(datatype, void)
        ├──(datatype, String)
        └──DECLARATION
            ├──VARIABLE DECLARATION
            │   └──variable initialisation
            │       ├──(datatype, int)
            │       ├──(id, a)
            │       └──(num, 10)
            └──INITIALIZATION
                ├──ASSIGNMENT STATEMENT
                │   └──=
                │       ├──(id, a)
                │       └──+
                │           ├──(id, a)
                │           └──(num, 15)
                └──IF ELSE STATEMNET
                    ├──IF STATEMENT
                    │   ├──>
                    │   │   ├──(id, a)
                    │   │   └──(num, 5)
                    │   └──DECLARATION
                    │       └──VARIABLE DECLARATION
                    │           └──variable initialisation
                    │               ├──(datatype, int)
                    │               ├──(id, b)
                    │               └──(num, 15)
                    ├──ELSE STATEMENT
                    │   └──DECLARATION
```



```
                    │               ├──(datatype, int)
                    │               ├──(id, b)
                    │               └──(num, 15)
                    ├──ELSE STATEMENT
                    │   └──DECLARATION
                    │       └──VARIABLE DECLARATION
                    │           └──variable initialisation
                    │               ├──(datatype, int)
                    │               ├──(id, c)
                    │               └──(num, 20)
                    └──FOR LOOP
                        ├──FOR CONDITION
                        │   ├──variable initialisation
                        │   │   ├──(datatype, int)
                        │   │   ├──(id, z)
                        │   │   └──(num, 10)
                        │   ├──<
                        │   │   ├──(id, z)
                        │   │   └──(num, 20)
                        │   └──UNARY OPERATION
                        │       ├──(id, z)
                        │       └──(increment, ++)
                        ├──DECLARATION
                        │   └──VARIABLE DECLARATION
                        │       └──variable initialisation
                        │           ├──(datatype, int)
                        │           ├──(id, k)
                        │           └──(num, 10)
                        └──DECLARATION
                            └──VARIABLE DECLARATION
                                └──variable initialisation
                                    ├──(datatype, int)
                                    ├──(id, sam)
                                    └──(num, 1)
```

Intermediate Code :

Input :

```java
public class a
{
    public static void main(String []args)
    {
        int a=3;
        int b=4;
        int c=5;
        a=a+3+b+c;

        if(a<2)
        {
            int j=1;
        }
        else
        {
            int i = 0;
            b = 10;
        }
    }
}
```

Output :

```
a = 3
b = 4
c = 5
T0 = a + 3
T1 = T0 + b
T2 = T1 + c
a = T2
T3 = T2 < 2
if T3 goto L1
goto L2
L1:
j = 1
goto L3
L2:
i = 0
b = 10
L3:
```

Optimized Code:

Input:

```
icg.txt    optimize.py
c = 10
T1 = 3 + 5
T2 = 3 + 5
T3 = T1 + T2
T4 = T1 + T2
a = 10
a = 3 + 5
a = T3
L1:
T4 = a < 25
if T4 goto L2
goto L3
L4:
T5 = a + 1
a = T5
goto L1
L2:
m = 15
goto L4
L3:
T6 = 3 * a
k = T6
```

Output:

```
myadmin@abhi: ~/Desktop/finalCD/cd/java compiler/new_new_phase5
myadmin@abhi:~/Desktop/finalCD/cd/java compiler/new_new_phase5$ python3 optimize.py icg.txt
c = 10
T1 = 8
T2 = T1
T3 = T1 + T2
T4 = T3
a = 10
a = T1
a = T3
L1:
T4 = a < 25
if T4 goto L2
goto L3
L4:
T5 = a + 1
a = T5
goto L1
L2:
m = 15
goto L4
L3:
T6 = 3 * a
k = T6
myadmin@abhi:~/Desktop/finalCD/cd/java compiler/new_new_phase5$
```

Target Code:

Input :

out1.s × target_code.py × test2.txt × out2.s × if.y × sym.l — phase4 ×

```
a = 3
b = 4
c = 5
T0 = a + 3
T1 = T0 + b
T2 = T1 + c
a = T2
T3 = 0
if T3 goto L1
goto L2
L1:
j = 1
goto L3
L2:
i = 0
b = 10
L3:
```

Output :

out1.s × target_code.py × test2.txt × out2.s × if.y × sym.l — phase4 ×

```
-------------------- ASSEMBLY CODE (MIPS ARCHITECTURE)--------------------
main:
    la $s0, a
    addi $s7,$0,3
    sw $s7, 0($s0)
    la $s0, b
    addi $s7,$0,4
    sw $s7, 0($s0)
    la $s0, c
    addi $s7,$0,5
    sw $s7, 0($s0)
    lw $s1, a
    addi t0, $s1, 3
    lw $s2, b
    add t1, $s2, t0
    lw $s2, c
    add t2, $s2, t1
    la $s0, a
    sw t2, 0($s0)
    la $s0, T3
    addi $s7,$0,0
    sw $s7, 0($s0)
    bgt $t3, $0, L1
    b L2

L1:
    la $s0, j
    addi $s7,$0,1
    sw $s7, 0($s0)
    b L3

L2:
    la $s0, i
    addi $s7,$0,0
    sw $s7, 0($s0)
    la $s0, b
    addi $s7,$0,10
    sw $s7, 0($s0)

L3:
.data
```

RESULTS:

The lex and yacc codes are compiled and executed by the following terminal commands to
parse the given input file.

In AST folder:

```
lex -l parser.l
yacc -vd parser.y
gcc lex.yy.c y.tab.c
./a.out < a.java
```

In ICG folder:

```
lex -l icg.l
yacc -vd icg.y
gcc lex.yy.c y.tab.c
./a.out < a.java
```

In optimize:

```
python optimize.py icg.txt
```

In target code :

```
python target_code.py
```

By running these commands we get the desired outcomes such as the Symbol Table, Abstract syntax tree, Intermediate code, Optimized code, Assembly code .

Shortcomings:

- We can use only INT. Its throws error if we use other data types
- Error handling is not that great.
- Limited to fewer constructs.

CONCLUSION:

We have built a mini Java compiler for constructs IF, ELSE and FOR conditions and also handled basic Java syntaxes. We also generated an Annotated syntax tree in preorder format, Intermediate code which is three address code and also Target code in MIPS architecture.

FUTURE ENHANCEMENTS:

- Extend grammar for while, do-while loop, switch and ternary operators
- Improve error handling
- To handle all other datatypes (char,string,bool)

REFERENCES:

- Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S.Lam, Ravi Sethi, Jeffrey D. Ullman
- https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
- https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm
- http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html