NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI

CSPC62

COMPILER DESIGN

TOPIC : Compiler for Base Typescript

LAB REPORT - 4

<u>Sub Topic: Intermediate Code Generation</u>

DONE BY

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

Index

Phase 4: Intermediate Code Generation

I. Three Address Code

Three Address Code (often abbreviated as **TAC** or **3AC**) is an **intermediate code** used by compilers to implement code-improving transformations. Each instruction has at most 3 references (2 for operands and 1 for the result) and is usually a combination of assignment and a binary operator. The value computed at each instruction is then stored in a temporary variable generated by the compiler. For example:

`a := b op c`

Where a, b, and c are used to represent memory addresses.
These instructions translate more easily to assembly language, and make it easier to detect common sub-expressions for the shortening of code.

The operands are most likely not concrete memory addresses or processor registers, but rather symbolic addresses to be translated into the actual addresses during the register allocation. Additionally, operand names are numbered sequentially as this code is typically generated by the compiler.
In this example, the expression to calculate one solution to the Quadratic Equation is:

```
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

This can be broken down into several smaller ones:

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

Representation of the Three Address Code

1. **Quadruples**
   Consists of 4 fields - op, arg1, arg2 and result - where op denotes the operator, arg1 and arg2 denote the two operands, and result is used to store the result of the expression. While this form of TAC is easy to rearrange for global optimization and quickly access the value of temporary variables using the symbol table, it increases the overall time and space complexity.

2. **Triples**
   Consists of only 3 fields to implement the TAC - op, arg1, arg2 - where op denotes the operator, and arg1 and arg2 denote the two source operands. The results of the respective sub-expressions are denoted by the position of the expression. As no extra temporary variable is used to represent the operation, a pointer is used to refer to a triple's value when needed. This makes it difficult to rearrange code and optimize as the temporaries are implicit.

3. **Indirect Triples**
   This representation makes use of a pointer to the listing of all references to the computations that are made separately and stored. Similar in utility to quadruple representation but requires less space and the temporaries are implicit and make it easier to rearrange code.

II. Directed Acyclic Graph for Three Address Code

The Directed Acyclic Graph (DAGs) is a directed graph, consisting of vertices and edges, with no directed cycles i.e. each edge is directed from one vertex to another, such that they never form a closed loop. A directed graph is said to be a DAG if and only if it can be topologically sorted - an ordering of the vertices into a sequence, such that for every edge, the start vertex of the edge appears earlier in the sequence than the ending vertex of that edge.

A DAG can be used to represent the control flow and data dependencies between various blocks of code of a program. It is hence used as an intermediate representation to facilitate program optimization and transformation code via common sub-expression elimination and loop invariant code motion. A DAG is the three-address code generated as the result of intermediate code generation.

DAGs in Compiler Design

1. **Representation of Expressions**
   Each node in the DAG is used to represent a sub-expression, and each edge represents the dependencies between sub-expressions. This enables the compiler to perform optimizations such as constant sub-expression elimination and constant folding.

2. **Representation of Control Flow of the Program**
   Each node represents a basic block of code, and each edge represents a control flow transfer between the blocks. This enables the compiler to perform optimizations such as dead code elimination and loop unrolling.

Characteristics of the DAG

1. **Leaves:** Each have a unique identifier to represent variable names or constants.
2. **Interior/Intermediate Nodes:** The interior nodes are labelled with the operator symbol.
3. **String of Identifiers:** Nodes are given a string of identifiers to use as labels for storing the computed value.

III. Intermediate Code Generation

The **final phase of the front end** of the compiler is the Intermediate Code Generator. In this phase, the source program in translated into an independent intermediate code, which is then used to generate the target (machine) code by the back end of the compiler.

Typically, **multiple passes** are required over the various intermediate forms as it is easier to apply the many algorithms for the code optimizations one at a time. This is because the input to one optimization relies on the output/processing by another optimization. Hence, this phase facilitates the creation of a compiler that can target multiple architectures.

The input to the code generator typically consists of the parser tree or the abstract syntax tree. This tree is then converted into a linear sequence of instructions, usually in an intermediate language such as the three-address code.

Thus, the Intermediate Code Generation phase facilitates code optimization and reuse, leading to the improved performance and efficiency of the generated code. Being closer to the source code, it makes debugging and the implementation of the input code easier, however, at the cost of increased complexity of the compiler. Lastly, intermediate code is platform-independent, and can be translated into the machine code/byte code for any platform.

a. Defining the Structures and Additional Variables Required

In order to implement Intermediate Code Generation, two additional structures called `treeNode3 and treeNode4` with the following details are created.

```
struct var_name3 {
    char name[100];
    char type[8];
    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;
```

`condition` and `M` are declared as type `treeNode3 and treeNode4` respectively.

```
%type  <treeNode3> condition
%type  <treeNode4> M
```

The following buffers are created:

1. **icg:** To store the intermediate TAC instructions generated for the body of the program.
2. **func_buff:** To store the function definitions.
3. **func_params:** To store the various parameters required for each function.

```
char icg[100][200];
char func_buff[200];
char func_params[20][200];
```

Additionally, the following variables are declared to keep track of the temporary variables used in the TAC, counter for labels, to keep track of the index in the icg and the func_params buffers, and a flag to check if the code block is a `loop` or `if-else`.

```
int label = 0;
int ic_idx = 0;
int param_idx = 0;
int temp_var = 0;
int is_loop = 0;
```

b. Implementing Backpatching

Boolean expressions are usually translated using jump statements as this is most convenient for optimization. However, more than a single pass may be needed to generate code for boolean expressions and control flow in bottom-up parsers. When translating forward jumps, we do not know the numerical address of the label to be branched to.

Backpatching is the method of generating a sequence of branching statements where the addresses of the jumps (labels) are temporarily left unspecified. For each boolean expression `E`, two synthesized attributes - lists - are maintained `tlist` and `flist` , to store the list of the jump statements appearing in the translation of E. When the code for E is generated, the addresses of the jumps corresponding to the values true and false are temporarily left unspecified and placed in the lists `E.tlist` and `E.flist` respectively.

A marker non-terminal `M` is created to capture the numerical address of the statement we want to branch to.

```
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);
 };
```

The `condition` block has three fields - `if_body`, `else-body` and `after_else_body` - to store the labels where the control is to be redirected to based on the result of the given condition. The `condition` production is thus modified to store details regarding the intermediate code, while the actual TAC instructions are stored in the `icg` buffer. The variable `is_loop` pis used to distinguish between the loop (for and while) block and the if-else block.

```
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;
}
| value relop value {
      $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
    char ifstt[400];
    if(is_loop) {
        $$.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;
    }
    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; }
;
```

C. Printing the Intermediate Code for Various Code Blocks

The Intermediate Code is generated and stored in the `icg` buffer for various code blocks described as follows:

i) For Loop

```
body: FOR { add('K'); is_loop = 1;}'(' statement ';' condition ';' statement ')' '{'
body '}' {
      struct node *temp = mknode($6.nd, $8.nd, "Condition", 0);
    struct node *temp2 = mknode($4.nd, temp, "Initialisation", 0);
    $$.nd = mknode(temp2, $11.nd, $1.name, 0);
      sprintf(icg[ic_idx++], "%s", buff);
    sprintf(icg[ic_idx++], BOLDMAGENTA);
      sprintf(icg[ic_idx++], "JUMP TO %s\n", $6.if_body);
    sprintf(icg[ic_idx++], BOLDBLUE);
      sprintf(icg[ic_idx++], "\nLABEL %s:\n", $6.else_body);
    sprintf(icg[ic_idx++], BOLDGREEN);
};
```

ii) While Loop

```
body: 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);
    }
};
```

iii) If-Else Block

```
body: 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);
};
```

iv) Functions

```
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);
    sprintf(func_buff, "FUNCTION DEFINITION %s\n", $3.name);
}
;
```

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;

    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];
    char func_buff[200];
    char proc_buff[200];
    char func_params[20][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;
        char type[8];
        float value;
    } treeNode2;

      struct var_name3 {
            char name[100];
        char type[8];
            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 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
%type  <treeNode2> init expression value number term factor base exponent function
procedure
%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 '}' {
    $$.nd = mknode($4.nd, $8.nd, $1.name, 0);
    sprintf(icg[ic_idx++], BOLDMAGENTA);
    sprintf(icg[ic_idx++], "JUMP TO L%d\n", $4.label_for_while_start);
    sprintf(icg[ic_idx++], BOLDGREEN);
    for(int i=0;i<$4.flistsize;i++){
        char temp[40];
        sprintf(temp, "\033[1m\033[35mGOTO L%d\n\033[1m\033[32m", label);
        strcat(icg[$4.flist[i]], temp);
    }
    sprintf(icg[ic_idx++], BOLDBLUE);
      sprintf(icg[ic_idx++], "\nLABEL L%d:\n",label++);
    sprintf(icg[ic_idx++], BOLDGREEN);
}
| FOR { add('K'); is_loop = 1;}'(' statement ';' condition ';' statement ')' '{' body
'}' {
      struct node *temp = mknode($6.nd, $8.nd, "Condition", 0);
    struct node *temp2 = mknode($4.nd, temp, "Initialisation", 0);
    $$.nd = mknode(temp2, $11.nd, $1.name, 0);
      sprintf(icg[ic_idx++], "%s", buff);
    sprintf(icg[ic_idx++], BOLDMAGENTA);
      sprintf(icg[ic_idx++], "JUMP TO %s\n", $6.if_body);
    sprintf(icg[ic_idx++], BOLDBLUE);
      sprintf(icg[ic_idx++], "\nLABEL %s:\n", $6.else_body);
    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);
    sprintf(func_buff, "FUNCTION DEFINITION %s\n", $3.name);
}
;

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); }
;

parameterList: parameter ',' parameterList { $$.nd = mknode($1.nd, $3.nd,
"ParameterList", 0); sprintf(func_params[param_idx++], "PARAM %s\n", $1.name); }
| parameter { $$.nd = $1.nd; sprintf(func_params[param_idx++], "PARAM %s\n", $1.name);
}
| { $$.nd = $$.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(); }
;


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;
}
| value relop value {
      $$.nd = mknode($1.nd, $3.nd, $2.name, 0);
    char ifstt[400];
    if(is_loop) {
        $$.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;
    }
    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 '=' expression {
       if(check_declaration($1.name)) {
              $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);
             }
      }
}
| 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++;
}
;


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; }
| { sprintf($$.type, "%s", "null"); $$.nd = mknode(NULL, NULL, "NULL", 0);
strcpy($$.name, "NULL"); }
;


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); }
;


return: RETURN { add('K');} value ';'  {
    $1.nd = mknode(NULL, NULL, "return", 0);
    $$.nd = mknode($1.nd, $3.nd, "ReturnStatement", 0);
    sprintf(icg[ic_idx++], BOLDYELLOW);
    sprintf(icg[ic_idx++], "\nRETURN %s\n", $3.name);
}
| { $$.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\033[24m\n");
    printf(BOLDYELLOW);
    printf("%s", func_buff);
    for(int i = 0; i < param_idx; i++) {
        printf("%s", func_params[i]);
    }
    printf(BOLDGREEN);
    printf("\n");
      for(int i = 0; i < ic_idx; i++){
            printf("%s", icg[i]);
      }
}

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
```

DAG for the 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 (Function Invoking)

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

    return valueNumber;
}
```

Terminal Output : Generated Intermediate Code

```
Intermediate Code Generation
FUNCTION DEFINITION trialFunction
PARAM param2
PARAM param1

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

RETURN valueNumber

FUNCTION CALL trialFunction
```

DAG for the Three Address Code

```
                        trialFunction
                          /       \
                         /         \
                      PARAM      PARAM
                     param2      param1
                        |           |
                        v           v
                valueNumber=3  valueBoolean=true
                     |               |
                     v               v
                     t0=5*6          t1=7*4
                       \             /
                        \           /
                         \         /
                          \       /
                            t2=t0+t1
                                |
                                v
                      valueExpression=t2
                                |
                                v
                             RETURN
                          valueNumber
                                |
                                v
                         FUNCTION CALL
                          trialFunction
```

Input Program (If-Else Backpatching)

```
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;
}
```

Terminal Output : Generated Intermediate Code

```
Intermediate Code Generation

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:
```

DAG for the Three Address Code

```
        valueNumber = 3
              |
        +-----+-------+
        |             |
valueBoolean    t0 = 5*6
        |             |
    +---+---+     +---+---+
    |       |     |       |
    t1 = 7*4    t2 = t0+t1
                    |
              +-----+-----+
              |           |
 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
```

Input Program (While Loop Backpatching)

```
let valueNumber: number = 3;
const valueBoolean: boolean = true;
var valueExpression: number = 5 * 6 + 7 * 4;

while (valueNumber < 5 || valueBoolean == false) {
    valueExpression = valueExpression + 7;
    valueNumber = valueNumber + 1;
}
```

Terminal Output : Generated Intermediate Code

```
Intermediate Code Generation

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

LABEL L0:

if valueNumber < 5
GOTO L1
GOTO S11

LABEL S11:
if valueBoolean == false
GOTO L1
GOTO L2

LABEL L1:
t3 = valueExpression + 7
valueExpression = t3
t4 = valueNumber + 1
valueNumber = t4
JUMP TO L0

LABEL L2:
```

DAG for the Three Address Code

```
          valueNumber=3     valueBoolean=true
              |                   |
              v                   v
              t0=5*6              t1=7*4
                \                 /
                 \               /
                  \             /
                   \           /
                    \         /
                      t2=t0+t1
                        |
                        v
             valueExpression=t2
                        |
                        v
                       / \
                      /   \
                     /     \
                    /       \
           LABEL L0        LABEL S11
              |               /   \
              v              /     \
     if valueNumber<5    if valueBoolean==false
        |   |               |            |
        v   |               |            |
    GOTO L1 |               |            |
        |   \               |            |
        v    \              |            |
  GOTO S11  |          GOTO L1    GOTO L2
        |   \               |            |
        |    \              |            |
        |     \             |            |
        |      \            |            |
        |       \           |            |
        v        \          |            |
       LABEL L1 \_________ /             |
          |               |               |
          v               v               v
t3=valueExpression+7    t4=valueNumber+1  [END]
valueExpression=t3      valueNumber=t4
          |               |
          v               v
         JUMP TO L0      [END]
                          |
                          v
                       LABEL L2
```