Report on

“Mini JAVA Compiler using C”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Aiswarya Y B	**PES1201701271**
Suhail Rahman	**PES1201701420**
Riya Vijay	**PES1201701814**

Under the guidance of

Professor Kiran P
Assistant Professor
PES University, Bengaluru

January – May 2020

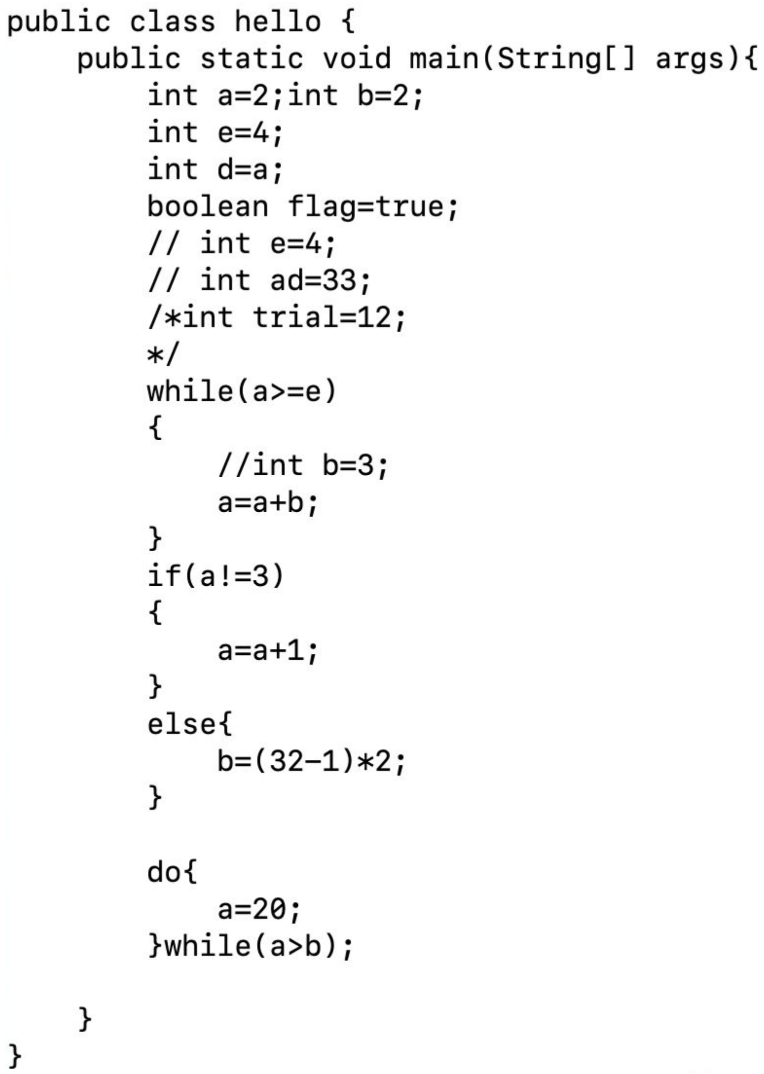
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

INTRODUCTION :

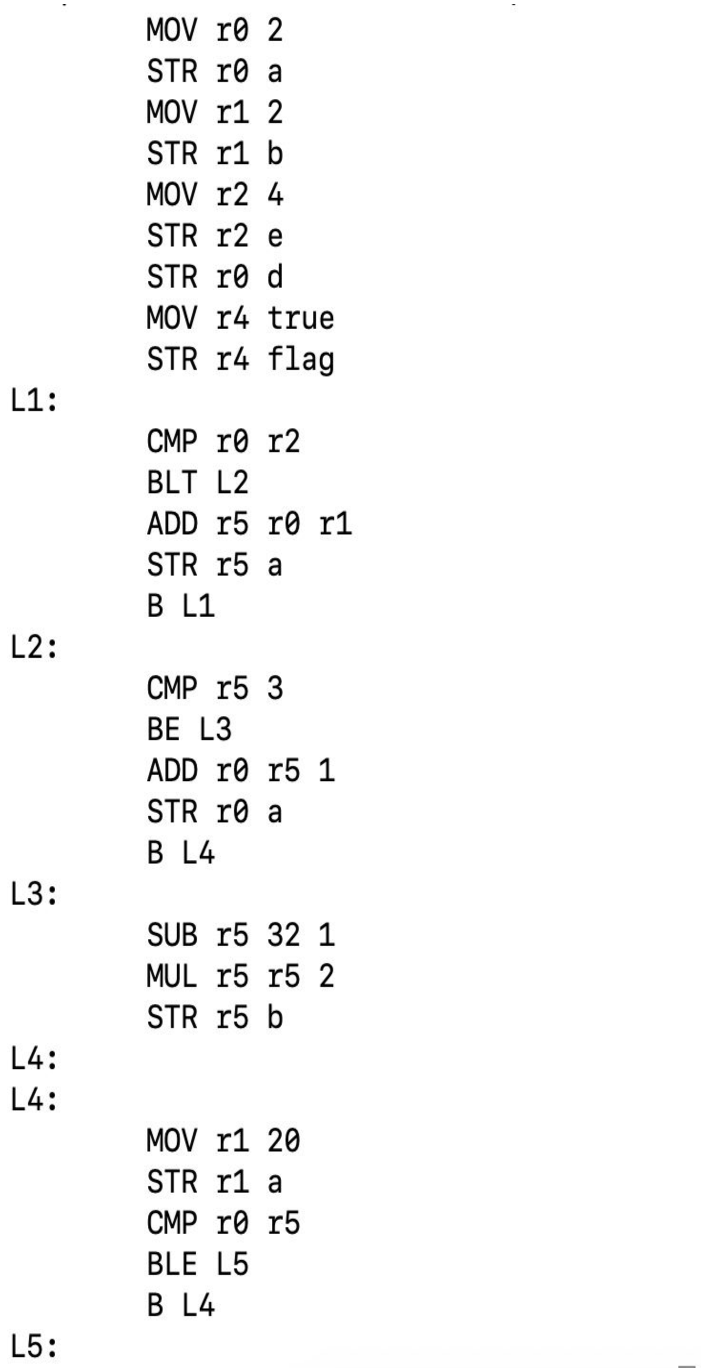
Our project aims at designing and implementing a JAVA compiler using C language. We have implemented various phases of the compiler using lex and yacc. The main objective of our project is to compile the input JAVA file to the assembly code.

SAMPLE INPUT :



```
public class hello {
    public static void main(String[] args){
        int a=2;int b=2;
        int e=4;
        int d=a;
        boolean flag=true;
        // int e=4;
        // int ad=33;
        /*int trial=12;
        */
        while(a>=e)
        {
            //int b=3;
            a=a+b;
        }
        if(a!=3)
        {
            a=a+1;
        }
        else{
            b=(32-1)*2;
        }

        do{
            a=20;
        }while(a>b);

    }
}
```

SAMPLE OUTPUT :



```
        MOV r0 2
        STR r0 a
        MOV r1 2
        STR r1 b
        MOV r2 4
        STR r2 e
        STR r0 d
        MOV r4 true
        STR r4 flag
L1:
        CMP r0 r2
        BLT L2
        ADD r5 r0 r1
        STR r5 a
        B L1
L2:
        CMP r5 3
        BE L3
        ADD r0 r5 1
        STR r0 a
        B L4
L3:
        SUB r5 32 1
        MUL r5 r5 2
        STR r5 b
L4:
L4:
        MOV r1 20
        STR r1 a
        CMP r0 r5
        BLE L5
        B L4
L5:
```

ARCHITECTURE OF LANGUAGE :

Following constructs will be handled by the mini-compiler :

- Data Types: int and boolean.
- Comments: Single line and multiline comments.
- Keywords: else,while,static,public,private,protected, do while,if-else, return, break,continue ,void, signed,unsigned ,main ,extends, String and boolean.
- Identification of valid identifiers used in the language.
- Looping Constructs: It will support do-while and while loops.
- Conditional Constructs: if-else statements.
- Semantic errors : Variables used in the program but not declared / initialised, type mismatch (e.g : int a = True)

LITERATURE SURVEY :

http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0068.pdf

https://www.isi.edu/~pedro/Teaching/CSCI565-Spring14/Materials/Backpatching.pdf

http://user.it.uu.se/~kostis/Teaching/KT1-11/Slides/lecture05.pdf

https://www.csie.ntu.edu.tw/~b93501005/slide5.pdf

http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2012/05-intermediatecode-2012-2013.pdf

https://enggsolution.co.in/DBATU/E-books%20and%20Notes/Third%20Year/Computer%20Science%20Engineering/Compiler%20Design1/Moodle-Compiler%20Design/UNIT%206/c2.pdf

CONTEXT FREE GRAMMAR :

```
program_unit                    : CLASS id '{' BODY  '}'
                                | access_specifier CLASS id '{' BODY   '}'
                                | error CLASS
                                ;


BODY                            : access_specifier STATIC type_const MAIN
                                   '('type_const '[' ']' id ')' '{' STMT '}'
                                |  error MAIN
                                ;


STMT                            : E ';' STMT
                                | ASSIGNEXPR STMT
                                | DECL STMT
                                | JUMP_STMT STMT
                                | PRNT STMT
                                | IF_CONDITION STMT
                                | LOOP STMT
                                | DO_WHILE STMT
                                | ';' STMT
                                | error STMT
                                |
                                ;

PRNT                            : PRINT '('PRNT_STAT  ')' ';'

PRNT_STAT                       : string
                                | string T_PLUS PRNT_STAT
                                | id
                                | id T_PLUS PRNT_STAT
                                |
                                ;
```

```
LOOP                                    : WHILE  '{' STMT  '}' ;


DO_WHILE                                : DO  '{' STMT  '}' WHILE '(' COND ')' ';' ;


IF_CONDITION                            : IF '(' COND ')' '{' STMT '}' ELSE_ELIF;
ELIF                                    : ELSEIF '(' COND ')' '{' STMT '}' ELSE_ELIF ;
IF_ELSE                                 : T_ELSE '{' STMT '}' ;

ELSE_ELIF                               : ELIF
                                        | IF_ELSE
                                        |
                                        ;


COND                                     : EXPR
                                         | EXPR LOGOP EXPR_E
                                         | true
                                         | false
                                         | error ';'
                                         ;


EXPR_E                                  : E
                                        | EXPR
                                        ;


JUMP_STMT                               : CONTINUE ';'
                                        | BREAK ';'
                                        | RETURN E ';'
                                        ;

EXPR                                    : RELEXP
                                        | LOGEX
                                        ;

RELEXP                                  : E  rel_const E
                                        | E eq_const  E
                                        ;


LOGEXP                                  : E LOGOP E
                                        ;
```

```
LOGOP                     : or_const
                          | and_const
                          ;


DECL                      : TYPE VARLIST ';'
                          | TYPE ASSIGNEXPR
                          ;

TYPE                      : type_const
                          ;

VARLIST                   : VARLIST ',' id
                          | id
                          ;

ASSIGNEXPR                : id '=' E ',' ASSIGNEXPR
                          | id '='  E ';'
                          ;

E                         : E T_PLUS T
                          | E T_SUB T
                          | T
                          ;

T                         : T T_MUL F
                          | T T_DIV F
                          | T T_MOD F
                          | F
                          ;

F                         : id
                          | NUM
                          | true
                          | false
                          | '(' E ')'
                          ;
```

DESIGN STRATEGY :

1) SYMBOL TABLE :

Symbol table is used to store the information about the occurrence of various entities . It is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

Entries and Information in the Table :

- Scope Number
- Data Type
- Variable Name
- Line Number
- Value

2) ABSTRACT SYNTAX TREE :

An abstract syntax tree is a tree that represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator.

We are using an inorder traversal - a special case of depth-first traversal - where all nodes of a tree recursively process the left subtree, then process the root, and finally the right subtree. The operators with higher precedence end up being lower in the tree.

3) INTERMEDIATE CODE GENERATION :

A three-address code is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.

The implementation of the intermediate code generation is done by using Triples.The Triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand. In triples, the results of respective sub-expressions are denoted by the position of expression.

4) CODE OPTIMIZATION :

This phase consists of the various machine independent code optimizations techniques applied onto the Intermediate code generated (ICG). The following are the optimizations implemented by us:

1. Constant Folding
2. Constant Propagation

Constant Folding : It is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

Constant Propagation: It is the process of substituting the values of known constants in expressions at compile time.

5) ERROR HANDLING :

- Syntax Error handling : The compiler supports and implements multi-line error handling. It also has a panic mode error recovery i.e., when an error is detected instead of terminating the program the compiler continues to read the successive instructions and later prints the line number where the error has occurred and how many errors have been generated .

- Semantic Error Handling : The compiler supports and implements semantic error handling . It handles both Incompatible types of operands and Undeclared variables.

- Logical Error Handling : During a division and modulus operation , when the denominator is zero , the compiler throws a logical error.

6) TARGET CODE GENERATION :

The compiler uses a python program to perform Assembly Code Generation. The Optimized Intermediate Code is converted into assembly language following the ARM instruction set. All the lines from the input code are split and then are classified into categories based on what type of action that line of code does. Register numbers, e.g. REG0, are determined at this stage and are stored in variables, depending on the type of expression.

IMPLEMENTATION DETAILS :

1. SYMBOL TABLE :

- Data Structures used : Struct table
- Our compiler parses the code line by code , and whenever it encounters a variable declaration or assignment operation , the symbol table gets updated with the Variable Name, Line Number, Scope number, latest Value and Data Type.
- Variables will have different values in different scopes .
- Have different functions to insert variables into the symbol table ,update the values of the variables , to check if a variable is present in the Symbol table and extract values of the variables for expression evaluation.
- The ‘YYLVAL’ has been modified to store integer and string values . Here, we used ‘ival’ to store integer values and ‘st’ to store boolean values.

2. ABSTRACT SYNTAX TREE :

- Data Structures used : Binary Tree and graphs
- An inorder traversal of the tree is implemented to print out the tree.
- The root node prints the operator or constructs and the children nodes will print the variables or values (i.e., the operands)
- Pointers are used to connect parent and child nodes
- Recursion is used to print the graph.
- Different structure arrays are defined for different data types. The values are stored in the respective structure array and extracted later for printing the graph.
- To print the datatype or operation, a switch case is being used.

3. INTERMEDIATE CODE GENERATION :

- Data structures used : Stacks
- When an assignment operation is encountered , the operators and operands are pushed into the stack.
- To print the expression, the values are popped from the stack.
- Since the format used to represent the intermediate code is the 3 address triple format, temporary variables are created when required.
- Several utility functions were made to define the format for different kinds of statements.

Backpatching

An important concept used for developing the intermediate code is the backpatch function. Let Z-> Y1 …. Yn be a production. Consider the case where for a grammar symbol Yi in the right side of the production possesses an inherited attribute which cannot be given a value before all the nodes of the parse tree are completely visited. This is where backpatching helps. When translating forward jumps at the time we generate code we do not know the address of the label we want to branch to.

The solution is to generate a sequence of branching statements where the addresses of the jumps are temporarily left unspecified.

3. ‘lab()’,’lab2()’ and ’lab3()’ functions are used to implement Backpatching.
4. ‘lab()’ function is used for printing the Label number , lab3() function branches to the true section of the code and lab2() function branches to false section

4. CODE OPTIMIZATION :

- C language is used for the implementation of this phase of the compiler.
- ICG is the input for this phase.
- Techniques used for this phase are:
    1. Constant Folding : On parsing the ICG , if any expression is encountered ,it first checks if both the operands are constants and then the expression gets evaluated .
    2. Constant Propagation : On parsing the ICG , if the variable is not present in the table ,it gets inserted into the table . On expression evaluation, if the expression has at least one operand as a variable then it extracts the value from the table and later constant folding is applied to these substituted values.

5. ASSEMBLY CODE GENERATION :

- The optimized intermediate code is passed as input to a python script to get the final target code.
- Several helper functions were used for different cases such as:
    1. begin() : This function evaluates the expressions, assignment operations, JUMP statements, labels by converting the icg to relevant assembly code.
    2. register() : This function checks if the variable has been assigned to a register , if it’s present it gets added to the end of the list just like a stack operation (added to the top) and if it's absent, a register is assigned to the variable . If all registers are used then it uses the least recently used register (LRU).
- Register values were kept track of using utility functions.

6. ERROR HANDLING :

- Error keyword generation in the yacc phase to prompt to execute yyerror without exiting the code.
- Variable redeclaration , Undeclared Variable and Type mismatch is also taken care of in the semantics phase of the compiler. For Type mismatch , we check the data type of the value and datatype of the variable in the symbol table , if it doesn’t match it throws an error. For Undeclared Variable , we assign null to the data type field for the variable in the symbol table .
- Logical Error ,operand divided by zero and modulus with zero .If the denominator (second operand) is equal to zero then it throws an error.

INSTRUCTIONS TO RUN THE COMPILER :

“**bash build.sh**” . This bash file consists of all the commands required to run each phase of the compiler.

The contents of the bash file are:

```
# Phase 1 & 2
cd symTab
lex project.l
yacc -d project.y
gcc y.tab.c lex.yy.c -ll -ly
cd ..
./symTab/a.out < code.java


 # Phase 3
cd AST
lex ast.l
yacc -d  ast.y
gcc lex.yy.c y.tab.c graph.c -ll -ly -o ast.out
cd ..
./AST/ast.out < code.java

 # Phase 4
echo $'\n\n'
echo "ICG"
cat icg.txt


 # Phase 5
echo $'\n\n'
echo "Optimized Code"
gcc optimized_code.c
./a.out > optimized_code.txt
cat optimized_code.txt


# Phase 6
echo $'\n\n'
echo "Target Code"
python assemble.py
```

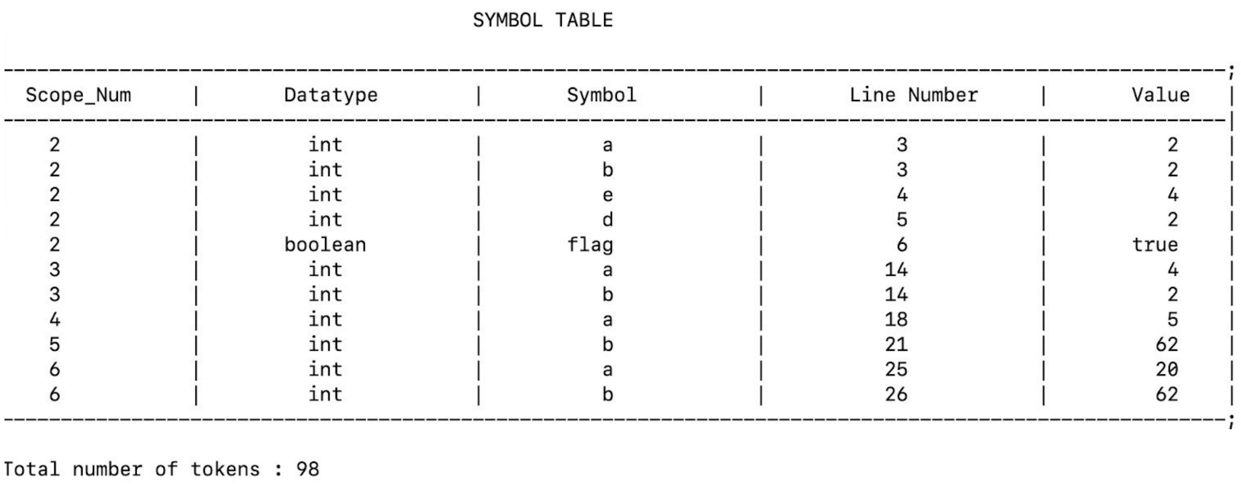
Results and Shortcomings of the Mini Compiler :

The mini compiler works as any other compiler would, on the provided constraints. It is able to efficiently compile input Java code, while catching any error that might be present in the source code, be it runtime, syntactic or semantic. In the first phase we were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types,values,scope number and line of declaration in the form of a table. The parser generates error messages in case of any syntactical errors in the test program or any semantic error or logical errors . Finally,this code is still in high level form and needs to be converted to machine level code for the computer to understand. The compiler converts the above obtained optimised ICG to machine level code as the final output.

It is not capable of handling each and every construct of the Java language as it currently stands, and will require significant changes to do so. We have not considered various data types such as String ,float ,double nor char in our grammar. We have also not taken care of functions, objects and input from the user (Scanner input) in our grammar .It is not a very robust compiler like the actual Java compilers but does well for the constructs it considers.

SNAPSHOTS :

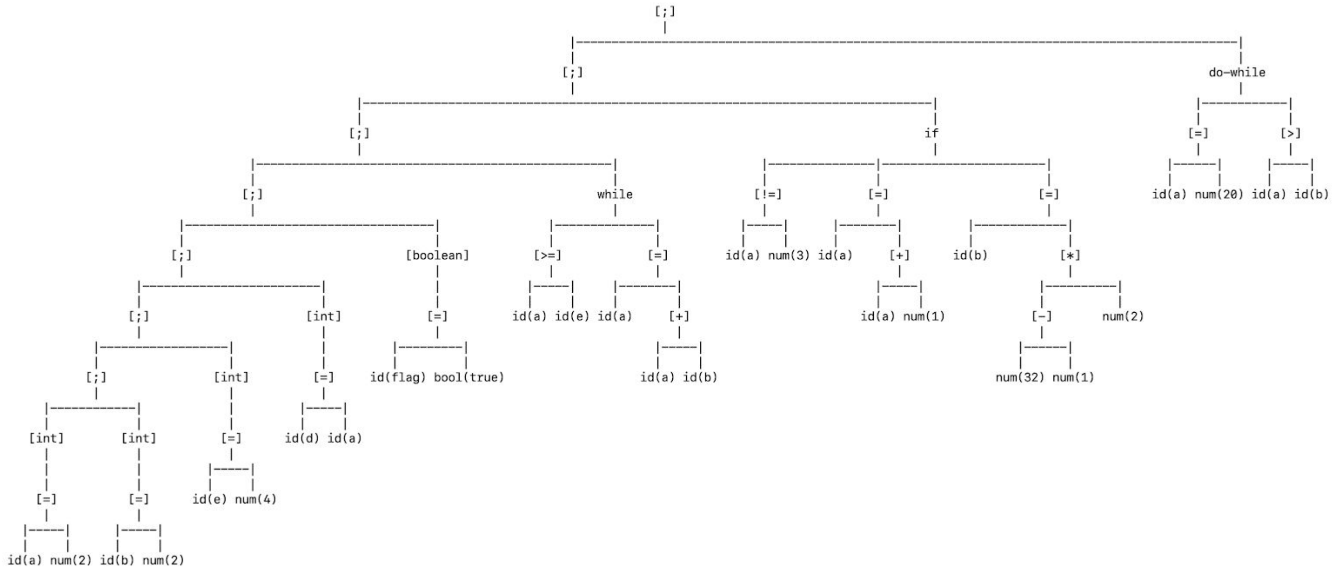
SYMBOL TABLE :



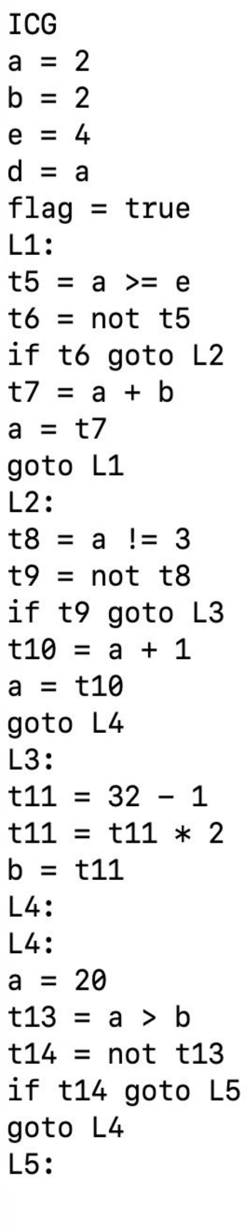
```
                                        SYMBOL TABLE

-----------------------------------------------------------------------------------------------------------;
  Scope_Num      |         Datatype          |          Symbol          |         Line Number       |          Value   |
-----------------------------------------------------------------------------------------------------------|
    2            |           int             |            a             |              3            |            2     |
    2            |           int             |            b             |              3            |            2     |
    2            |           int             |            e             |              4            |            4     |
    2            |           int             |            d             |              5            |            2     |
    2            |         boolean           |          flag            |              6            |          true    |
    3            |           int             |            a             |             14            |            4     |
    3            |           int             |            b             |             14            |            2     |
    4            |           int             |            a             |             18            |            5     |
    5            |           int             |            b             |             21            |           62     |
    6            |           int             |            a             |             25            |           20     |
    6            |           int             |            b             |             26            |           62     |
-----------------------------------------------------------------------------------------------------------;

Total number of tokens : 98
```

ABSTRACT SYNTAX TREE :

Abstract Syntax Tree :

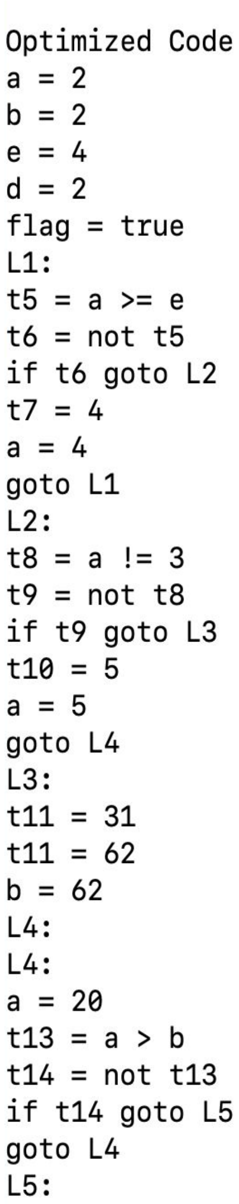


INTERMEDIATE CODE - Triples format :



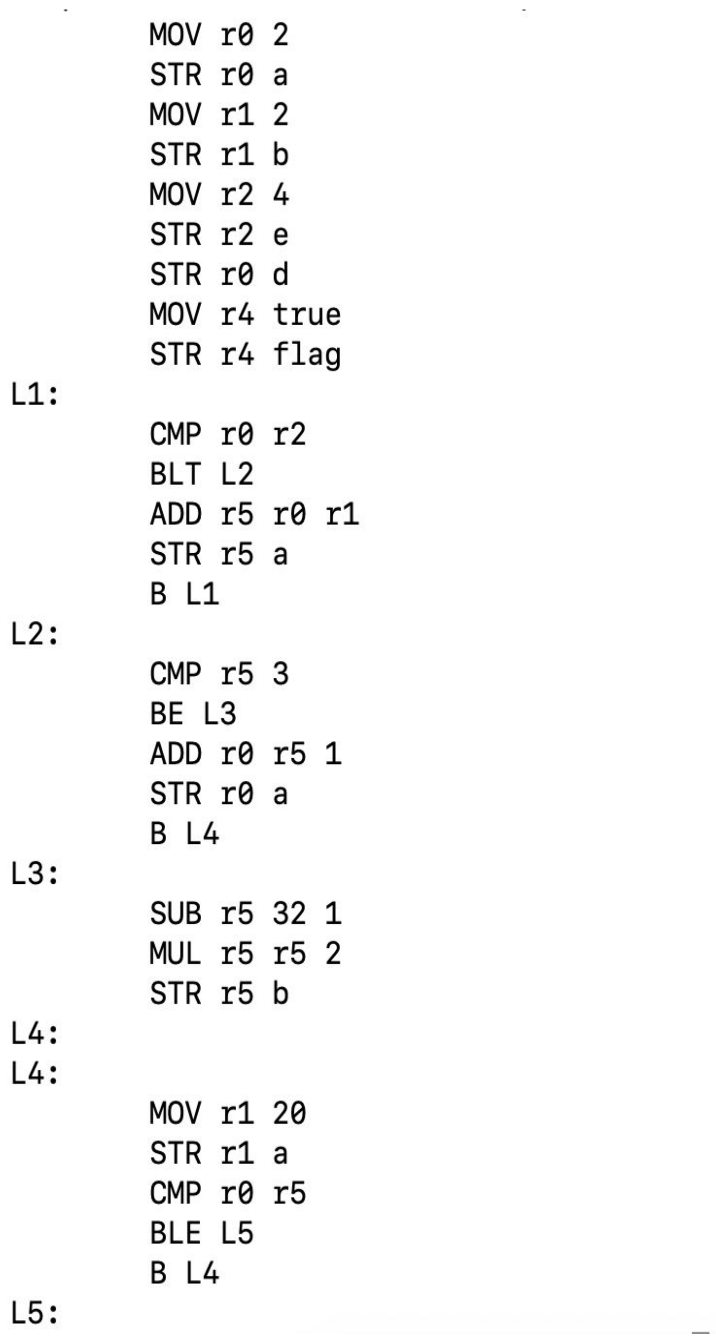
```
ICG
a = 2
b = 2
e = 4
d = a
flag = true
L1:
t5 = a >= e
t6 = not t5
if t6 goto L2
t7 = a + b
a = t7
goto L1
L2:
t8 = a != 3
t9 = not t8
if t9 goto L3
t10 = a + 1
a = t10
goto L4
L3:
t11 = 32 - 1
t11 = t11 * 2
b = t11
L4:
L4:
a = 20
t13 = a > b
t14 = not t13
if t14 goto L5
goto L4
L5:
```

OPTIMIZED CODE :



```
Optimized Code
a = 2
b = 2
e = 4
d = 2
flag = true
L1:
t5 = a >= e
t6 = not t5
if t6 goto L2
t7 = 4
a = 4
goto L1
L2:
t8 = a != 3
t9 = not t8
if t9 goto L3
t10 = 5
a = 5
goto L4
L3:
t11 = 31
t11 = 62
b = 62
L4:
L4:
a = 20
t13 = a > b
t14 = not t13
if t14 goto L5
goto L4
L5:
```

TARGET CODE :



```
        MOV r0 2
        STR r0 a
        MOV r1 2
        STR r1 b
        MOV r2 4
        STR r2 e
        STR r0 d
        MOV r4 true
        STR r4 flag
L1:
        CMP r0 r2
        BLT L2
        ADD r5 r0 r1
        STR r5 a
        B L1
L2:
        CMP r5 3
        BE L3
        ADD r0 r5 1
        STR r0 a
        B L4
L3:
        SUB r5 32 1
        MUL r5 r5 2
        STR r5 b
L4:
L4:
        MOV r1 20
        STR r1 a
        CMP r0 r5
        BLE L5
        B L4
L5:
```

CONCLUSION :

This mini compiler is built as a mini project for our Compiler Design Lab course. It takes source code of the language Java and gives machine code as the output, along with intermediate phase outputs. It is written in the language C for code optimisation and python for assembly level code. The tools used for building this are the Lex and Yacc.

FURTHER ENHANCEMENTS :

The yacc script presented in this report takes care of all the rules of Java language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of Java language and making it more efficient. We would like to develop a live variable analysis of the intermediate code and implement different data types and usage of functions .

REFERENCES :

https://www.professionalcipher.com/2017/07/intermediate-code-generation-using-lex-and-yacc-using-switch-case-and-control-flow.html

http://dinosaur.compilertools.net/bison/bison_9.html#SEC81

https://en.wikipedia.org/wiki/Optimizing_compiler

https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm

http://dinosaur.compilertools.net/lex/index.html

http://dinosaur.compilertools.net/yacc/index.html