NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI

CSPC62

COMPILER DESIGN

TOPIC : Compiler for Base Typescript

LAB REPORT - 5

<u>Sub Topic: Code Optimization</u>

DONE BY

S.No	Name	Roll No.
1.	Dhrubit Hajong	106121037
2.	Mercia Melvin Perinchery	106121077
3.	Nishith Eedula	106121085

Index

Phase 5: Code Optimization

I. Basic Blocks and the Flow Control Graph

A **basic block** refers to a simple combination of statements. There are no branches within the block like in and out, except for entry and exit. Hence, the flow of control always enters at the beginning and leaves at the end without any halt. The execution of a basic block's instructions always takes place in the form of a sequence.

A basic block begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, then the control will flow from one instruction to the next in sequential order.

The first task is to partition a sequence of three-address codes into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of a jump, control moves further consecutively from one instruction to another. The idea is standardized in the algorithm below:

Algorithm: Partitioning three-address code into basic blocks.
Input: A sequence of three address instructions.
Process: Instructions from intermediate code which are leaders are determined. The following are the rules used for finding a leader:

- The first three-address instruction of the intermediate code is a leader.
- Instructions that are targets of unconditional or conditional jump/goto statements are leaders.
- Instructions that immediately follow unconditional or conditional jump/goto statements are considered leaders.

A **Control Flow Graph (CFG)** is the graphical representation of control flow or computation during the execution of programs or applications. The control flow graph is process-oriented and shows all the paths that can be traversed during a program execution. A control flow graph is a directed graph. Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks. There exist 2 designated blocks in the CFG:

- **Entry Block:** The entry block allows the control to enter into the control flow graph.
- **Exit Block:** Control flow leaves through the exit block.

Code to Determine the Basic Blocks and the Control Flow Graph

1. Define different types of instructions that can be generated during intermediate code generation, in order to use this information to identify leaders in the algorithm described above.

```
enum InstructionType {
    INSTRUCTION,
    BRANCH,
    JUMP,
    LABEL
};
```

2. Define define a basic block as a structure that contains a start and an end index. Everything in between these two indices is part of the same basic block and will be executed in a continuous manner.

```
struct BasicBlock {
    int start;  // Start index of the basic block
    int end;    // End index of the basic block
};
```

3. Identify basic blocks by iterating over the intermediate code produced and checking for leaders.

```
void genBasicBlock()
{
      findLeaders();
      map<ll,ll> lineToBlock;
      ll bbno=-1;
      for(map<ll,string>::iterator
it=instList.begin();it!=instList.end();it++)
      {
            if(leaders.find(it->first)!=leaders.end())
            {
                  bbno++;
            }
            lineToBlock[it->first]=bbno;
      }
      bbno=-1;
      for(map<ll,string>::iterator
it=instList.begin();it!=instList.end();it++)
      {
            if(leaders.find(it->first)!=leaders.end())
            {
                  bbno++;
            }
            int ind=-1;
            string bbinst=it->second;
            if((ind=it->second.rfind("goto"))!=-1)
            {
                  bbinst=it->second.substr(0,ind+5) +
to_string(lineToBlock[to_int(it->second.substr(ind+5))]);
            }
            basicBlock[bbno].push_back(bbinst);
      }}
```

4. Print the basic blocks and their control flow in the int main() function of the parser.

```
printf("Basic Blocks:\n");
    for (int j = 0; j < blockCount; j++) {
           printf("Block %d: Start = %d, End = %d\n", j+1, blocks[j].start,
blocks[j].end);
    }

    // Print the control flow between basic blocks
    printf("\nControl Flow:\n");
    for (int j = 0; j < blockCount; j++) {
        printf("Block %d -> Block %d\n", j+1, j+2);
    }
```

Input Program

```
function dummy1(apr: number, prd: number){
    let ap: string = "p";
    apr = apr + 5;
    return ap;
}

function dummy2(){
    let a: number = 0;
    let b: number = 0;
    let p: number = 3 + 5;
    dummy1(a,b);
    let d: number = 5 + 0 ;

    if(a>=d) {
        int r = 5;
    }
    else {
        int r3 = 5;
        console.log("Abc");
    }
    d = d + 10;
    return a;
}
```

Output

```
Intermediate Code Generated is:
```

```
Block 1:
ap = "p"
t0 = apr + 5
apr = t0

Block 2:
a = 0
b = 0
t1 = 3 + 5
p = t1

Block 3:
t2 = 5 + 0
d = t2

Block 4:
if (a >= d) GOTO L0 else GOTO L1

Block 5 (L0):
r = 5

Block 6 (L1):
r3 = 5

Block 7:
t3 = d + 10
d = t3
Func-Call: dummy2

Control Flow:
Block 1 -> Block 2
Block 2 -> Block 3
Block 3 -> Block 4
Block 4 -> Block 5 (if true)
Block 4 -> Block 5 (if false)
Block 5 -> Block 7
Block 6 -> Block 7
Block 7 -> Block 2
```

II. Optimization Techniques

The Code Optimization phase, in the front end of the compiler (machine-independent), is a program transformation technique. Since it reduces readability and adds code used to increase the performance, it is performed at the end of the development stage. An optimizing compiler tries to minimize or maximize come attributes of the executable computer program. Most commonly, it aims to minimize the program's execution time, memory footprint, storage size and power consumption. This is generally implemented using a series of optimizing transformations wherein algorithms take a program and transform it to produce a semantically equivalent intermediate/output program the uses fewer resources and/or executes faster.

Some optimization problems have been shown to be NP-Complete or even undecidable. As such, this process is generally very CPU and memory-intensive. Hence, the compiler optimizing process should meet the following objectives:

- The meaning of the program should remain unchanged.
- The overall speed and performance of the program should increase.
- The compilation time must be reasonable.
- The optimization process should not delay the overall compiling process.

The machine independent optimization that we have implemented attempts to improve the **intermediate code** (three-address code) to obtain a better target code as the output. Thus, the part of the intermediate code transformed does not involve any CPU registers or absolute memory locations.

a. Compile Time Evaluation

The value of some variable is determined during compile time, as opposed to during the runtime. This reduces the corresponding assembly/target code instructions that will be required to compute such an expression during execution.

```
A = 2*(22.0/7.0)*r
>> Perform 2*(22.0/7.0)*r at compile time.
x = 12.4
y = x/2.3
>> Evaluate x/2.3 as 12.4/2.3 at compile time.
```

b. Variable Propagation

It refers to the internal processes of analyzing the flow of variables throughout the code enabling the optimizer in its optimizations. Analyzing the flow of variables refers to understanding how, what and where variables are used and modified and whether or not they are modified and have been used in the code anywhere. This helps determine what code is redundant and which variables can be replaced, so that the optimizer can perform other optimizations like constant propagation, constant folding etc.

```
Before Optimization
c = a * b
x = a
d = x * b + 4
========================================================================
After Optimization
c = a * b
x = a
d = a * b + 4
```

c. Constant Propagation

It is the process of substituting the constant value of variables in the expression. Constants assigned to a variable are propagated through the flow graph and can be replaced wherever the variable is used. This process is executed using the reaching defintion analysis results in compilers i.e. if reaching definition of all variables have the same assignment which assigns the same constant to the variable, then that variable has a constant value and can be substituted with the constant.

Constant propagation reduces the number of cases wherein values are directly copied from one location or variable to another, to simply allocate their value to another variable.

```
Before Constant Propagation
a = 30
b = 20 - a /2
c = b * ( 30 / a + 2 ) -  a
========================================================================
After Constant Propagation
a = 30
b = 20 - 30/2
c = b * ( 30 / 30 + 2) - 30
```

d. Constant Folding

Constant folding is an optimization technique that recongises and evaluates constant expressions at compile time rather than computing them at runtime. Expressions generating a constant value are evaluated and later calculated and stored in the designated variables during compilation time.

Note that in Constant Propagation, the variable is substituted with its assigned constant whereas in Constant Folding, the variables whose values are expressions which can be computed at compile time are determined and computed.

```
Before Constant Propagation
a = 30
b = 20 - a /2
c = b * ( 30 / a + 2 ) -  a
========================================================================
After Constant Propagation
a = 30
b = 20 - 30/2
c = b * ( 30 / 30 + 2) - 30

========================================================================
1. After Constant Folding
a = 30
b = 5
```

```
c = 5 * ( 30 / 30 + 2) - 30

2. After Constant Folding
a = 30
b = 5
c = - 20
```

e. Copy Propagation

It is used to replace the occurrence of target variables that are the direct assignments with their values. This is related to the approach of a common subexpression, in which the expression values are unchanged after the first expression is computed. Copy propagation aims to reduce the unnecessary expression variables, which results in faster execution and less memory utlization. It can be used only when the exact value of the variable is known at the time of compilation, and the value can be inferred from the context of the given program.

```
Before Copy Propagation
a = 1 + 2
b = a
c = b + 6

========================================================================
After Copy Propagation
a = 1 + 2
c = a + 6
```

f. Common Subexpression Elimination

This compiler optimization technique searches for instances of identical expressions or sub-expressions (i.e. those that evaluate to the same value), and analyzes whether it is worthwhile to replace them with a single variable holding the computed value. As an advantage, it makes the computation faster and better by avoiding the re-computation of the expression, and utlizes the memory efficiently. Based on available expression analysis (data flow analysis), an expression is said to be available at any point p in a program if:

- every path from the initial node to p evaluates b*c before reaching p,
- and there are no assignments to b or c after the evaluation but before p.

The two types of elimination methods in common sub-expression elimination are:
1. Local Common Sub-expression elimination– Used within a single basic block.
2. Global Common Sub-expression elimination– Used for an entire procedure of common sub-expression elimination.

```
Before Common Subexpression Elimination
a = 10
b = a + 1 * 2
c = a + 1 * 2
```

```
d = c + a
========================================================================
After Common Subexpression Elimination
a = 10
b = a + 1 * 2
d = b + a
```

g. Dead Code Elimination

Dead Code Elimination is an optimization technique that removes code which does not affect the program's results. This reduces the program size and allows the running program to avoid executing any irrelevant operations, thereby reducing the overall execution time and complexity. It also simplifies program structure and reduces the resource usage, improving the program's efficiency, readability, and maintainability. Dead code includes code that can never be executed (unreachable code), and code that only affects dead variables (written to, but never read again), that is, irrelevant to the program.

```
Before Dead Code Elimination
c = a * b
x = a
till
d = a * b + 4
========================================================================
After Dead Code Elimination
c = a * b
till
d = a * b + 4
```

h. Induction Variable and Strength Reduction

An induction variable is a variable used in a loop to control its iteration. These variables are typically first checked against the termination condition, and depending on the value, the control goes inside the loop for further execution and the induction variable is incremented or decremented. These variables lead to increased time complexity, as the compiler can run the loop only a limited number of times.

Strength Reduction is an optimization technique to optimize loop induction variables, wherein expensive arithmetic operations like multiplication and division are replaced by cheaper operations like bit shifting, addition or subtraction.

```
Before Strength Reduction
a = 1 + 2
b = a
c = b + 6

========================================================================
After Strength Reduction
a = 1 + 2
c = a + 6
```

III. Python Code for Implementing the Optimizer for Three-Address Code

```
import re
import sys

def isTemporary(s):
    return bool(re.match(r"^t[0-9]*$", s))


def isIdentifier(s):
    return bool(re.match(r"^[A-Za-z][A-Za-z0-9_]*$", s))


def showICG(allLines):
    for line in allLines:
        print(line.strip())


def createSubexpressions(allLines):
    expressions = {}
    variables = {}
    for line in allLines:
        tokens = line.split()
        if len(tokens) == 5:
            if tokens[0] in variables and variables[tokens[0]] in expressions:
                print(tokens[0], variables[tokens[0]], expressions[variables[tokens[0]]])
                del expressions[variables[tokens[0]]]
            expressionRHS = tokens[2] + " " + tokens[3] + " " + tokens[4]
             if expressionRHS not in expressions:
                expressions[expressionRHS] = tokens[0]
                if isIdentifier(tokens[2]):
                    variables[tokens[2]] = expressionRHS
                if isIdentifier(tokens[4]):
                    variables[tokens[4]] = expressionRHS
    return expressions


def eliminateCommonSubexpressions(allLines):
    expressions = createSubexpressions(allLines)
    updatedAllLines = allLines[:]
    for i in range(len(allLines)) :
        tokens = allLines[i].split()
        if len(tokens) == 5 :
            expressionRHS = tokens[2] + " " + tokens[3] + " " + tokens[4]
            if expressionRHS in expressions and expressions[expressionRHS] != tokens[0]:
                               updatedAllLines[i]  =  tokens[0]  +  "  "  +  tokens[1]  +  "  "  +
expressions[expressionRHS]
    return updatedAllLines
```

```
def evaluateExpression(expression) :
    tokens = expression.split()
    if len(tokens) != 5 :
        return expression
     acceptedOperators = {"+", "-", "*", "/", "*", "&", "|", "^", "==", ">=", "<=", "!=", ">",
"<"}
    if tokens[1] != "=" or tokens[3] not in acceptedOperators:
        return expression
    if tokens[2].isdigit() and tokens[4].isdigit() :
               return  "  ".join([tokens[0],  tokens[1],  str(eval(str(tokens[2]  +  tokens[3]  +
tokens[4])))])
     if tokens[2].isdigit() or tokens[4].isdigit() : #Replace the identifier with a number and
evaluate
        op1 = "5" if isIdentifier(tokens[2]) else tokens[2]
        op2 = "5" if isIdentifier(tokens[4]) else tokens[4]
        op = tokens[3]
        try :
            result = int(eval(op1+op+op2))
            if result == 0 : #multiplication with 0
                return " ".join([tokens[0],tokens[1], "0"])
            elif result == 5 : #add zero, subtract 0, multiply 1, divide 1
                if isIdentifier(tokens[2]) and tokens[4].isdigit() :
                    return " ".join([tokens[0], tokens[1], tokens[2]])
                elif isIdentifier(tokens[4]) and tokens[2].isdigit():
                    return " ".join([tokens[0], tokens[1], tokens[4]])
            elif result == -5 and tokens[2] == "0" : # 0 - id
                return " ".join([tokens[0], tokens[1], "-"+tokens[4]])
            return " ".join(tokens)
        except NameError :
            return expression
        except ZeroDivisionError :
            print("Division By Zero!")
            quit()
    return expression


def constantFolding(allLines) :
    updatedAllLines = []
    for line in allLines :
        updatedAllLines.append(evaluateExpression(line))
    return updatedAllLines

def strengthReductionAndInductionVariableOptimization(lines, variables):
    for index, line in enumerate(lines):
        tokens = line.split()
        if len(tokens) == 5 and tokens[3] == '*' and (tokens[4] == '2' or tokens[2] == '2'):
            variable = tokens[2] if tokens[4] == '2' else tokens[4]
            if variable in variables:
                lines[index] = f"{tokens[0]} = {variable} + {variable}"
    return lines

def optimizeUsingNextUseInfo(lines, variables):
    # Simple next use optimization: Remove lines where the variable is not used again.
    usage = {var: i for i, var in enumerate(reversed(variables), 1)}
    optimized_lines = []
    for line in lines:
        tokens = line.split()
        if len(tokens) >= 3 and tokens[0] in usage:
            optimized_lines.append(line)
    return optimized_lines


def deadCodeElimination(allLines):
    definedTempVars = set()
    usedTempVars = set()

    for line in allLines:
        tokens = line.split()

        if not tokens:
            continue
        if isTemporary(tokens[0]):
            definedTempVars.add(tokens[0])
        for token in tokens[1:]:
            if isTemporary(token) and token in definedTempVars:
                usedTempVars.add(token)

    liveTempVars = usedTempVars.copy()
    additionalScan = True

    while additionalScan:
        additionalScan = False
        for line in reversed(allLines):
            tokens = line.split()
            if not tokens:
                continue
             # If the output of the line is a temp var that contributes to a live temp var, mark
its inputs as live
            if tokens[0] in liveTempVars:
                for token in tokens[2:]:  # Skip the '=' operator at tokens[1]
                    if isTemporary(token) and token not in liveTempVars:
                        liveTempVars.add(token)
                          additionalScan = True

    updatedAllLines = []
    for line in allLines:
        tokens = line.split()
        if not tokens:
            continue
         if tokens[0] in liveTempVars or not isTemporary(tokens[0]):
             updatedAllLines.append(line)

    return updatedAllLines

if __name__ == "__main__":
     with open("output_file.txt","r") as inFile, open("output_file_final.txt", "w") as
outFile:
        for line in inFile:
            if not line.isspace():
                outFile.write(line)

    allLines = []
    f = open("output_file_final.txt", "r")
    for line in f:
        allLines.append(line)
    f.close()

    print("\n")

    icgElimCommonSubexpr = eliminateCommonSubexpressions(allLines)
    icgConstFold = constantFolding(icgElimCommonSubexpr)
        variables = set()
    for line in icgConstFold:
        tokens = line.split()
        if len(tokens) >= 3:
            variables.add(tokens[0])
    variables=list(variables)
    icgStrRed   =   strengthReductionAndInductionVariableOptimization(   icgConstFold,
variables)
        icgNextUseInfo = optimizeUsingNextUseInfo(icgStrRed, variables)
        icgAfterDeadCodeElimination = deadCodeElimination(icgAfterConstantFolding)
    print("Optimized ICG: \n")
    showICG(icgAfterDeadCodeElimination)
```

IV. YACC Code for Implementing the Syntax Directed Translations to the Three Address Code

```
%{
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include <ctype.h>
    #include <math.h>
    #include "lex.yy.c"

    #define BOLDRED "\033[1m\033[31m"
    #define BOLDBLUE "\033[1m\033[34m"
    #define BOLDCYAN "\033[1m\033[36m"
    #define BOLDYELLOW "\033[1m\033[33m"
    #define BOLDMAGENTA "\033[1m\033[35m"
    #define BOLDGREEN "\033[1m\033[32m"
    #define RESET "\033[0m"

    int yyerror(const char *s);
    int yylex(void);
    int yywrap();
    float get_value(char *var);
    void set_value(char *var, float value);

    struct node *head;
    struct node {
        struct node *left;
        struct node *right;
        char *token;
        float value;
    };

     struct node* mknode(struct node *left, struct node *right, char *token, float
value);
    void printBT(struct node*);

    struct dataType {
        char *id_name;
        char *data_type;
        char *type;
        int   line_no;
        float  value;
    } symbol_table[100];

    char type[100]; // Added declaration for type
      char id_name[100];
    int count = 0; // Added declaration for count
    int q; // Added declaration for q
    int sem_errors = 0;

    int label = 0;
      int ic_idx = 0;
    int param_idx = 0;
      int temp_var = 0;
      int is_loop = 0;
      int is_array = 0;
      int temp_arr = 0;

    char buff[400];
    char errors[10][100];
      char reserved[10][20] = {"number", "import", "async", "string", "void", "if",
"else", "for", "while", "return"};
    char icg[100][200];

    void insert_type();
      void store_name();
    void add(char c);
    int search(char *);

    void printBTHelper(char* prefix, struct node* ptr, int isLeft);
    void printBT(struct node* ptr);
    void printSymbolTable();
      void printSemanticErrors();
      void printIntermediateCode();

    int check_declaration(char *);
    void check_return_type(char *);
    int check_types(char *, char *);
    char *get_type(char *);
      void insert_type();

    extern int countn;
```

```
%}


%union {
      struct var_name {
            char name[100];
            struct node* nd;
        float value;
      } treeNode;

    struct var_name2 {
        char name[100];
        struct node* nd;
        int temp_arr;
        char type[10];
        float value;
    } treeNode2;

      struct var_name3 {
            char name[100];
        char type[10];
        int temp_arr;
            struct node* nd;
            char if_body[5];
            char else_body[5];
            char after_else_body[5];
        float value;
        int tlist[10];
        int tlistsize;
        int flistsize;
        int flist[10];
        int label_for_while_start;
      } treeNode3;

      struct var_name4 {
        int next_quad;
      } treeNode4;
}

%token <treeNode>  IMPORT FROM AS CONSOLELOG SCAN IF WHILE ELSE RETURN ELIF LET VAR
CONST ADD SUB MULT DIV LOG GE LE GT LT EQ NE TRUE FALSE AND OR NUMBERTYPE STRINGTYPE
BOOLEANTYPE NUMBERARRAYTYPE STRINGARRAYTYPE BOOLEANARRAYTYPE FUNCTION INTEGER FLOAT
STRINGVALUE INC DEC FOR
%token <treeNode2> POW IDENTIFIER
%type  <treeNode>  main importList imports moduleList modules parameter parameterList
argument argumentList datatype body block console_outputs else statement declaration
mulops addops relop return and_or list integerList stringList
%type  <treeNode2> init expression value number term factor base exponent function
procedure array assign
%type  <treeNode3> condition
%type  <treeNode4> M
%define parse.error verbose
```

```
%%

main: importList body { $$.nd = mknode($1.nd, $2.nd, "Program", 0); head = $$.nd; }
;


importList: imports importList { $$.nd = mknode($1.nd, $2.nd, "ImportList", 0); }
| imports { $$.nd = mknode(NULL, $1.nd, "ImportList", 0); }
| { $$.nd = mknode(NULL, NULL, "ImportList", 0); }
;


imports: IMPORT '{' moduleList '}' FROM STRINGVALUE ';' { $$.nd = $3.nd; }
| IMPORT '{' modules '}' AS IDENTIFIER FROM STRINGVALUE ';' { $$.nd = $3.nd; }
| IMPORT moduleList FROM STRINGVALUE ';' { $$.nd = $2.nd; }
| IMPORT modules AS IDENTIFIER FROM STRINGVALUE ';' { $$.nd = $2.nd; }
;


moduleList: moduleList ',' moduleList { $$.nd = mknode($1.nd, $3.nd, "ModuleList", 0);
}
| modules { $$.nd = mknode(NULL, NULL, $1.name, 0); }
;


modules: IDENTIFIER
| MULT
;


body: block body {$$.nd = mknode($1.nd, $2.nd, "Scope", 0); }
| { $$.nd = mknode(NULL, NULL, "EndOfScope", 0); }
;


block: function { $$.nd = $1.nd; }
| procedure { $$.nd = $1.nd; }
| WHILE { add('K'); is_loop = 1;} '(' condition ')' {
    sprintf(icg[ic_idx++], BOLDBLUE);
    sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i = 0; i <$4.tlistsize; i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label - 1);
        strcat(icg[$4.tlist[i]], temp);
    }
} '{' body '}' {
    $$.nd = mknode($4.nd, $8.nd, $1.name, 0);
    sprintf(icg[ic_idx++], BOLDMAGENTA);
    sprintf(icg[ic_idx++], "JUMP TO L%d\n", $4.label_for_while_start);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i=0;i<$4.flistsize;i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label);
        strcat(icg[$4.flist[i]], temp);
    }
    sprintf(icg[ic_idx++], BOLDBLUE);
      sprintf(icg[ic_idx++], "\nLABEL L%d:\n",label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
}
| FOR { add('K'); is_loop = 1;}'(' statement ';' condition {
    sprintf(icg[ic_idx++], BOLDBLUE);
    sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i = 0; i <$6.tlistsize; i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label - 1);
        strcat(icg[$6.tlist[i]], temp);
    }
} ';' statement ')' '{' body '}' {
      struct node *temp = mknode($6.nd, $9.nd, "Condition", 0);
    struct node *temp2 = mknode($4.nd, temp, "Initialisation", 0);
    $$.nd = mknode(temp2, $12.nd, $1.name, 0);
    sprintf(icg[ic_idx++], BOLDMAGENTA);
    sprintf(icg[ic_idx++], "JUMP TO L%d\n", $6.label_for_while_start);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i = 0; i < $6.flistsize; i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label);
        strcat(icg[$6.flist[i]], temp);
    }
    sprintf(icg[ic_idx++], BOLDBLUE);
      sprintf(icg[ic_idx++], "\nLABEL L%d:\n",label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
}
| IF { add('K'); is_loop = 0;}  '(' condition ')' {
        sprintf(icg[ic_idx++], BOLDBLUE);
        sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
        sprintf(icg[ic_idx++], BOLDGREEN);
        for(int i = 0; i < $4.tlistsize; i++){
            char temp[40];
            sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label - 1);
            strcat(icg[$4.tlist[i]], temp);
        }
}
'{' body '}' {
    sprintf(icg[ic_idx++], BOLDMAGENTA);
    sprintf(icg[ic_idx++],"JUMP TO L%d\n", label+1);
    sprintf(icg[ic_idx++], BOLDBLUE);
    sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i=0;i<$4.flistsize;i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label-1);
        sprintf(icg[$4.flist[i]], temp);
    }
} else {
    struct node *iff = mknode($4.nd, $8.nd, $1.name,0);
    $$.nd = mknode(iff, $11.nd, "conditionalBranch",0);
    sprintf(icg[ic_idx++], BOLDBLUE);
    sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
}
| statement ';' { $$.nd = $1.nd; }
|  CONSOLELOG  {  add('K');}  '('  console_outputs  ')'  ';'  {  struct  node  *data  =
mknode(NULL, NULL, $4.name, 0); $$.nd = mknode(NULL, data, "ConsoleLog", 0); }
;


console_outputs: STRINGVALUE { add('C');}
| IDENTIFIER  { check_declaration($1.name);}
| expression  { $$.nd = $1.nd; }
;


function:  FUNCTION  {  add('F');  }  IDENTIFIER  {  add('I');  }  '('  parameterList  ')'  '{'
body return '}' {
    $9.nd = mknode($9.nd, $10.nd, "FunctionBody", 0);
    struct node *main = mknode($9.nd, $6.nd, $3.name, 0);
    $$.nd = mknode($1.nd, main, "Function", 0);
}
;

procedure: IDENTIFIER '(' argumentList ')' ';' { $1.nd = mknode(NULL, NULL, $1.name,
0);   $$.nd   =   mknode($1.nd,   $3.nd,   "FunctionCall",   0);   sprintf(icg[ic_idx++],
BOLDYELLOW);    sprintf(icg[ic_idx++],    "\nFUNCTION    CALL    %s\n",    $1.name);
sprintf(icg[ic_idx++], BOLDGREEN); }
;

argumentList:  argument  ','  argumentList  {  check_declaration($1.name);  $$.nd  =
mknode($1.nd, $3.nd, "ArgumentList", 0); }
| argument { check_declaration($1.name); $$.nd = $1.nd; }
| { $$.nd = $$.nd = mknode(NULL, NULL, "Argument", 0); }
;

argument: IDENTIFIER {store_name();} { $1.nd = mknode(NULL, NULL, $1.name, 0); $$.nd =
mknode(NULL, $1.nd, "Argument", 0); }
|  value  {  $1.nd  =  mknode(NULL,  NULL,  $1.name,  0);  $$.nd  =  mknode(NULL,  $1.nd,
"Argument", 0); }
;

parameterList:   parameter   ','   parameterList   {   $$.nd   =   mknode($1.nd,   $3.nd,
"ParameterList", 0); sprintf(icg[ic_idx++], BOLDYELLOW); sprintf(icg[ic_idx++], "PARAM
%s\n", $1.name); sprintf(icg[ic_idx++], BOLDGREEN); }
|    parameter    {    $$.nd    =    $1.nd;    sprintf(icg[ic_idx++],    BOLDYELLOW);
sprintf(icg[ic_idx++], "\nPARAM %s\n", $1.name); sprintf(icg[ic_idx++], BOLDGREEN); }
| { $$.nd = $$.nd = mknode(NULL, NULL, "Parameter", 0); }
;
```

```
parameter: IDENTIFIER {store_name();} ':' datatype {add('I');} {$4.nd = mknode(NULL, NULL, $4.name, 0); $1.nd = mknode(NULL, NULL, $1.name, 0); $$.nd = mknode($4.nd, $1.nd, "Parameter", 0);}
;


datatype: NUMBERTYPE { insert_type(); }
| STRINGTYPE { insert_type(); }
| BOOLEANTYPE { insert_type(); }
| NUMBERARRAYTYPE { insert_type(); }
| STRINGARRAYTYPE { insert_type(); }
| BOOLEANARRAYTYPE { insert_type(); }
;


else: ELSE { add('K');} '{' body '}' {  struct node *cond = mknode(NULL, NULL, "EndOfConditional", 0); $$.nd = mknode($4.nd, cond, $1.name, 0); }
| { $$.nd = NULL; }
;


M: {
    $$.next_quad = ic_idx;
    char new1[100];
    sprintf(new1, "%d:\033[1m\033[32m", ic_idx);
    char new2[100];
    sprintf(new2, "\033[1m\033[34m\nLABEL S");
    strcat(new2, new1);
    strcpy(icg[ic_idx], new2);
 };

condition: condition AND M condition {
      $$.nd = mknode($1.nd, $4.nd, "AND",0);
      for (int i = 0; i < $1.tlistsize; i++) {
        char temp[40];
        sprintf(temp, "%d\n\033[1m\033[32m", $3.next_quad);
        char temp2[100];
        sprintf(temp2,"\033[1m\033[35mGOTO S");
        strcat(temp2, temp);
        strcat(icg[$1.tlist[i]], temp2);
    }
    $$.tlistsize = 0;
    $$.flistsize = 0;
    for (int i = 0; i < $4.tlistsize; i++) {
        $$.tlist[$$.tlistsize++] = $4.tlist[i];
    }
    for (int i = 0; i < $1.flistsize; i++) {
        $$.flist[$$.flistsize++] = $1.flist[i];
    }
    for(int i=0;i<$4.flistsize;i++){
        $$.flist[$$.flistsize++] = $4.flist[i];
    }
```

```
      $$.label_for_while_start = $1.label_for_while_start;
}
| condition OR M condition {
      $$.nd = mknode($1.nd, $4.nd, "OR",0);
    for (int i = 0; i < $1.flistsize; i++) {
        char temp[40];
        sprintf(temp, "%d\n\033[1m\033[32m", $3.next_quad);
        char temp2[100];
        sprintf(temp2, "\033[1m\033[35mGOTO S");
        strcat(temp2, temp);
        strcat(icg[$1.flist[i]], temp2);
    }
    $$.tlistsize = 0;
    $$.flistsize = 0;
    for (int i = 0; i < $1.tlistsize; i++) {
        $$.tlist[$$.tlistsize++] = $1.tlist[i];
    }
    for (int i = 0; i < $4.tlistsize; i++) {
        $$.tlist[$$.tlistsize++] = $4.tlist[i];
    }
    for(int i=0;i<$4.flistsize;i++){
        $$.flist[$$.flistsize++] = $4.flist[i];
    }
      $$.label_for_while_start = $1.label_for_while_start;
}
| expression relop expression {
      $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
    char ifstt[400];
    if(is_loop) {
        $$.label_for_while_start = label;
        sprintf(icg[ic_idx++], BOLDBLUE);
        sprintf(icg[ic_idx++], "\nLABEL L%d:\n", label++);
        sprintf(icg[ic_idx++], BOLDGREEN);
        is_loop = 0;
    }

    if (is_array == 0) {
        if (strcmp($2.name, "==") == 0) {
            sprintf(ifstt, "if %s %s %s ", $1.name, "==", $3.name);
        } else if (strcmp($2.name, "==") == 0) {
            sprintf(ifstt, "if %s %s %s ", $1.name, "!=", $3.name);
        } else {
            sprintf(ifstt, "if %s %s %s ", $1.name, $2.name, $3.name);
        }
    } else {
        if (strcmp($2.name, "==") == 0) {
            sprintf(ifstt, "if u%d %s u%d ", $1.temp_arr, "==", $3.temp_arr);
        } else if (strcmp($2.name, "=!=") == 0) {
            sprintf(ifstt, "if u%d %s u%d ", $1.temp_arr, "!=", $3.temp_arr);
        } else {
            sprintf(ifstt, "if u%d %s u%d ", $1.temp_arr, $2.name, $3.temp_arr);
        }
        is_array = 0;
    }

    sprintf(ifstt, "\nif %s %s %s\n", $1.name, $2.name, $3.name);
    strcat(icg[ic_idx++], ifstt);
    $$.tlistsize = 0;
    $$.flistsize = 0;
    $$.tlist[$$.tlistsize++] = ic_idx - 1;
    $$.flist[$$.flistsize++] = ic_idx++;
}
| '(' condition ')' {
    $$.nd = $2.nd;
    $$.tlistsize = $2.tlistsize;
    $$.flistsize = $2.flistsize;
    for(int i=0;i<$2.tlistsize;i++){
        $$.tlist[i] = $2.tlist[i];
    }
    for(int i=0;i<$2.flistsize;i++){
        $$.flist[i] = $2.flist[i];
    }
      $$.label_for_while_start = $2.label_for_while_start;
}
| value { $$.nd = $1.nd;}
| TRUE { add('K');} {$$.nd = NULL; }
| FALSE { add('K');} {$$.nd = NULL; }
;


statement: declaration IDENTIFIER { store_name(); } ':' datatype {add('I');} init {
    $5.nd = mknode(NULL, NULL, $5.name, 0); //making for datatype
      $2.nd = mknode(NULL, NULL, $2.name, 0); //making for identifier
      $1.nd = mknode($5.nd, $2.nd, $1.name, 0); //making for the declaration
      int t = check_types($5.name, $7.type); //here we're checking types
      if(t>0) {
             if(t == 1) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration -
number and string\n", countn+1);
                    sem_errors++;
        }
             else if(t == 2) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration -
string and number\n", countn+1);
                    sem_errors++;
        }
             else if(t == 3) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration -
number and boolean\n", countn+1);
                    sem_errors++;
        }
             else if(t == 4) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration -
boolean and number\n", countn+1);
                    sem_errors++;
        }
```

```
             else if(t == 5) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration - boolean and string\n", countn+1);
                    sem_errors++;
        }
             else if(t == 6) {
                 sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration - string and boolean\n", countn+1);
                    sem_errors++;
        }
             else {
                    sprintf(errors[sem_errors], "Line %d: Type Mismatch in Declaration\n", countn+1);
                    sem_errors++;
             }
       }
       else {
             $$.nd = mknode($2.nd, $7.nd, "Initialisation", 0);

       }
    $2.value = $7.value;
    $2.nd->value = $7.value;
    set_value($2.name, $7.value);
       sprintf(icg[ic_idx++], "%s = %s\n", $2.name, $7.name);
}
| IDENTIFIER '=' assign {
       if(check_declaration($1.name)) {
        if(is_array == 0) {
            $1.nd = mknode(NULL, NULL, $1.name, 0);
            char *id_type = get_type($1.name);
            if(strcmp(id_type, $3.type)) {
                int t =  check_types(id_type,$3.type);
                if(t>0) {
                    if(t == 1) {
                              sprintf(errors[sem_errors], "Line %d: Type Mismatch in Assignment - number and string\n", countn+1);
                        sem_errors++;
                    }
                    else if(t == 2) {
                              sprintf(errors[sem_errors], "Line %d: Type Mismatch in Assignment - string and number\n", countn+1);
                        sem_errors++;
                    }
                    else if(t == 3) {
                              sprintf(errors[sem_errors], "Line %d: Type Mismatch in Assignment - number and boolean\n", countn+1);
                        sem_errors++;
                    }
                    else if(t == 4) {
                              sprintf(errors[sem_errors], "Line %d: Type Mismatch in Assignment - boolean and number\n", countn+1);
                        sem_errors++;
                    }
                    else if(t == 5) {
                                sprintf(errors[sem_errors], "Line %d: Type Mismatch in
Assignment - boolean and string\n", countn+1);
                        sem_errors++;
                    }
                    else if(t == 6) {
                                sprintf(errors[sem_errors], "Line %d: Type Mismatch in
Assignment - string and boolean\n", countn+1);
                        sem_errors++;
                    }
                }
            }
            else {
                $$.nd = mknode($1.nd, $3.nd, "=", 0);
                $1.value = $3.value;
                $1.nd->value = $3.value;
                set_value($1.name, $3.value);
                sprintf(icg[ic_idx++], "%s = %s\n", $1.name, $3.name);
            }
        }
        else {
            is_array = 0;
            $$.nd = mknode($1.nd, $3.nd,"=", 0);
            sprintf(icg[ic_idx++],"%s = u%d\n", $1.name,$3.temp_arr);
        }
      }
}
| array '=' expression {
    is_array = 0;
    $$.nd = mknode($1.nd, $3.nd, "=", 0);
    sprintf(icg[ic_idx++], "u%d = %s\n", $1.temp_arr, $3.name);
}
| array '=' array {
    is_array = 0;
    $$.nd = mknode($1.nd, $3.nd, "=", 0);
    sprintf(icg[ic_idx++], "u%d = u%d\n", $1.temp_arr, $3.temp_arr);
}
| IDENTIFIER relop expression {
      if(check_declaration($1.name)) {
             char *id_type = get_type($1.name);
             if(strcmp(id_type, $3.type)) {
                    int t =  check_types(id_type,$3.type);
                    if(t>0) {
                    if(t == 1) {
                           sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - number and string\n", countn+1);
                           sem_errors++;
                    }
                    else if(t == 2) {
                           sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - string and number\n", countn+1);
                           sem_errors++;
                    }
                    else if(t == 3) {
                          sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - number and boolean\n", countn+1);
                          sem_errors++;
                    }
                    else if(t == 4) {
                            sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - boolean and number\n", countn+1);
                           sem_errors++;
                    }
                    else if(t == 5) {
                          sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - boolean and string\n", countn+1);
                          sem_errors++;
                    }
                    else if(t == 6) {
                          sprintf(errors[sem_errors],  "Line  %d:  Type  Mismatch  in
Relational Operation - string and boolean\n", countn+1);
                          sem_errors++;
                    }
         }
      }
      $1.nd = mknode(NULL, NULL, $1.name, 0);
      $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
      }
}
| IDENTIFIER INC {
      if(check_declaration($1.name)){
            char* x=get_type($1.name);
            if(strcmp(x, "number") != 0) {
                  sprintf(errors[sem_errors],  "Line  %d:  Cannot  increment  type  %s
(only number)\n", countn+1, x);
                  sem_errors++;
            }
      }
      $1.nd = mknode(NULL, NULL, $1.name, 0); $2.nd = mknode(NULL, NULL, $2.name, 0);
$$.nd = mknode($1.nd, $2.nd, "Iterator", 0);
      sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $1.name, $1.name, temp_var);
      temp_var++;
}
| IDENTIFIER DEC {
      if(check_declaration($1.name)){
            char* x=get_type($1.name);
            if(strcmp(x, "number") != 0) {
                  sprintf(errors[sem_errors],  "Line  %d:  Cannot  decrement  type  %s
(only number)\n", countn+1, x);
                  sem_errors++;
            }
      }
      $1.nd = mknode(NULL, NULL, $1.name, 0); $2.nd = mknode(NULL, NULL, $2.name, 0);
$$.nd = mknode($1.nd, $2.nd, "Iterator", 0);
      sprintf(buff,"t%d  =  %s  +  1\n%s  =  t%d\n",temp_var,$1.name,$1.name,temp_var);
temp_var++;
```

```
}
| INC IDENTIFIER {
      if(check_declaration($1.name)){
            char* x=get_type($1.name);
            if(strcmp(x, "number") != 0) {
                  sprintf(errors[sem_errors],  "Line  %d:  Cannot  decrement  type  %s
(only number)\n", countn+1, x);
                  sem_errors++;
            }
      }
      $1.nd = mknode(NULL, NULL, $1.name, 0); $2.nd = mknode(NULL, NULL, $2.name, 0);
$$.nd = mknode($1.nd, $2.nd, "Iterator", 0);
      sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $2.name, $2.name, temp_var);
temp_var++;
}
| DEC IDENTIFIER {
      if(check_declaration($1.name)){
            char* x=get_type($1.name);
            if(strcmp(x, "number") != 0) {
                  sprintf(errors[sem_errors],  "Line  %d:  Cannot  decrement  type  %s
(only number)\n", countn+1, x);
                  sem_errors++;
            }
      }
      $1.nd = mknode(NULL, NULL, $1.name, 0); $2.nd = mknode(NULL, NULL, $2.name, 0);
$$.nd = mknode($1.nd, $2.nd, "Iterator", 0);
      sprintf(buff, "t%d = %s - 1\n%s = t%d\n", temp_var, $2.name, $2.name, temp_var);
temp_var++;
}
;

assign: expression {
    strcpy($$.type, $1.type);
    $$.nd = $1.nd;
}
| array {
    $$.nd = $1.nd;
    $$.temp_arr = $1.temp_arr;
}
| procedure
;

declaration: LET { add('K');}
| VAR { add('K');}
| CONST { add('K');}
;


init:  '='  value  {  $$.nd  =  $2.nd;  sprintf($$.type,  "%s",  $2.type);  strcpy($$.name,
$2.name); $$.value = $2.value; $$.nd->value = $2.value; }
|  '='  expression  {  $$.nd  =  $2.nd;  sprintf($$.type,  "%s",  $2.type);  strcpy($$.name,
$2.name); $$.value = $2.value; $$.nd->value = $2.value; }
| '=' list {
    $$.nd = mknode(NULL, NULL, "[]", 0);
    is_array = 1;
}
|  {  sprintf($$.type,  "%s",  "null");  $$.nd  =  mknode(NULL,  NULL,  "NULL",  0);
strcpy($$.name, "NULL"); }
;

integerList: INTEGER ',' integerList { $$.nd = mknode($1.nd, $3.nd, "IntegerList", 0);
sprintf(icg[ic_idx++], "%s, ", $1.name); }
| INTEGER { $$.nd = $1.nd; sprintf(icg[ic_idx++], "%s, ", $1.name); }
;


expression : expression addops term {
	if(strcmp($1.type, $3.type)){
		if(!strcmp($1.type, "number") && !strcmp($3.type, "string")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and string\n", countn+1);
			sem_errors++;
		}
		else if(!strcmp($1.type, "string") && !strcmp($3.type, "number")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and number\n", countn+1);
			sem_errors++;
		}
		else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "number")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and number\n", countn+1);
			sem_errors++;
		}
		else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "string")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and string\n", countn+1);
			sem_errors++;
		}
		else if(!strcmp($1.type, "number") && !strcmp($3.type, "boolean")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and boolean\n", countn+1);
			sem_errors++;
		}
		else if(!strcmp($1.type, "string") && !strcmp($3.type, "boolean")) {
			sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and boolean\n", countn+1);
			sem_errors++;
		}
		else{
			sprintf(errors[sem_errors],   "Line   %d:   Type   Mismatch   in
Expression.\n", countn+1);
			sem_errors++;
		}
	}
	else {
		sprintf($$.type, "%s", $1.type);
```

```
            $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
        }
        sprintf($$.name, "t%d", temp_var);
        temp_var++;
     $$.value = $1.value + $3.value;
     $$.nd->value = $$.value;
        sprintf(icg[ic_idx++], "%s = %s %s %s\n",  $$.name, $1.name, $2.name, $3.name);
}
| term { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd;
$$.value = $1.value; $$.nd->value = $1.value; }
;


term : term mulops factor {
        if(strcmp($1.type, $3.type)){
                if(!strcmp($1.type, "number") && !strcmp($3.type, "string")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and string\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "string") && !strcmp($3.type, "number")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and number\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "number")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and number\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "string")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and string\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "number") && !strcmp($3.type, "boolean")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and boolean\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "string") && !strcmp($3.type, "boolean")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and boolean\n", countn+1);
                        sem_errors++;
                }
                else{
                        sprintf(errors[sem_errors],   "Line   %d:   Type   Mismatch   in
Expression.\n", countn+1);
                        sem_errors++;
                }
        }
        else {
                sprintf($$.type, "%s", $1.type);
                $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
        }
        sprintf($$.name, "t%d", temp_var);
        temp_var++;
     $$.value = $1.value * $3.value;
     $$.nd->value = $$.value;
        sprintf(icg[ic_idx++], "%s = %s %s %s\n",  $$.name, $1.name, $2.name, $3.name);
}
| factor { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd;
$$.nd->value = $1.value; }
;


factor : base exponent base {
        if(strcmp($1.type, $3.type)){
                if(!strcmp($1.type, "number") && !strcmp($3.type, "string")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and string\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "string") && !strcmp($3.type, "number")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and number\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "number")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and number\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "boolean") && !strcmp($3.type, "string")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
boolean and string\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "number") && !strcmp($3.type, "boolean")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
number and boolean\n", countn+1);
                        sem_errors++;
                }
                else if(!strcmp($1.type, "string") && !strcmp($3.type, "boolean")) {
                        sprintf(errors[sem_errors], "Line %d: Type Mismatch in Expression -
string and boolean\n", countn+1);
                        sem_errors++;
                }
                else{
                        sprintf(errors[sem_errors],   "Line   %d:   Type   Mismatch   in
Expression.\n", countn+1);
                        sem_errors++;
                }
        }
        else {
                sprintf($$.type, "%s", $1.type);
```

```
             $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
       }
       sprintf($$.name, "t%d", temp_var);
       temp_var++;
    $$.value = pow($1.value, $3.value);
    $$.nd->value = $$.value;
       sprintf(icg[ic_idx++], "%s = %s %s %s\n",  $$.name, $1.name, $2.name, $3.name);
}
| LOG '(' value ',' value ')' {$$.nd = mknode($3.nd, $5.nd, $1.name, 0); }
| base { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd;
$$.value = $1.value; $$.nd->value = $1.value; }
;


base : value  { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd =
$1.nd; $$.value = $1.value; $$.nd->value = $1.value; }
| '(' expression ')'  { strcpy($$.name, $2.name); sprintf($$.type, "%s", $2.type);
$$.nd = $2.nd; }
;


and_or : AND { $$.nd = mknode(NULL, NULL, $1.name, 0); }
| OR { $$.nd = mknode(NULL, NULL, $1.name, 0); }
;


exponent: POW
;


mulops: MULT
| DIV
;


addops: ADD
| SUB
;


relop: LT
| GT
| LE
| GE
| EQ
| NE
;


number: INTEGER {
    strcpy($$.name, $1.name);
    sprintf($$.type, "%s", "number");
    add('N');
    $$.nd = mknode(NULL, NULL, $1.name, atoi($1.name));
    $$.value = atoi($1.name);
}
| FLOAT {
    strcpy($$.name, $1.name);
    sprintf($$.type, "%s", "number");
    add('N');
    $$.nd = mknode(NULL, NULL, $1.name, atof($1.name));
    $$.value = atof($1.name);
}
;


value: number { $$.nd = mknode(NULL, NULL, $1.name, 0); }
| IDENTIFIER {
    strcpy($$.name, $1.name);
    char *id_type = get_type($1.name);
    sprintf($$.type, "%s", id_type);
    check_declaration($1.name);
    $$.nd = mknode(NULL, NULL, $1.name, 0);
    q = search($1.name);
    if(q==-1) {
        $$.value = get_value($1.name);
        $$.nd->value = $$.value;
    }
}
| STRINGVALUE { strcpy($$.name, $1.name); sprintf($$.type, "string"); add('C'); $$.nd
= mknode(NULL, NULL, $1.name, 0); }
| TRUE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name, 0);}
| FALSE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name, 0);}
| SCAN { add('K');} '('')' { $$.nd = mknode(NULL, NULL, "scan", 0); }
| array { $$.nd = $1.nd; strcpy($$.type,$1.type); strcpy($$.name,$1.name); $$.temp_arr
= $1.temp_arr; }
;

array: IDENTIFIER '[' expression ']' {
    check_declaration($1.name);
    char *id_type = get_type($1.name);
    if(id_type != NULL) strcpy($$.type, id_type);
    char temp[100] = "";
    $$.nd = mknode(NULL, NULL, $$.name, 0);
    is_array = 1;
    $$.temp_arr = temp_arr++;
    sprintf(icg[ic_idx++], "t%d = 4 * %s\n", temp_var++, $3.name);
    sprintf(icg[ic_idx++], "u%d = %s[t%d]\n", $$.temp_arr, $1.name, temp_var - 1);
}


return: RETURN { add('K');} value ';'  {
    $1.nd = mknode(NULL, NULL, "return", 0);
    $$.nd = mknode($1.nd, $3.nd, "ReturnStatement", 0);
    sprintf(icg[ic_idx++], BOLDYELLOW);
    sprintf(icg[ic_idx++], "\nRETURN %s\n\n", $3.name);
    sprintf(icg[ic_idx++], BOLDGREEN);
}
| { $$.nd = NULL; }
;


%%


int main() {
    extern FILE *yyin, *yyout;

    int p = -1;
    p = yyparse();
    if(p)
        printf("Parsing Successful\n");
    printf("\033[4mParse Tree\033[24m\n");
      printBT(head);
    printf("\n\n");

    printSymbolTable();

    printSemanticErrors();

      printIntermediateCode();
    return p;
}

int yyerror(const char *msg)
{
    extern int yylineno;
    printf("%sParsing Failed\nLine Number: %d,%s\n",BOLDRED,yylineno,msg);
    exit(0);
    return 0;
}

void printBTHelper(char* prefix, struct node* ptr, int isLeft) {
    if( ptr != NULL ) {
        printf("%s",prefix);
        if(isLeft) { printf("├──"); }
             else { printf("└──"); }
        if(ptr->value != 0) {
            printf("%s",ptr->token);
            printf(BOLDMAGENTA);
            printf(" (Value -> %0.1f)",ptr->value);
            printf(RESET);
        }
        else {
            printf("%s",ptr->token);
        }
             printf("\n");
             char* addon = isLeft ? "│   " : "    ";
      int len2 = strlen(addon);
```

```
        int len1 = strlen(prefix);
        char* result = (char*)malloc(len1 + len2 + 1);
        strcpy(result, prefix);
        strcpy(result + len1, addon);
                printBTHelper(result, ptr->left, 1);
                printBTHelper(result, ptr->right, 0);
        free(result);
    }
}

void printBT(struct node* ptr) {
        printf("\n");
    printBTHelper("", ptr, 0);
}

struct node* mknode(struct node *left, struct node *right, char *token, float value) {

        struct node *newnode = (struct node *)malloc(sizeof(struct node));
        char *newstr = (char *)malloc(strlen(token)+1);
        strcpy(newstr, token);
        newnode->left = left;
        newnode->right = right;
        newnode->token = newstr;
        newnode->value = value;
        return(newnode);
}

void printSymbolTable(){
        printf("\033[4mSymbol Table\033[24m\n");

        int i=0;

        for(i=0; i<count; i++) {
                printf("Line    No:\033[1m\033[34m%d\033[0m    \t\033[1m\033[32m%s\033[0m
\t\t\t\033[1m\033[35m%s\033[0m           \t\t\033[1m\033[34m%s\033[0m           \t\n",
symbol_table[i].line_no,       symbol_table[i].data_type,       symbol_table[i].type,
symbol_table[i].id_name);
        }

        for(i=0;i<count;i++) {
                free(symbol_table[i].id_name);
                free(symbol_table[i].type);
        }
        printf("\n\n");
}


void printSemanticErrors() {
        printf("\033[4mSemantic Errors\033[24m\n");
        if(sem_errors>0) {
          printf(BOLDRED);
                printf("Semantic analysis completed with %d errors\n", sem_errors);
                for(int i=0; i<sem_errors; i++){
                    printf("%s", errors[i]);
                }
        } else {
                printf("Semantic analysis completed with no errors");
        }
}

void printIntermediateCode() {
        printf("\n\n");
    printf(RESET);
        printf("\033[4mIntermediate Code Generation (Before Optimisation)\033[24m\n");
    printf(BOLDGREEN);
    printf("\n");
        for(int i = 0; i < ic_idx; i++) {
                printf("%s", icg[i]);
        }

    FILE* icgFile = fopen("icg_base.txt", "w");
    fprintf(icgFile, "%s", BOLDGREEN);
    fprintf(icgFile, "\n");
    for(int i = 0; i < ic_idx; i++) {
        fprintf(icgFile, "%s", icg[i]);
    }
    fclose(icgFile);

        printf("\033[0m\033[4mIntermediate         Code         Generation         (After
Optimisation)\033[24m\n");
}

float get_value(char *var){
    for(int i=0; i<count; i++) {
        if(!strcmp(symbol_table[i].id_name, var)) {
            return symbol_table[i].value;
        }
    }
    return 0;
}

void set_value(char *var, float value){
    for(int i=0; i<count; i++) {
        if(!strcmp(symbol_table[i].id_name, var)) {
            symbol_table[i].value = value;
            return;
        }
    }
}

int search(char *type) {
        int i;
        for(i=count-1; i>=0; i--) {
                if(strcmp(symbol_table[i].id_name, type)==0) {
                        return -1;
                        break;
```

```
              }
        }
        return 0;
}

int check_declaration(char *c) {
    q = search(c);
    if(!q) {
            sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before
usage\n", countn+1, c);
              sem_errors++;
              return 0;
    }
        return 1;
}

void check_return_type(char *value) {
        char *main_datatype = get_type("main");
        char *return_datatype = get_type(value);
        if((!strcmp(main_datatype, "int") && !strcmp(return_datatype, "CONST")) ||
!strcmp(main_datatype, return_datatype)){
              return ;
        }
        else {
              sprintf(errors[sem_errors], "Line %d: Return type mismatch\n", countn+1);
              sem_errors++;
        }
}

int check_types(char *type1, char *type2){
        // declaration with no initialization
        if(!strcmp(type2, "null"))
              return -1;
        // both datatypes are same
        if(!strcmp(type1, type2))
              return 0;
        // both datatypes are different
        if(!strcmp(type1, "number") && !strcmp(type2, "string"))
              return 1;
        if(!strcmp(type1, "string") && !strcmp(type2, "number"))
              return 2;
        if(!strcmp(type1, "number") && !strcmp(type2, "boolean"))
              return 3;
        if(!strcmp(type1, "boolean") && !strcmp(type2, "number"))
              return 4;
        if(!strcmp(type1, "boolean") && !strcmp(type2, "string"))
              return 5;
        if(!strcmp(type1, "string") && !strcmp(type2, "boolean"))
              return 6;
}

char *get_type(char *var){
        for(int i=0; i<count; i++) {
              // Handle case of use before declaration
              if(!strcmp(symbol_table[i].id_name, var)) {
                    return symbol_table[i].data_type;
              }
        }
}


void add(char c) {
        if(c == 'I'){
              for(int i=0; i<10; i++){
                    if(!strcmp(reserved[i], strdup(id_name))){
                    sprintf(errors[sem_errors],  "Line  %d:  Variable  name  \"%s\"  is  a
reserved keyword\n", countn+1, id_name);
                        sem_errors++;
                        return;
                    }
              }
        }
        char *qtext;
        if(c =='I') qtext = id_name;
        else qtext = yytext;
    q=search(qtext);
        if(!q) {
              if(c == 'H') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Header");
                    count++;
              }
              else if(c == 'K') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("N/A");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Keyword\t");
                    count++;
              }
              else if(c == 'I') {
                    symbol_table[count].id_name=strdup(id_name);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Variable");
                    count++;
              }
              else if(c == 'C') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("Value");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Constant");
                    count++;
              }
              else if(c == 'F') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Function");
                    count++;
             }
             else if(c == 'N') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("number");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("NumberLiteral");
                    count++;
             }
    }
    else if(c == 'I' && q) {
           sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not 
allowed\n", countn+1, id_name);
             sem_errors++;
    }
}

void insert_type() {
      strcpy(type, yytext);
}

void store_name() {
      strcpy(id_name, yytext);
}
```

V. Sample Input Program and Terminal Output

Input Program (TypeScript File)

```
// Ignored
/* Ignored */
import { importedValue1, importedValue2 } from "../Syntax Analyser/input";

function trialFunction(param1: number, param2: string) {
    let valueNumber: number = 3;
    const valueBoolean: boolean = true;
    var valueExpression: number = 5 * 6 + 7 * 4;

    if (valueNumber > valueExpression && 3 > 4) {
        console.log("Hello");
        valueNumber = valueNumber + 5;
    } else {
        console.log(valueBoolean);
        valueNumber = valueNumber - 2;
    }
    while (valueNumber < 5 || 3 < 4) {
        valueExpression = valueExpression + 7;
        while (true) {
            valueNumber = valueNumber * 2;
        }
        valueNumber = valueNumber + 1;
    }

    return valueNumber;
}
```

Terminal Output : Annotated Parse Tree

```
Parse Tree
└──Program
    ├──ImportList
    │   └──ModuleList
    │       ├──importedValue1
    │       └──importedValue2
    └──Scope
        ├──Function
        │   └──trialFunction
        │       ├──FunctionBody
        │       │   ├──Scope
        │       │   │   ├──Initialisation
        │       │   │   │   ├──valueNumber (Value -> 3.0)
        │       │   │   │   └──3 (Value -> 3.0)
        │       │   │   └──Scope
        │       │   │       ├──Initialisation
        │       │   │       │   ├──valueBoolean
        │       │   │       │   └──true
        │       │   │       └──Scope
        │       │   │           ├──Initialisation
        │       │   │           │   ├──valueExpression (Value -> 58.0)
        │       │   │           │   └──+ (Value -> 58.0)
        │       │   │           │       ├──* (Value -> 30.0)
        │       │   │           │       │   ├──5 (Value -> 5.0)
        │       │   │           │       │   └──6 (Value -> 6.0)
        │       │   │           │       └──* (Value -> 28.0)
        │       │   │           │           ├──7 (Value -> 7.0)
        │       │   │           │           └──4 (Value -> 4.0)
        │       │   │           └──Scope
        │       │   │               ├──conditionalBranch
        │       │   │               │   ├──if
        │       │   │               │   │   ├──AND
        │       │   │               │   │   │   ├──>
        │       │   │               │   │   │   │   ├──valueNumber (Value -> 3.0)
        │       │   │               │   │   │   │   └──valueExpression (Value -> 58.0)
        │       │   │               │   │   │   └──>
        │       │   │               │   │   │       ├──3
        │       │   │               │   │   │       └──4
        │       │   │               │   │   └──Scope
        │       │   │               │   │       ├──ConsoleLog
        │       │   │               │   │       │   └──"Hello"
        │       │   │               │   │       └──Scope
        │       │   │               │   │           ├──=
        │       │   │               │   │           │   ├──valueNumber (Value -> 8.0)
        │       │   │               │   │           │   └──+ (Value -> 8.0)
        │       │   │               │   │           │       ├──valueNumber (Value -> 3.0)
        │       │   │               │   │           │       └──5 (Value -> 5.0)
        │       │   │               │   │           │
        │       │   │               │   │           └───EndOfScope
        │       │   │               │   └───else
        │       │   │               │       ├───Scope
        │       │   │               │       │   ├───ConsoleLog
        │       │   │               │       │   │   └───valueBoolean
        │       │   │               │       │   └───Scope
        │       │   │               │       │       ├───=
        │       │   │               │       │       │   ├───valueNumber (Value -> 10.0)
        │       │   │               │       │       │   └───- (Value -> 10.0)
        │       │   │               │       │       │       ├───valueNumber (Value -> 8.0)
        │       │   │               │       │       │       └───2 (Value -> 2.0)
        │       │   │               │       │       └───EndOfScope
        │       │   │               │       └───EndOfConditional
        │       │   │               └───Scope
        │       │   │                   ├───while
        │       │   │                   │   ├───OR
        │       │   │                   │   │   ├───<
        │       │   │                   │   │   │   ├───valueNumber (Value -> 10.0)
        │       │   │                   │   │   │   └───5
        │       │   │                   │   │   └───<
        │       │   │                   │   │       ├───3
        │       │   │                   │   │       └───4
        │       │   │                   │   └───Scope
        │       │   │                   │       ├───=
        │       │   │                   │       │   ├───valueExpression (Value -> 65.0)
        │       │   │                   │       │   └───+ (Value -> 65.0)
        │       │   │                   │       │       ├───valueExpression (Value -> 58.0)
        │       │   │                   │       │       └───7 (Value -> 7.0)
        │       │   │                   │       └───Scope
        │       │   │                   │           ├───while
        │       │   │                   │           │   └───Scope
        │       │   │                   │           │       ├───=
        │       │   │                   │           │       │   ├───valueNumber (Value -> 20.0)
        │       │   │                   │           │       │   └───* (Value -> 20.0)
        │       │   │                   │           │       │       ├───valueNumber (Value -> 10.0)
        │       │   │                   │           │       │       └───2 (Value -> 2.0)
        │       │   │                   │           │       └───EndOfScope
        │       │   │                   │           └───Scope
        │       │   │                   │               ├───=
        │       │   │                   │               │   ├───valueNumber (Value -> 21.0)
        │       │   │                   │               │   └───+ (Value -> 21.0)
        │       │   │                   │               │       ├───valueNumber (Value -> 20.0)
        │       │   │                   │               │       └───1 (Value -> 1.0)
        │       │   │                   │               └───EndOfScope
        │       │   │                   └───EndOfScope
        │       │   └───ReturnStatement
        │       │       ├───return
        │       │       └───valueNumber (Value -> 21.0)
        │       └───ParameterList
        │           ├───Parameter
        │           │   ├───number
        │           │   └───param1
        │           └───Parameter
        │               ├───string
        │               └───param2
        └───EndOfScope
```

Terminal Output : Generated Symbol Table

```
LINE NUMBER     DATATYPE                  TYPE                    SYMBOL
──────────────────────────────────────────────────────────────────────────────────────

Line No:5                               Function                function
Line No:5                               Variable
Line No:5       number                  Variable                param1
Line No:5       string                  Variable                param2
Line No:6       N/A                     Keyword                 let
Line No:6       number                  Variable                valueNumber
Line No:6       number                  NumberLiteral           3
Line No:7       N/A                     Keyword                 const
Line No:7       boolean                 Variable                valueBoolean
Line No:7       N/A                     Keyword                 true
Line No:8       N/A                     Keyword                 var
Line No:8       number                  Variable                valueExpression
Line No:8       number                  NumberLiteral           5
Line No:8       number                  NumberLiteral           6
Line No:8       number                  NumberLiteral           7
Line No:8       number                  NumberLiteral           4
Line No:10      N/A                     Keyword                 if
Line No:11      N/A                     Keyword                 console.log
Line No:11      Value                   Constant                )
Line No:13      N/A                     Keyword                 else
Line No:15      number                  NumberLiteral           2
Line No:18      N/A                     Keyword                 while
Line No:23      number                  NumberLiteral           1
Line No:26      N/A                     Keyword                 return
```

Terminal Output : Generated Semantic Errors

```
Semantic analysis completed with no errors
```

Terminal Output : Three Address Code

```
FUNCTION DEFINITION trialFunction
PARAM param2
PARAM param1

valueNumber = 3
valueBoolean = true
t0 = 5 * 6
t1 = 7 * 4
t2 = t0 + t1
valueExpression = t2

if valueNumber > valueExpression
GOTO S8
GOTO L1

LABEL S8:
if 3 > 4
GOTO L0
GOTO L1

LABEL L0:
t3 = valueNumber + 5
valueNumber = t3
JUMP TO L2

LABEL L1:
t4 = valueNumber - 2
valueNumber = t4

LABEL L2:

LABEL L3:

if valueNumber < 5
GOTO L4
GOTO S30

LABEL S30:
if 3 < 4
GOTO L4
GOTO L7

LABEL L4:
t5 = valueExpression + 7
valueExpression = t5

LABEL L5:
t6 = valueNumber * 2
valueNumber = t6
JUMP TO L0

LABEL L6:
t7 = valueNumber + 1
valueNumber = t7
JUMP TO L3

LABEL L7:

RETURN valueNumber
```

Optimized Three Address Code

```
       trialFunction
        /     |     \
    param2 param1  (valueBoolean=true)
```

```
        |       |
  (valueNumber=3) (t0=5*6)
           \      /
           (t2=t0+t1)
              |
           (valueExpression=t2)
               |
           +------+-------+
           |              |
     valueNumber > valueExpression
     (GOTO S8)         (GOTO L1)
     /     \             /     \
LABEL S8  LABEL L1   t4 = valueNumber - 2
   |        |         valueNumber = t4
   |        |            |
(3>4)   (GOTO L0)     LABEL L2
   |        |              |
LABEL L0   LABEL L2   LABEL L3
   |        |              |
  t3 = valueNumber+5   valueNumber < 5
  valueNumber = t3     (GOTO L4)
   |         |              |
JUMP TO L2  LABEL L3    LABEL S30
            |              |
          (valueNumber<5) (3<4)
            |              |
         (GOTO L4)      (GOTO L7)
            |              |
         LABEL L4       LABEL L7
            |              |
       t5 = valueExpression+7
       valueExpression = t5
            |
        LABEL L5
            |
       t6 = valueNumber*2
       valueNumber = t6
            |
        JUMP TO L0
            |
         LABEL L6
            |
       t7 = valueNumber + 1
       valueNumber = t7
            |
         JUMP TO L3
            |
         RETURN valueNumber
```

VI. Various Input Programs and their Terminal Output

Input Program (Bubble Sort)

```
function bubbleSort(arr: number[]) {
    let n: number = 7;
    for (let i: number = 0; i < n; i++) {
        for (let j: number = 0; j < n; j++) {
            if (arr[j] > arr[j + 1]) {
                let temp: number = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}

let inputArray: number[] = [5, 6, 7, 10, 2, 8, 1];
bubbleSort(inputArray);
```

Terminal Output : Generated Intermediate Code

```
Intermediate Code Generation (Before Optimisation)
PARAM arr
n = 7
i = 0

LABEL L0:

if i < n
GOTO L1
GOTO L8

LABEL L1:
j = 0

LABEL L2:

if j < n
GOTO L3
GOTO L7

LABEL L3:
t2 = 4 * j
u0 = arr[t2]
t3 = j + 1
t4 = 4 * t3
u1 = arr[t4]

if arr > arr
GOTO L4
GOTO L5

LABEL L4:
```

```
t5 = 4 * j
u2 = arr[t5]
temp = arr
t6 = 4 * j
u3 = arr[t6]
t7 = j + 1
t8 = 4 * t7
u4 = arr[t8]
u3 = u4
t9 = j + 1
t10 = 4 * t9
u5 = arr[t10]
u5 = temp
JUMP TO L6

LABEL L5:

LABEL L6:
JUMP TO L2

LABEL L7:
JUMP TO L0

LABEL L8:

RETURN arr

inputArray = [1, 8, 2, 10, 7, 6, 5]

FUNCTION CALL bubbleSort
```

Terminal Output : Optimised Intermediate Code

```
Intermediate Code Generation (After Optimisation)
PARAM arr
n = 7
i = 0

LABEL L0:

if i < n
GOTO L1
GOTO L8

LABEL L1:
j = 0

LABEL L2:

if j < n
GOTO L3
GOTO L7
```

```
LABEL L3:
t2 = 4 * j
u0 = arr[t2]
t3 = j + 1
u1 = arr[t4]
if arr > arr
GOTO L4
GOTO L5

LABEL L4:
t5 = t2
u2 = arr[t5]
temp = arr
u3 = arr[t6]
t7 = t3
u4 = arr[t8]
u3 = u4
t9 = t3
u5 = arr[t10]
u5 = temp
JUMP TO L6

LABEL L5:

LABEL L6:
JUMP TO L2

LABEL L7:
JUMP TO L0

LABEL L8:

RETURN arr
inputArray = [1, 8, 2, 10, 7, 6, 5]

FUNCTION CALL bubbleSort
```